Feature-Based Inspection and Control System

Thomas R. Kramer
John Horst
Hui Huang
Elena Messina
Frederick M. Proctor
Harry Scott

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

NISTIR 7098
March 8, 2004

Abstract

This report describes an architecture and software system for automatically performing process planning and control code generation for cutting and inspecting prismatic piece parts. This "Feature-Based Inspection and Control System" (FBICS) consists of seven processes joined by interprocess communication. FBICS starts with a feature-based description of a part. Planners are provided at three hierarchical control levels that consider, in turn: (1) an entire part, (2) work done in a single part fixturing, and (3) work done on a single part feature. FBICS implements the RCS architecture. For data handling, FBICS uses ISO 10303 (STEP) methodology, standards, and tools. FBICS includes a solid modeler and 3D interactive graphics. Control code is written in RS274 for machining and DMIS for inspection.

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

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.

Copyright

This publication was prepared by United States Government employees as part of their official duties and is, therefore, a work of the U.S. Government and not subject to copyright.

Keywords

ALPS, DMIS, feature, inspection, machining, planner, process plan, RCS, RS274, solid model, STEP.

CONTENTS

1.0 Introduction 1

2.0 FBICS Background 3

2.1 Real-time Control System (RCS) Architecture 3
2.2 Machine Control Languages 3
2.2.1 RS274 3
2.2.2 DMIS 3
2.3 Architecture Project 4
2.4 Enhanced Machine Controller Project 4
2.5 Earlier Work at NIST Leading to FBICS 4
2.5.1 Vertical Workstation 4
2.5.2 Off-Line Programming System 5
2.5.3 Feature-Based Control System 5
2.5.4 Next Generation Controller 5
2.5.5 ALPS 5
2.5.6 Tool Catalog 5
2.6 STEP 6

3.0 Overview of FBICS 7

3.1 Features and Parts 7
3.2 FBICS Architecture 8
3.2.1 Stand-Alone Architecture 9
3.2.2 Integrated Controllers 12
3.3 FBICS on Screen 14
3.3.1 Cell Controller Terminal Window 15
3.3.2 Work Controller Terminal Window 15
3.3.3 Task Controller Terminal Window 15
3.3.4 Task2 Process Terminal Window 15
3.3.5 Fbics_draw Graphic Display 15
3.3.6 Graphic Display Terminal Window 15
3.3.7 Modeler Terminal Window 15
3.4 File Formats 17
3.5 Two-Stage Planning 18
3.5.1 Flow of Planning 18

3.6 Saving and Saying “Do This Task” 18
3.7 FBICS Operation 19
3.7.1 Introductory Facts 20
3.7.2 FBICS Initialization 20
3.7.3 Planning Three Levels Deep for Machining with Inspection 21
3.7.4 Planning Three Levels Deep for Pure Inspection 22
3.7.5 Executing a Stage-one Cell-level Plan 23
3.7.6 Executing a Stage-two Cell Plan 24
3.7.7 Feedback from Inspection to Machining Planning 25
3.8 Length Units 25
3.8.1 Principles for Length Units 25
3.8.2 Implementation Details 27
3.9 Speed of Execution 28

4.0 FBICS Planning 29

4.1 Two-stage Planning 29
4.2 ALPS 30
4.2.1 Planning Stages 31
4.2.2 Preconditions 31
4.2.3 Method of Providing Extensions to the ALPS Schema 31
4.2.4 Use of FBICS_ALPS in FBICS 32
4.2.5 Split-Join Complexes in an ALPS Plan 34
4.2.6 Traversing an ALPS Plan 36
4.3 Shape Representation 40
4.3.1 Types of Shape Representation in FBICS 40
4.3.2 STEP AP 224 40
4.3.3 Part_out Shape 41
4.3.4 Part_in Shape 41
4.3.5 Feature Shape 42
4.3.6 Access Volume 42
4.3.7 Block Bodies 42
4.4 Inspection Planning 42
4.4.1 Reasons for Inspection 42
4.4.2 Deciding Which Features to Inspect and How Thoroughly 43
4.4.3 Deciding When to Inspect, During Machining 43
4.4.4 Selecting Inspection Points 44
4.4.5 The Inspection Point Selection Algorithm 45
4.4.6 Using AP 224 Features for Inspection 49

4.5 Human Input 50
4.6 Planning Issues....51
4.6.1 Plan Off-line or On-line 51
4.6.2 Maximum Off-Line Planning Depth....51
4.6.3 Distributed Planning Alternatives....52
4.6.4 Plan Validity 53

5.0 FBICS Interfaces.... 54

5.1 Modules and Processes 54
5.2 Types of Interface 54
5.3 FBICS APIs 55
5.4 Message Interfaces....55
5.4.1 Introduction....55
5.4.2 NML Messaging 56
5.4.3 Module Connections 56
5.4.4 Message Protocols 57
5.4.5 Message Types and Names....59
5.5 Data Interfaces 59
5.5.1 Cell Planner to Work Planner 60
5.5.2 Cell Planner to Modeler....60
5.5.3 Work Planner to Task Planner 60
5.5.4 Work Planner to Modeler....60
5.5.5 Task Planner to Modeler....61
5.5.6 Modeler to Graphic Display....61
5.5.7 Fbics_Task to Fbics_Task2....61
5.6 FBICS User Interfaces 61
5.6.1 Interface Modes 61
5.6.2 Time-outs 62

6.0 FBICS Verification 63

6.1 Verification Methods and Tools 63
6.2 Shape Verification Using Modeler 63
6.3 Other Part and Feature Checking....64
6.4 Machining Operation Verification....64
6.5 Other Checks....64

7.0 Cell Controller 65

7.1 Cell Controller Architecture 65

7.2 User Interface to the Cell Controller 66

7.2.1 Cell Controller Terminal Window 66
7.2.2 User Commands to the Cell Controller 67
7.2.3 Help 68
7.2.4 Quit 68
7.2.5 Init 68
7.2.6 Plan_part_machine 68
7.2.7 Plan_part_inspect 69
7.2.8 Run_part_plan1 70
7.2.9 Run_part_plan2 70
7.2.10 Exit 71
7.2.11 Work_manual 71

7.3 Cell Planner API 71

7.3.1 Cellpl_close_plan1 71
7.3.2 Cellpl_close_plan2 72
7.3.3 Cellpl_exit 72
7.3.4 Cellpl_init 72
7.3.5 Cellpl_make_op2 72
7.3.6 Cellpl_next_op1 72
7.3.7 Cellpl_next_op2 73
7.3.8 Cellpl_open_plan1 73
7.3.9 Cellpl_open_plan2 74
7.3.10 Cellpl_plan_part1_inspect 74
7.3.11 Cellpl_plan_part1_machine 75

7.4 Cell Planner Options 78

8.0 Work Controller 79

8.1 Work Controller Architecture 79

8.2 User Interface to the Work Controller 80

8.2.1 Work Controller Terminal Window 80
8.2.2 User Commands to the Work Controller 81
8.2.3 Help 82
8.2.4 Quit 82
8.2.5 Auto 82
8.2.6 Init 82
8.2.7 Plan_setup 83
8.2.8 Plan_inspect_setup 83
8.2.9 Run_setup_plan1 84
8.2.10 Run_setup_plan2 85

8.2.11 Exit 85
8.2.12 Task_manual 85
8.3 Command Messages to the Work Controller 85
8.3.1 WORK_EXIT_MSG 86
8.3.2 WORK_IDLE_MSG 86
8.3.3 WORK_INIT_MSG 86
8.3.4 WORK_MANUAL_MSG 86
8.3.5 WORK_PLAN_MSG 86
8.3.6 WORK_PLAN_INSP_MSG 86
8.3.7 WORK_RUN1_MSG 87
8.3.8 WORK_RUN2_MSG 87
8.4 Work Planner API 87
8.4.1 Workpl_close_plan1 87
8.4.2 Workpl_close_plan2 87
8.4.3 Workpl_exit 88
8.4.4 Workpl_init 88
8.4.5 Workpl_next_op1 88
8.4.6 Workpl_next_op2 88
8.4.7 Workpl_open_plan1 89
8.4.8 Workpl_open_plan2 89
8.4.9 Workpl_plan_inspect_setup1 89
8.4.10 Workpl_plan_setup1 90
8.4.11 Workpl_plan_setup2 92
8.5 Work Planner Options 93
8.5.1 Work Options 93
8.5.2 Shop Options 93

9.0 Task Controller 94

9.1 Task Controller Architecture 94
9.2 User Interface to the Task Controller 96
9.2.1 Task Controller Terminal Window 96
9.2.2 User Commands to the Task Controller 96
9.2.3 Help 97
9.2.4 Quit 97
9.2.5 Auto 97
9.2.6 Init 97
9.2.7 Open_setup 98
9.2.8 Generate_nc 98
9.2.9 Execute_nc 98
9.2.10 Generate_dmis 98
9.2.11 Execute_dmis 98
9.2.12 Close_setup 98

9.2.13 Exit 99
9.3 Command Messages to the Task Controller 99
9.3.1 TASK_CLOSE_MSG 99
9.3.2 TASK_EXEC_DMIS_MSG 99
9.3.3 TASK_EXEC_NC_MSG 99
9.3.4 TASK_EXIT_MSG 100
9.3.5 TASK_GEN_DMIS_MSG 100
9.3.6 TASK_GEN_NC_MSG 100
9.3.7 TASK_IDLE_MSG 100
9.3.8 TASK_INIT_MSG 100
9.3.9 TASK_MANUAL_MSG 100
9.3.10 TASK_OPEN_MSG 101
9.4 Task Planner API Functions 101
9.4.1 Taskpl_close_setup 101
9.4.2 Taskpl_exit 101
9.4.3 Taskpl_generate_dmis 101
9.4.4 Taskpl_generate_nc 102
9.4.5 Taskpl_init 102
9.4.6 Taskpl_open_setup 102
9.5 The FBICS RS274 NC Code Generator 102
9.5.1 Coolant_ex Generator 103
9.5.2 Counterboring_ex Generator 103
9.5.3 End_nc_ex Generator 104
9.5.4 Finish_mill_ex Generator 104
9.5.5 Nc_change_ex Generator 106
9.5.6 Start_nc_ex Generator 106
9.5.7 Twist_drilling_ex Generator 107
9.6 The FBICS DMIS Code Generator 108
9.6.1 Start_inspect_ex Generator 108
9.6.2 End_inspect_ex Generator 109
9.6.3 Tool-using Inspection Executable Operations 109
9.6.4 Inspect_change_ex Generator 110
9.6.5 Inspect_geometry_ex Generator 111
9.6.6 Locate_part_block_block_ex Generator 112
9.7 Command Messages to the Fbics_Task2 Process 113
9.7.1 TASK2_EXEC_DMIS_MSG 113
9.7.2 TASK2_EXEC_NC_MSG 113
9.7.3 TASK2_EXIT_MSG 113
9.7.4 TASK2_INIT_MSG 114
9.8 Fbics_Task2 Process 114
9.8.1 DMIS Interpreter API 114
9.8.2 RS274/NGC Interpreter API 115

9.9 Task Planner Options 116
9.9.1 Task Options 116
9.9.2 Shop Options 116

10.0 Modeler 117

10.1 Command and Status Messages of the Modeler 117
10.1.1 MODEL_ATTACH_MSG 117
10.1.2 MODEL_DETACH_MSG 117
10.1.3 MODEL_FUNCTION_MSG 117
10.1.4 MODEL_READY_MSG 118
10.2 Model Function Subtypes 118
10.2.1 block_intersects_part 119
10.2.2 bodies_intersect 119
10.2.3 circle_on_part 120
10.2.4 contains 120
10.2.5 copy_entity 120
10.2.6 c_shell_intersects_part 121
10.2.7 find_box 121
10.2.8 find_clear_length 122
10.2.9 find_tag 122
10.2.10 make_access 122
10.2.11 make_block 123
10.2.12 model_fixture 123
10.2.13 model_part_features 124
10.2.14 model_part_in 124
10.2.15 model_part_now 125
10.2.16 model_part_out 125
10.2.17 modify_part_now_file 126
10.2.18 modify_part_now_tag 126
10.2.19 point_on_part 126
10.2.20 rectangle_on_part 127
10.2.21 relocate_body 127
10.2.22 show_access 127
10.2.23 show_part_now 128
10.2.24 show_volume_file 128
10.2.25 show_volume_tag 128
10.2.26 unite_bodies 128

11.0 Graphic Display 129

11.1 Command Messages to the Graphic Display 129

11.1.1 DRAW_ACCESS_MSG 129
11.1.2 DRAW_FIXTURE_MSG 129
11.1.3 DRAW_FLUSH_MSG 129
11.1.4 DRAW_PART_IN_MSG 129
11.1.5 DRAW_PART_NOW_MSG 130
11.1.6 DRAW_PART_OUT_MSG 130
11.1.7 DRAW_VOLUME_MSG 130

11.2 Graphic Display User Interface 130

11.2.1 What the Scene Shows 131
11.2.2 Overview of User Interface Controls 133
11.2.3 Scene View Control 134
11.2.4 Object Visibility Control 135
11.2.5 Other Graphic Display Controls 135

12.0 Data Types 136

12.1 STEP Part 21 Formats 136

12.1.1 Introduction 136
12.1.2 Information Modeling In EXPRESS 138
12.1.3 Data Handling Tools 138
12.1.4 STEP AP 224 139
12.1.5 Options Introduction 139
12.1.6 Shop Options 139
12.1.7 Task Options 141
12.1.8 Work Options 144
12.1.9 FBICS_ALPS 145
12.1.10 Expressions 145
12.1.11 Tool Usage Rules 147
12.1.12 Cell-level Tasks 148
12.1.13 Work-level Inspection Tasks 149
12.1.14 Work-level Machining Tasks 149
12.1.15 Work-level Executable Operations 150
12.1.16 Setup 154
12.1.17 Stage-two Plans 154
12.1.18 Tool Catalog 154
12.1.19 Tool Inventory 155

12.2 DMIS Files 156

12.2.1 Introduction 156
12.2.2 Statements, Lines, Major Words, Minor Words 156
12.2.3 Programs and Files 156

12.2.4 Program Subunits 157
12.2.5 Geometric Features 157
12.2.6 Tolerances 157
12.2.7 Comments 158
12.3 RS274 Files 158
12.3.1 Numerical Control Programming Language RS274 158
12.3.2 The RS274/NGC Language 158
12.3.3 FBICS use of RS274 158
12.4 Graphics Files 159
12.5 File Names 159
12.5.1 Setup Files 159
12.5.2 Intermediate Workpiece Files 160
12.5.3 Process Plan Files 160
12.5.4 Feature Files 160
12.5.5 Task-level Executable Instruction and Code Files 160
12.5.6 Graphics Files 161

13.0 Strengths and Limitations 162

13.1 Strengths 162
13.2 Limitations 162

14.0 Software 164

14.1 Modularization 164
14.2 In-Line Documentation 164
14.3 Software Files 165
14.3.1 Handwritten C++ Code 165
14.3.2 Automatically Generated C++ Code 166
14.3.3 Object and Archive Files from Other ISD Projects 166
14.3.4 Commercial Software Libraries 166
14.3.5 Essential Data 166
14.3.6 Executables 167
14.3.7 Other 167
14.4 Error Handling 167
14.4.1 Automatic Generation of Error Software 167
14.4.2 Error Recovery 168

References 169

Appendix A An Example 173

A.1 Part_out Design File 175
A.2 Part_in 177
A.3 Intermediate Workpiece Shape 177
A.4 Cell-level Stage-one Plan 178
A.5 Cell-level Stage-two Plan 179
A.6 Setup File for First Setup 179
A.7 Setup File for Second Setup 180
A.8 Features File for First Setup 180
A.9 Features File for Second Setup 181
A.10 Work-level Stage-one Plan for First Setup 183
A.11 Work-level Stage-one Plan for Second Setup 183
A.12 Executable Operation and Code Files for First Setup 184
A.13 Executable Operation and Code Files for Second Setup 185
A.14 DMIS Output File 196
A.15 Graphics File for Part_in 198

Appendix B Other Sample Files 201

B.1 Shop Options 201
B.2 Task Options 201
B.3 Work Options 202
B.4 Tool Catalog 203
B.5 Tool Inventory 205
B.6 Tool Usage Rules 205

List of Figures

Figure 1. FBICS Stand-Alone Architecture. 10
Figure 2. FBICS Advanced Integrated Architecture 13
Figure 3. FBICS Screen Layout 16
Figure 4. FBICS Use of FBICS_ALPS 33
Figure 5. ALPS Plan Example 34
Figure 6. Split-join Complexes 35
Figure 7. ALPS Plan Traversal Example. 38
Figure 8. Point Counter 46
Figure 9. Point Placement 47
Figure 10. Inspection Point Selection Algorithm Implemented in C. 48
Figure 11. FBICS Cell Controller. 66
Figure 12. Cell Controller Help 68
Figure 13. FBICS Work Controller 80
Figure 14. Work Controller Help 82
Figure 15. FBICS Task Controller 95
Figure 16. Task Controller Help. 97
Figure 17. Graphic Display Controls 134
Figure 18. FBICS Expressions 146
Figure 19. Work-level Tasks 151
Figure 20. Work-level Executable Operations 153
Figure 21. In-Line Documentation Example 165
Figure 22. Part1 173

1 Introduction

This report describes an architecture for a Feature-Based Inspection and Control System (FBICS) for machining and inspecting mechanical piece parts, and an implementation of it. By “feature-based inspection and control”, we mean that a feature-based description of the shape of the object to be made is a principal input for machining and/or inspection. As used in FBICS, a feature is a volume whose shape is an instance of some member of a predefined set of types of shape — a hole or a pocket, for example.

FBICS is implemented in the Intelligent Systems Division (ISD) of the National Institute of Standards and Technology (NIST), employing the Real-time Control System (RCS) architecture developed in ISD.

FBICS focuses on machining using a 3-axis machining center and on inspection either in-process on the machining center or using a coordinate measuring machine with a touch probe. For machining, FBICS has been integrated with a Bridgeport machining center run by a controller built by the ISD. For inspection, FBICS has been integrated with a Cordax coordinate measuring machine run by a hierarchical control system built by the ISD.

FBICS has the following characteristics:

has tightly integrated open architecture,
uses hierarchical task decomposition and control,
has clearly defined modules, command interfaces, and data interfaces,
is architected to take full advantage of both computer capabilities and human capabilities,
takes full advantage of available data,
uses standard data representations and modeling languages,
is STEP-based, by using STEP standard information models, the EXPRESS modeling language, and STEP exchange files,
does automatic generative process planning,
generates RS274 NC code automatically for machining,
generates DMIS code automatically for inspection,
includes user preferences at all levels,
uses both off-line and on-line planning.

In the remainder of this report:

Section 2 gives the background of FBICS at the National Institute of Standards and Technology.
Section 3 is an overview of how FBICS works.
The next three sections are overviews of topics that apply to FBICS as a whole:
Section 4 is an overview of planning,
Section 5 is an overview of interfaces, and
Section 6 is an overview of verification.
The next five sections look at the five big pieces into which FBICS is divided:
Section 7 discusses the Cell Controller,
Section 8 discusses the Work Controller,
Section 9 discusses the Task Controller,

Section 10 discusses the Modeler, and
Section 11 discusses the Graphic Display.
Section 12 describes FBICS data types in moderate detail.
Section 13 presents FBICS strengths and limitations.
Section 14 describes the FBICS software.
Appendix A provides an example of FBICS in operation.

This report is intended primarily as a description of FBICS. While some discussion of issues is included, the report does not attempt to discuss issues in full detail and does not attempt to present what other researchers have written about the issues.

This report describes FBICS as it is in August, 2003, not as we might wish it were. Although a great deal has been done, some parts of the software are unfinished, some parts of the architecture beg to be changed, and dozens of types of functionality could be improved.

This report covers all aspects of FBICS. Other papers and reports on FBICS include (1) a brief overview of FBICS [Proctor2] (2) an overview in the context of an inspection system using FBICS [Messina], and (3) a more extensive overview [Kramer19].

2 FBICS Background

This section describes the background of the FBICS system.

2.1 Real-time Control System (RCS) Architecture

James Albus of the Intelligent Systems Division at the National Institute of Standards and Technology (NIST) has for many years spearheaded development of a control architecture known as the Real-time Control System (RCS).

Under the tenets of RCS, controllers are arranged in a hierarchy. Each controller, save one at the top, is subordinate to a single superior controller, and each controller may have several subordinates. Those at the bottom of the hierarchy have no subordinate controllers but control actuators. Controllers interact by superior controllers sending commands to subordinates, in response to which the subordinates perform actuation or send commands to their subordinates and then send status back. Each controller performs some type of real-time planning, which may range from selecting a pre-made plan without change to totally generative planning.

The field of discrete parts manufacture is amenable to RCS control, and FBICS conforms to the RCS architecture. In RCS, each controller includes components that perform sensory processing, world modeling, planning, job assignment, and execution. The emphasis in FBICS development has been on planning and world modeling. A world model is a set of data representing important facts about the domain with which a control system (or part of a control system) is dealing. World modeling is the function of maintaining this data and using it to provide information to other parts of the control system.

Many papers are available providing more detail on RCS, including [Albus1], [Albus2], [Albus3].

2.2 Machine Control Languages

Standard machine control languages exist: RS274 for machining and DMIS for inspection. Interpreters for these languages were developed at NIST independently of the FBICS project. Since the languages are standard and the interpreters were available, these two languages have been used in FBICS.

2.2.1 RS274

RS274 is a venerable programming language for numerically controlled machine tools. The most recent standard version, RS274-D, was completed in 1979. The NGC project (see Section 2.5.4) developed a specification for the RS274/NGC language, which has many capabilities beyond those of RS274-D. The most recent version of that specification was released in draft in 1994 by the National Center for Manufacturing Sciences. The NIST RS274/NGC interpreter uses the 1994 draft as the specification. Further RS274 details are given in Section 12.3.

2.2.2 DMIS

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 4.0, which was completed in 2001 [CAM-I]. The NIST DMIS interpreter uses the previous version, 3.0. Further DMIS details are given in Section 12.2.

2.3 Architecture Project

The Intelligent Systems Division (ISD) has a continuing project in RCS architecture development. James Albus is the Chief Architect for this project. One of us (Scott) has been the administrative leader of the project for several years. Another of us (Kramer) was a previous co-leader. This architecture project has provided much of the support for the development of FBICS.

The architecture project has provided support, as well, for the development of RCS controller templates by one of the authors (Huang). Controllers built using the templates have been used in a version of FBICS in which planning functions are integrated with inspection and actual machining. Integration of the system was accomplished under the supervision of another of the authors (Messina).

The architecture project also supported the development of a DMIS interpreter that could be used in ISD inspection projects. There are two documented versions of the DMIS interpreter [Kramer16], [Kramer17].

2.4 Enhanced Machine Controller Project

The Enhanced Machine Controller (EMC) project, conducted in ISD, is headed by one of the authors (Proctor). The primary objective of the EMC project is to build a testbed for evaluating application programming interfaces (APIs) for open-architecture machine controllers. FBICS is being used in the EMC project and is a component of the testbed. FBICS development has been supported by the EMC project.

In prior work, the EMC project built a machine tool controller and retrofitted a 4-axis machining center at General Motors with it [Proctor1]. That controller and variants of it are called “the EMC controller.” EMC controllers have been installed on several machining centers in commercial machine shops. The EMC controller incorporates an NC-program interpreter for programs written in the RS274 language, of which there are several documented versions [Kramer11], [Kramer12], [Kramer13], [Kramer14], [Kramer18].

2.5 Earlier Work at NIST Leading to FBICS

2.5.1 Vertical Workstation

The Vertical Workstation System (VWS) of the NIST Automated Manufacturing Research Facility, developed between 1986 and 1989, was a feature-based control system. Software for the system was written in Lisp.

In the VWS, a primitive CAD system was used with a design protocol [Kramer5] which constrained the user to design in terms of machining features. The operations for cutting the features were automatically defined and partially sequenced by a generative process planner [Kramer2]. A combined process plan traverser and NC code generator [Kramer4], controllers [Kramer1], and a feature recognition module [Kramer6] which could extract features of the required sort from a boundary representation were included. The individual components of this system were unsophisticated, (except for the circular arc spline fitter [Kramer7]) but they were very well integrated. Within a limited range of design, a part could be designed and cut within an hour using this system.

From experience with the Vertical Workstation, it became apparent that using machining features for design, while a good technique for a few special situations, is not a general solution for piece part manufacture. It is common to be able to machine a part more effectively (faster, cheaper, tighter tolerances, etc.) if machining features may be used which are not explicit in the design. Many other researchers have come to the same conclusion.

2.5.2 Off-Line Programming System

The Off-Line Programming System (OLPS) was an NC code generation system which was intended to be used in a larger system in which machining features are defined separately from the design [Kramer8]. The issues were studied [Kramer10], and a library of parametric machining features was defined for use with OLPS [Kramer9]. OLPS was developed between 1988 and 1990, but was never used as envisioned in a larger system. OLPS code was written in Lisp.

2.5.3 Feature-Based Control System

In 1995 and 1996 a prototype Feature-Based Control System was developed which included many of the elements of FBICS [Kramer15]. It served to show the feasibility of feature-based control, but did not include many key FBICS elements, such as generative planning, inspection, solid modeling, graphics, advanced process plan traversal, or the use of rules as data. It used two control levels where FBICS has three, since the need for a separate level to handle setups was not recognized at the time.

2.5.4 Next Generation Controller

In the late 1980's and early 1990's, the Department of Defense supported a "Next Generation Controller" (NGC) project. As part of ISD assistance to the NGC project, ISD prepared a report "NIST Support to the Next Generation Controller Program: 1991 Final Technical Report," [Albus4] containing a variety of suggestions. Appendix C to that report proposed three sets of commands for 3-axis machining, one set for each of three proposed hierarchical control levels. The suite proposed for the middle level of control evolved into the one used in FBICS for workstation-level tasks, described in Section 12.1.14. The suite proposed for the lowest control level evolved into a suite, known in the EMC project as the "canonical machining functions," for 3-axis to 6-axis machining [Proctor3]. This suite is used in FBICS in the NC code interpreter.

2.5.5 ALPS

The ALPS language (A Language for Process Specification) was developed at NIST by Steven Ray and Bryan Catron [Catron]. It is a general language for writing process plans for discrete operations. One of the authors (Kramer) built the first EXPRESS model of ALPS in 1990. Additions to ALPS, and EXPRESS models containing them, were made by the NIST Manufacturing Systems Integration (MSI) Project [Wallace]. Further details on ALPS are provided in Section 4.2 and Section 12.1.9.

The MSI project also developed resource concepts which have been used in FBICS.

2.5.6 Tool Catalog

As part of a Rapid Response Manufacturing program at NIST (part of a larger national effort), a manufacturing resource model for cutting tools and tooling components was developed under the

leadership of Kevin Jurrens. The requirements specification document [Jurrens] was translated into an EXPRESS model by William Burkett under contract to NIST. This model, with modifications, is being used in FBICS. Efforts for standardization of cutting tool data, based on the model, are in progress in Technical Committee 29 Working Group 34 of the International Organization for Standardization (ISO TC29/WG34). The developing standard is ISO 13399.

2.6 STEP

STEP (Standard for the Exchange of Product Model Data) is the common name for standard 10303 of the International Organization for Standardization (ISO). This standard is composed of individual documents known as STEP “Parts”. STEP Part 11 defines the EXPRESS data modeling language [ISO1]. An EXPRESS model definition is contained in one or more constructs called EXPRESS “schemas”. STEP Part 21 defines an exchange file format for transmitting instances of data which has been modeled in EXPRESS schemas [ISO2]. STEP also provides data models for various domains. The models fall in several classes. The class of model intended to be used is called an “Application Protocol” (AP).

3 Overview of FBICS

This section gives an overview of FBICS, describing what it is and how it works, without getting into details — those are covered in subsequent sections.

The primary purposes of FBICS are:

1. to demonstrate feature-based inspection and control in an open-architecture control system.
2. to serve as a testbed for solving problems in feature-based manufacturing, particularly the partitioning of manufacturing activities into separate activities and the definition of interfaces between activities.
3. to test the usability of STEP methods and models.

FBICS exists as (1) a stand-alone system using minimally functional controllers but fully functional planners, with simulated inspection or machining, and (2) as part of a loosely integrated inspection system using the same FBICS processes with a Cordax coordinate measuring machine controlled by an RCS controller hierarchy, and (3) as part of a tightly integrated machining system also with the same FBICS processes, but with an EMC controller for a machining center. Machining centers run using this last system include a Bridgeport 3-axis machining center and a 3-axis “mini-mill” machining center. In addition, stand-alone FBICS has been used off-line to write NC programs for a Hexapod machining center and used loosely integrated to drive simulated inspection on a separate (non-FBICS) system.

Software for all versions of FBICS is in the C++ language.

The two principal capabilities of FBICS are: to generate process plans automatically at each level of a control hierarchy, and to execute the plans to make and/or inspect piece parts.

The FBICS architecture is designed to allow for human participation at every significant step of the process. Elements of this include using appropriate data types at each interface between control levels, having an open file format for all data types passing across interfaces, having variable planning depth, and allowing either off-line or on-line planning. The user-friendly editors and interfaces required for effective human participation in FBICS activities, however, have not been built.

3.1 Features and Parts

There are many definitions of “feature”. Here we deal only with the definition used in FBICS. For FBICS, the general idea is that a feature represents a portion of the shape of a part having some relevance to one or more machining or inspection operations. More specifically, FBICS deals with features from a pre-defined library of stereotypical shapes, where the exact shape of a feature is determined by giving its type and the values of one or more parameters associated with that type of feature. The library of features used in FBICS is from STEP AP 224, discussed more deeply in Section 4.3.2. The shape of a part may be described in terms of features by giving a base shape and a list of features. The base shape is usually a block. Each feature is regarded as a closed solid which is removed from the base shape. The part shape is what results after all the features have been removed from the base shape.

In FBICS, a feature is related to a machining operation as follows: when the operation is finished, there must be no material remaining inside the feature, and the operation may remove no material

outside the feature. In solid modeling terms, the operation produces a boolean subtraction of the feature from the workpiece. For describing the shape of an entire part, a set of machining features is defined so that a boolean subtraction of the features from the original workpiece (stock or a partially finished part) will result in the desired final shape for the part.

A feature is not always the volume of material to be cut away by the operation. A feature might not be totally within material. The actual swept volume of the cutting tool in carrying out a machining operation (after finishing its approach) may be exactly the same as the feature, may be wholly contained in the feature, or may intersect the feature and extend outside it into air.

How features relate to inspection is described in Section 4.4.

FBICS focuses on parts whose shape can be readily described in terms of features. This includes all of what are usually called prismatic parts. The shape of the parts may be simple or complex. The parts and the materials from which they are made are expected to be rigid. Parts are assumed to made of typical industrial metals (iron, aluminum, brass, titanium, etc.) or other machinable materials (plastic, wax, graphite, etc.). The materials are assumed to be homogeneous enough that material structure (grain, crystal orientation, etc.) does not need to be considered.

This document refers repeatedly to parts at three phases:

1. The as-designed part, which is what the part should look like on the way out, after it has been processed. This will be called the “part_out”.
2. The part as it appears before processing starts. This will be called the “part_in”.
3. The part as it is right now, during processing. This will be called the “part_now”.

If a part is only being inspected, its shape does not change, so different versions of the part are not needed. For inspection only the part_now is used.

If a part is processed in several stages, there is a part_in and part_out for each stage. The part_out from one stage is the part_in for the next stage. At the end of a processing stage, the part_now should have the same shape as the part_out for that stage.

3.2 FBICS Architecture

The FBICS controller hierarchy is generally as has been described often in RCS literature. FBICS includes controllers at the RCS “Cell”, “Workstation”, and “Task” levels. Each controller runs in a separate process or processes (in the operating system sense). Processes may be on the same computer or different computers. Controllers communicate via a messaging system [Shackleford].

This hierarchy corresponds to the way many large machine shops actually run. The shop is divided into groups of machines, each called a cell. A cell is expected to be able to make and inspect a finished part from a starting workpiece. A cell includes several workstations. Each workstation has a principal machine and, possibly, auxiliary machines. Each machine can perform several types of task, each task is composed of several elementary moves, each elementary move is decomposed into primitive motions, and each primitive motion is controlled by a servo system.

Planning in FBICS follows the paradigm just described. The Cell Planner determines how many setups are required to make or inspect the part and what features to make or inspect in each setup. For each setup, the Work[1] Planner determines how each feature is to be made or inspected, the

1. For brevity, “workstation” is shortened to “work” here and in the remainder of this document.

tools that are required, and the order in which the operations are to be done. For each feature, the Task Planner determines tool paths for cutting or inspecting the feature.

In addition to the controllers, the FBICS architecture includes a Solid Modeling Server (Modeler, for short), a Data Repository, a Graphic Display, and a Communications Server. The Modeler works for the planners in the Cell, Work, and Task controllers. The Graphic Display is connected directly only to the Modeler, but graphics commands that run through the Modeler are available to those three planners. The Communications Server acts as a focal point for communications among the other processes.

3.2.1 Stand-Alone Architecture

The stand-alone FBICS architecture is shown in Figure 1.

In stand-alone FBICS, emphasis has been placed on the planning component of each controller. A clean, small, function call interface has been defined for each planner, to be used by the rest of the controller. In the stand-alone version, the rest of each controller includes only enough functionality to make the system run and to help with debugging during development.

Figure 1 shows only one subordinate for each superior controller, since that is how stand-alone FBICS is built. In general, in RCS, each superior controller may have several subordinates.

The “command message” arrows on Figure 1 show the direction of flow of commands. In every case (although no return arrows are drawn), a status message flows back to the issuer of the command. Almost all the status messages are simple, meaning “I did it,” or “I could not do it.” Status messages from the Solid Modeling Server may be more meaty (see Section 10). Many of them include a boolean yes/no answer to a question, some include a single numerical value, and others include several related values.

In the remainder of this document, names of command and status messages will be used without explaining what they mean, since the names are self-explanatory. The name of every message has three parts separated by underscores: (1) the name of a process, (2) the kind of message, and (3) the suffix `MSG`. If the kind of message is `READY`, the message is a status message and the process name refers to the sender; otherwise the message is a command message and the process name refers to the receiver. For example, a `TASK_READY_MSG` is a status message sent by the Fbics_Task process and a `TASK_INIT_MSG` is a command message received by Fbics_Task telling it to initialize itself. Command and status message names are set in `courier` type throughout this document.

The Communications Server is not shown on Figure 1, but all command and status messages (the solid-line arrows on the figure) flow through the Communications Server. Other flows on the figure do not use the Communications Server. Rather, they directly use the infrastructure provided by the computer, its operating system, and the network to which the computer is attached. The Communications Server also uses that infrastructure, as would be expected.

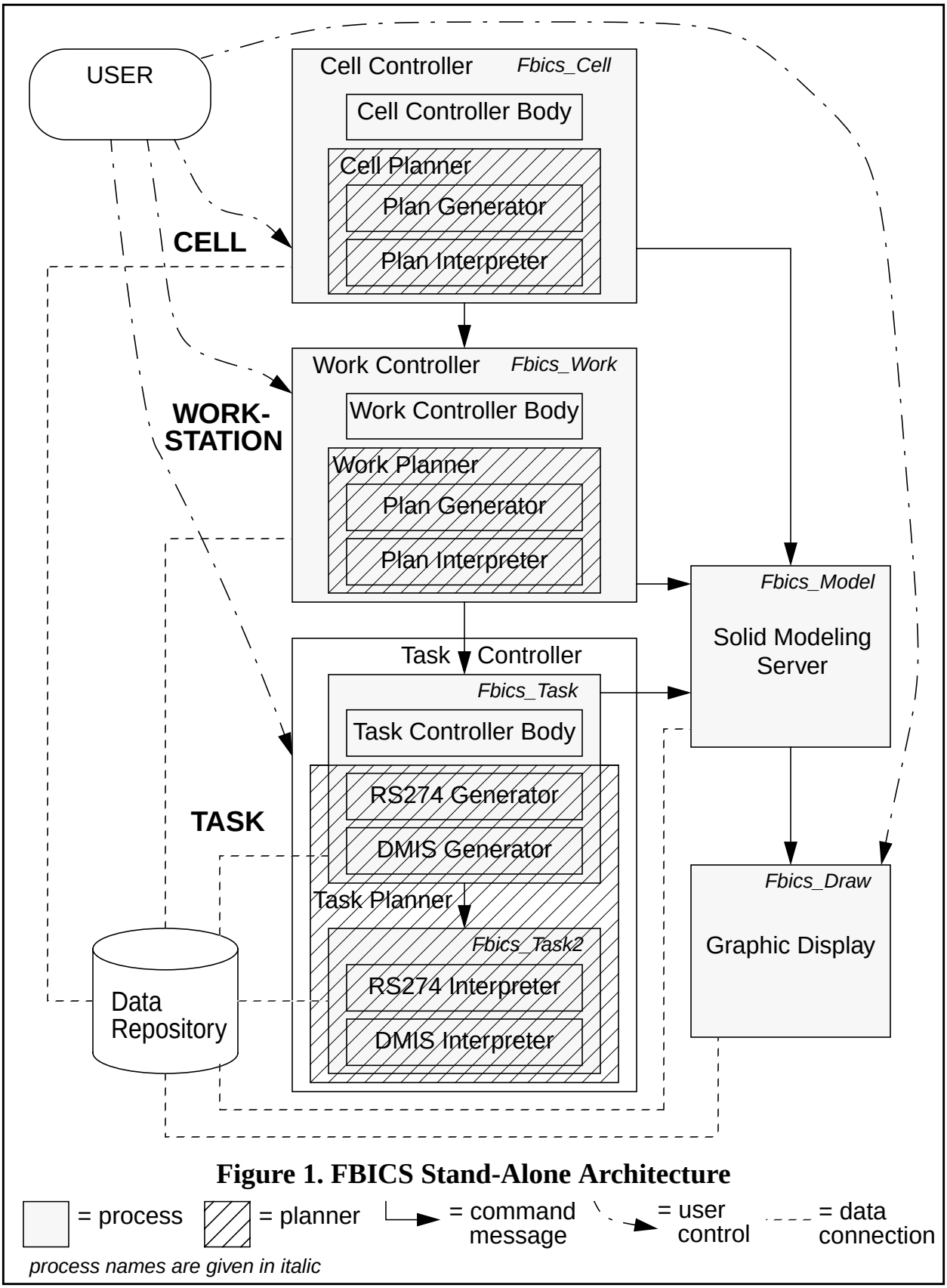


Figure 1. FBICS Stand-Alone Architecture

3.2.1.1 Cell Controller

The (top-level) Cell Controller is focused on making a part from a part blank (or from a partially machined workpiece) or on inspecting an entire previously made part. It is typical that several setups will be required to make or inspect a whole part. The Cell Controller can make a cell-level process plan for the part, can give planning commands to the Work Controller, can execute a cell-level process plan, and can give execution commands to the Work Controller.

3.2.1.2 Work Controller

The (second-level) Work Controller handles a single setup of the part (i.e., processing the part without moving it). The Work Controller focuses on transforming a workpiece from an incoming shape to an outgoing shape, possibly with intermittent inspection, or on inspecting those features that may be inspected in a single setup. The Work Controller can make a work-level process plan for one setup, can give planning commands to the Task Controller, can execute a work-level process plan, and can give execution commands to the Task Controller.

3.2.1.3 Task Controller

The (third-level) Task Controller focuses on making or inspecting single features. The Task Controller can make a task-level process plan for machining one feature or inspecting one feature. At the task level, a machining process plan is an NC code file (in the RS274 language), and an inspection plan is an inspection code file (in the DMIS language). The Task Controller can execute task-level plans.

The Task Planner is split between two processes: Fbics_Task and Fbics_Task2. The Fbics_Task2 process contains the parts of the planner that run during plan execution. During execution, the Fbics_Task process sends messages to the Fbics_Task2 process, telling it to run a file of RS274/NGC code or DMIS code. Splitting the planner between two processes was originally done for convenience. The split has been kept in stand-alone FBICS so that integrated systems are easy to build. In the existing integrated systems, messages that would be sent to the Fbics_Task2 process in stand-alone FBICS are redirected to a task-level controller that controls actual equipment.

The Fbics_Task2 process includes version three of the NIST DMIS interpreter, which is undocumented, but very similar to version two, documented in [Kramer17]. The Fbics_Task2 process also includes version three of the NIST RS274/NGC interpreter [Kramer18].

The messages Fbics_Task2 understands include `TASK2_EXEC_DMIS_MSG` and `TASK2_EXEC_NC_MSG`. In response to these messages, the appropriate interpreter reads the file and executes the instructions in the file. In the stand-alone version, this results in some degree of emulation of machining or inspection.

3.2.1.4 Solid Modeling Server

The Solid Modeling Server (Modeler) provides solid modeling services to its clients, which are the controllers. The Modeler maintains a separate view of the shape of things for each client. The types of service the Modeler provides include, for example, maintaining a model of the current shape of the part, faceting a model and telling the Graphic Display to show it, and determining if a candidate touch point for probing is present on the current part. The Parasolid solid modeler serves as the underlying modeling engine.

3.2.1.5 Graphic Display

The Graphic Display shows 2D views of 3D objects on a color monitor, using the “movie camera” paradigm. The display is manipulable by the user in the usual ways: move around the object, zoom in or out, etc. The types of object which can be shown are: part_out, part_in, part_now, current feature, access volume, and fixture. Each object type may be displayed as a solid object, as a wire frame, or not at all — under immediate control of the user. The HOOPS graphics system is the underlying graphics engine.

3.2.1.6 Data Repository

FBICS uses a file system as the Data Repository. Data in the repository includes, for example, part designs, process plans, and user option files. A complete (and much longer) list of data types is provided in Section 12. During FBICS operation, data files are written by one process and read by others. Data files are used as a supplement to messaging for sending commands from one controller to another.

Using a file system as the Data Repository has been fully adequate. As FBICS matures it may become desirable to use a database system for the Data Repository, but the need has not yet arisen.

3.2.1.7 User Interfaces

As shown in Figure 1, stand-alone FBICS has user interfaces to the three controllers and to the Graphic Display. The user interface to the Graphic Display has the capabilities mentioned above and is totally mouse-driven (see Section 11.2). The stand-alone version’s user interfaces to the controllers are all simple text-based interfaces that allow the user to exercise the two principal functions of the controller: plan or execute. See Section 5 for a general discussion of the user interfaces. See Section 7.2, Section 8.2, and Section 9.2 for details about the individual interfaces.

3.2.2 Integrated Controllers

3.2.2.1 Loosely Integrated FBICS

Loosely integrated versions of FBICS have been built with all upper-level components as described in Section 3.2.1, except for Fbics_Task2. For a loosely integrated version, the Fbics_Task2 process is replaced by a fully functional RCS controller that includes an RS274/NGC interpreter and/or a DMIS interpreter. For an integration with inspection, the message interface between Fbics_Task and Fbics_Task2 is used unaltered. For an integration with machining, the interface is modified. The fully functional RCS controllers control actual hardware for cutting or inspection.

3.2.2.2 Advanced Integrated FBICS

An advanced integrated version of FBICS has yet to be built. Figure 2 shows the architecture planned for an advanced integrated version.

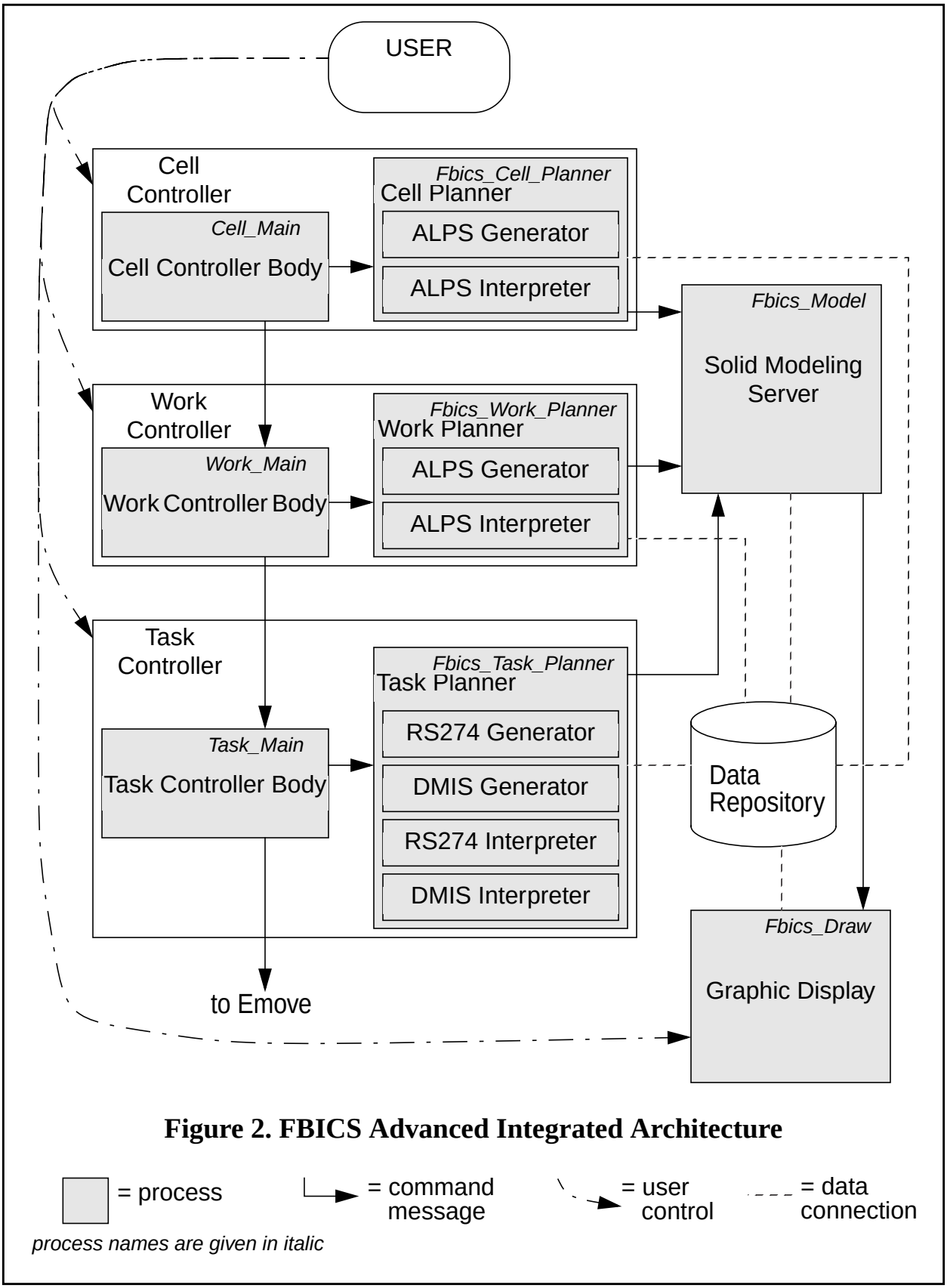


Figure 2. FBICS Advanced Integrated Architecture

The Advanced Integrated FBICS architecture is roughly the same as the stand-alone architecture, except that in the integrated architecture, each controller is split into two (or more) processes, and the Task Planner is all in one process.

Each controller is divided into a planner and the rest of the controller (called the “controller body” on Figure 2). The controller body might be further broken down into more than one process, but that is not shown on the figure. It is useful to have the planner in a separate process from the controller body so that the controller body can run cyclically at a fixed frequency or with a known maximum cycle time. This is desirable so that the controller can react quickly to the user, to status reports from subordinates, or to events in the environment. The planners in FBICS are not required to run with a known maximum response time, but may take as long as required to plan properly. Planners in other environments may not have this luxury. In any commercial version of FBICS, the planners should be interruptible to insure reasonable response time for the user; they are not currently interruptible in research versions of FBICS.

3.3 FBICS on Screen

This section describes how FBICS has been run and how it looks on a computer monitor in its usual configuration.

To run FBICS, the command “fbics” (the name of an executable startup script file) is given in a terminal window on a Sun computer. The script file then executes and starts up the various processes, each in its own terminal window. Because stand-alone FBICS consists of seven processes connected by reconfigurable interprocess communications, it is easy to assemble a FBICS system running on from one to seven computers. Only the startup script and configuration files need to be changed to change how FBICS is assembled.

The “fbics” script for running FBICS on a single computer brings up windows as shown in Figure 3. The Graphic Display (the Fbics_Draw process) has a graphics window in addition to a terminal window. The Fbics_Serve process window is entirely hidden by the Cell Controller window.

The three controllers (Cell, Work, and Task) are arranged top to bottom on the left side of the screen. The Fbics_Task2 process is part of the Task Controller, even though it has its own window. The drawing window and message window of the Graphic Display are at the top and middle of the right side of the screen, and the Modeler window is on the bottom right.

Each terminal window may have status information or error messages printed to it by the process running in the window. The terminal windows for the Modeler and the three controllers may also print time-out messages and accept “go ahead” commands from the user in case of a time-out. Status information is printed just to let the user know that each process is working and what it is doing. The type of status information printed varies from process to process and activity to activity. The status information printed in the WORKSTATION CONTROLLER, TASK CONTROLLER, TASK2, and MODELER windows in Figure 3 resulted from a `run_part_plan2(npl1)` command being given in the CELL CONTROLLER window.

The terminal window for each of the three controllers provides the user interface for the controller. Text may be entered there when the user interface is enabled. If the user has given a command, the user must wait for that command to be executed before giving another command. There is no way to abort or otherwise interrupt a command in progress.

3.3.1 Cell Controller Terminal Window

The Cell Controller terminal window is always enabled as a user interface. In Figure 3, the last line of text in the window shows that the user has just typed in another `run_part_plan2(npl1)` command. All the lines of text above that are part of the message that is printed when the user enters `help`. The user can give any of the user commands described in Section 7.2.2 when given the prompt `Cell =>` for user input.

3.3.2 Work Controller Terminal Window

The Work Controller terminal window may be enabled or disabled as a user interface. In Figure 3, user input has been disabled, so no user input is shown. On the figure, the `processing node` *`n`* output messages indicate how a stage-one ALPS process plan was traversed. The user can give any of the user commands described in Section 8.2.2 when user input is enabled and the `Work =>` prompt is showing.

3.3.3 Task Controller Terminal Window

The Task Controller terminal window may be enabled or disabled as a user interface. In Figure 3, user input has been disabled, so no user input is shown. On the figure, the text in the window shows that the Task Controller was most recently alternately generating and executing NC code files and has closed the setup on which it was working. The user can give any of the user commands described in Section 9.2.2 when user input is enabled and the `Task =>` prompt is showing.

3.3.4 Task2 Process Terminal Window

The Task2 process terminal window does not allow user input. It only prints output, which is either canonical inspection commands being called by a DMIS interpreter or canonical machining commands being called by an RS274/NGC interpreter. In Figure 3, ten canonical machining commands are showing.

3.3.5 Fbics_draw Graphic Display

In Figure 3, the Fbics_draw Graphic Display shows a wire frame view of the part to be made with a shaded view of one feature of the part. The display consists of a set of user controls on the right and a picture on the left. Details regarding the Graphic Display are given in Section 11.

3.3.6 Graphic Display Terminal Window

The Graphic Display terminal window (labelled “MESSAGES FROM FBICS DRAW”) is blank on Figure 3. Messages appear there only in case of error, which is rare. This window accepts no user input.

3.3.7 Modeler Terminal Window

The Modeler terminal window reports when its underlying engine, the Parasolid modeler, starts and stops. Two such messages are showing in Figure 3. It also reports modeling errors. The only user input the Modeler window accepts is responses to time-out messages.

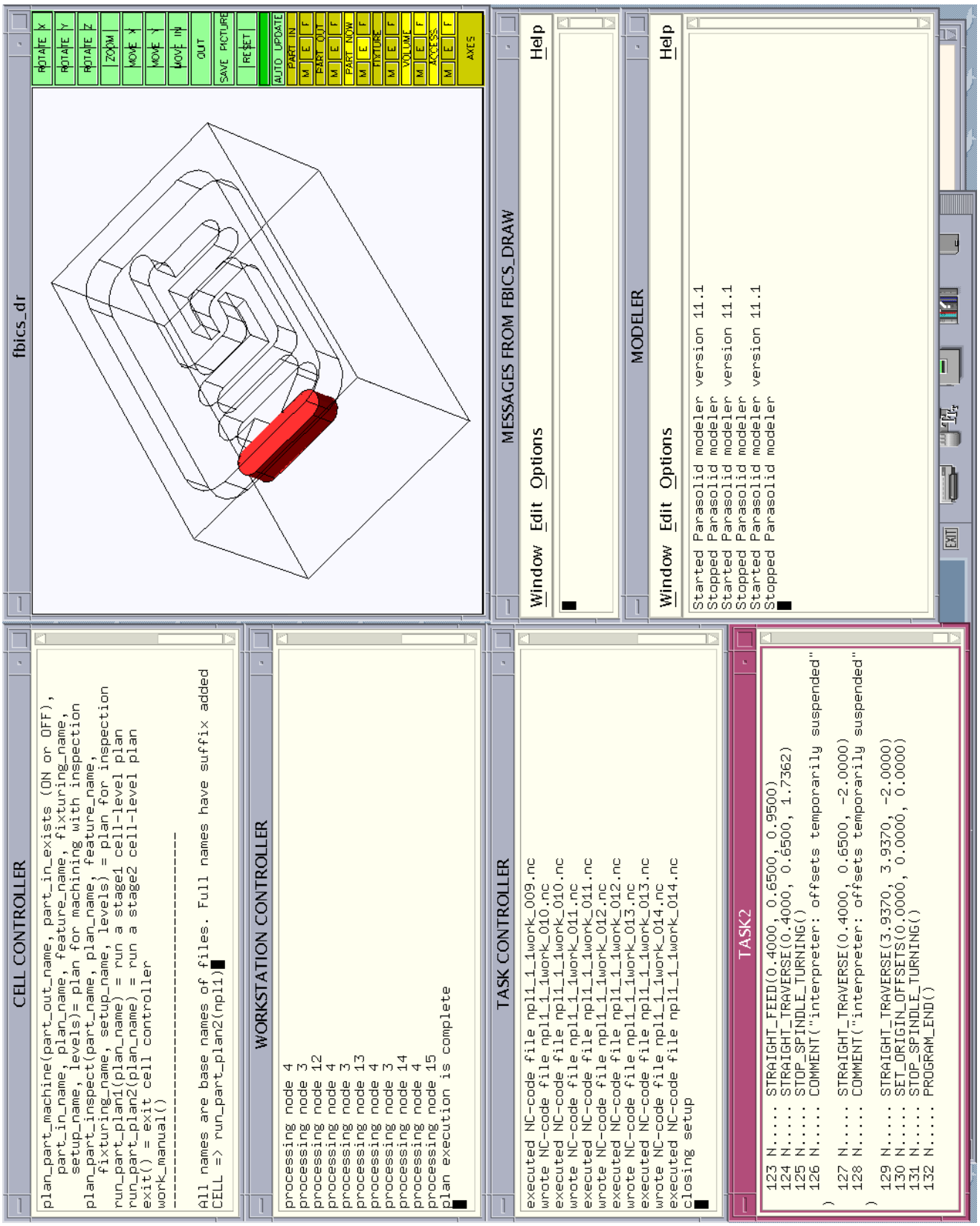


Figure 3. FBICS Screen Layout

3.4 File Formats

This section briefly introduces file formats[1] used in FBICS (excluding C++, the language in which the software is written). Details on all the file formats, including producers and consumers of files written in each format, are provided in Section 12. Section 12 also gives the naming conventions used in FBICS when it generates files of the various types.

All the file formats employed by FBICS are text-based, using ASCII characters; no binary formats are used. Two types of format are used: (1) those described using STEP with semantics in an EXPRESS model and grammar and syntax from STEP Part 21, and (2) those described in a single specification covering semantics, syntax and grammar.

EXPRESS models for FBICS STEP Part 21 file formats, with their uses in FBICS, include:

1. STEP AP 224, an international standard model providing a parametric library of machining features — used for describing whole parts (part_in, part_out, and part_now), fixtures, and individual features.
2. ALPS modified and with cell-level domain-specific task types added, a nonstandard model — used for stage-one cell-level plans.
3. An unnamed, nonstandard model for listing one_operations — used for stage-two cell-level and work-level plans.
4. An unnamed, nonstandard model — used for describing a setup of a part.
5. An unnamed, nonstandard model — used for describing shop options.
6. An unnamed, nonstandard model — used for describing task options.
7. An unnamed, nonstandard model — used for describing work options.
8. An unnamed model being standardized — used for describing a tool catalog.
9. An unnamed, nonstandard model consisting of the tool catalog language plus a few additional items — used for describing a tool inventory.
10. An unnamed, nonstandard model — used for describing tool_usage_rules.
11. ALPS modified and with work-level domain-specific task types added, a nonstandard model — used for stage-one work-level plans.
12. An unnamed, nonstandard model describing work-level machining and inspection executable operations in detail — used for work-level operation descriptions.

The FBICS file formats of the second type, with their uses in FBICS, include:

1. RS274 — used for task-level plans for controlling a machining center.
2. RS274/NGC variables — used for persistent variables of the RS274/NGC interpreter.
3. DMIS input format — used for task-level plans for controlling a CMM and for task-level plans for controlling inspection operations on a machining center.
4. DMIS output format — used to report the results of inspection back to the user or the Work Planner.
5. DMIS variables — used for common variables of the DMIS interpreter.
6. An unnamed, non-standard file format representing lines and planar polygons in 3D — used for describing pictures of whole parts, individual features, and fixtures.

---

1. For files of executable instructions, the combination of semantics and file format is generally called a “language”. Of the above, RS274 and DMIS are generally called languages, and ALPS should be considered a language. The cell-level and work-level file formats for ordered executable operations should also be considered languages, but no name has been applied to them.

3.5 Two-Stage Planning

FBICS employs two-stage planning at the cell and work levels. The stage-one plans prepared in the Cell and Work planners are flexible plans. A flexible plan is a plan with variables, alternatives, and (possibly) omissions. A stage-one plan may be executed directly or used to prepare a stage-two plan. A stage-two plan is an (inflexible) ordered list of one_operations to be executed. A fuller discussion of two-stage planning is given in Section 4.1.

3.5.1 Flow of Planning

FBICS planning follows one pattern for off-line planning and another for on-line planning. On-line planning means that a subordinate controller makes its plans while its superior is executing. Only stage-one plans are involved in on-line planning.

During off-line planning, the top one, two, or three levels are involved. When a planning command is given to the Cell Controller, one of the command arguments is an integer specifying the number of levels to plan for. If the value of the argument is one, only a stage-one cell-level plan is made (by the Cell Planner). If the value is two, a stage-one cell-level plan is made (by the Cell Planner), a stage-two cell-level plan is made (by the Cell Planner) and some number of stage-one work-level plans are made (by the Work Planner). If the value is three, all the plans just mentioned are made, and in addition, for each stage-one work-level plan, a stage-two work-level plan is made (by the Work Planner) and some additional number of task-level plans (NC and/or DMIS programs) are made (by the Task Planner).

3.6 Saving and Saying “Do This Task”

To accomplish machining and inspection tasks, the various parts of FBICS must be able to save plans to “do this task” and to tell other parts of FBICS to “do this task”. There are five ways of saving or saying “do this task” used by the three controllers in FBICS.

1. The user may type in a command to “do this task” at the user interface of any of the three controllers, as described in Section 7.2.2, Section 8.2.2, and Section 9.2.2. At the cell level there is a file format for saving commands. The file format is simply what the user would type. Each line of the file has a separate command.
2. In a stage-one process plan, “do this task” is a node (other than a navigation node) of the plan. The node may have attributes providing direct or indirect information about what to do. This form of “do this task” is modeled in EXPRESS and, hence (using STEP Tools), has both an in-memory representation and a file representation. The models for this data are described in Section 12.1.12 for cell-level plans and in Section 12.1.13 and Section 12.1.14 for work-level plans.
3. In a stage-two process plan, “do this task” is a one_operation in a list of one_operations. A one_operation has two components: the operation type and the name of a file giving more information about what to do. One_operation is also modeled in EXPRESS and is described in Section 12.1.17 for both cell-level and work-level plans.

4. Between a superior controller and a subordinate controller, “do this task” is a command message. There is an in-memory representation of command messages but no file representation (other than source code). Command messages to “do this task” are described in Section 8.3 for command messages received by the Work Controller and Section 9.3 for command messages received by the Task Controller. The Cell Controller has no superior, and, hence, receives no “do this task” messages. A command message provides only an operation type and one or two file names. “Do this task” messages to the Work Controller identify a setup file (see Section 12.1.16) for planning or a stage-one (see Section 12.1.13 and Section 12.1.14) or stage-two (see Section 12.1.17) plan file for execution. “Do this task” messages to the Task Controller identify a file describing an executable machining or inspection operation in detail (see Section 12.1.15).
5. Between a controller body and a planner, “do this task” is a function call to an interface function of the planner. These are described in Section 7.3, Section 8.4, and Section 9.4, for the Cell, Work, and Task controllers, respectively.

Much of what FBICS does involves converting these “do this task” items from one form to another and writing the various files used to “do this task”. To keep the various forms straight, and in keeping with the terminology introduced above, “do this task” will be called:

1. in a stage-one (i.e., ALPS) process plan: “node”,
2. in a stage-two process plan: “one_operation”,
3. in a command message: “message” (or the name of a specific message),
4. in an executable operation file “executable operation”.

For use as function return values and in command messages and one_operations, nine operation types are defined in FBICS: CLOSE_SETUP, INSPECTION, MACHINING, NONE, OPEN_SETUP, RUN_PLAN1, RUN_PLAN2, RUN_SETUP, and RUN_INSPECT_SETUP. These are set in HELVETICA type wherever they appear in this document, to make it clear they refer to operation types. The NONE, RUN_SETUP, and RUN_INSPECT_SETUP operation types are used only as function return values.

For example, at the work level, a twist_drilling node in a stage-one plan leads (if the plan is executed or a stage-two plan is made) (1) to a twist_drilling_ex executable operation file being written by the Work Controller, and (2) to a TASK_GEN_NC_MSG containing the name of the twist_drilling_ex executable operation file and the name of an NC code file to write being sent to the Task Controller from the Work Controller. When the Task Controller receives the message, it writes the NC code file. The NC code file contains RS274 commands for drilling a hole. In a stage-two work-level plan there is then a MACHINING one_operation containing the name of the NC code file. When the Work Controller executes either the stage-one or stage-two plan, it sends a TASK_EXEC_NC_MSG to the Task Controller containing the name of the NC code file.

3.7 FBICS Operation

This section presents introductory facts about FBICS then describes what happens during:

1. FBICS initialization.
2. planning three levels deep for machining with inspection.
3. planning three levels deep for pure inspection.
4. execution of the cell-level stage-one plan made in item 2 above.

5. execution of the cell-level stage-two plan made in item 2 above.
6. adaptive machining, when inspection results feed back to planning.

The user may give commands directly to the Cell Controller, Work Controller, or Task Controller. This section deals only with what happens when the user gives a command to the Cell Controller.

Execution of a cell-level plan for pure inspection does not differ in character from execution of a plan for machining with inspection (other than being devoid of machining), so no separate description of it is given here.

3.7.1 Introductory Facts

3.7.1.1 Setup

The notion of a “setup” is central to FBICS. A setup is a fixturing of a workpiece on the table of a machining center or coordinate measuring machining. The workpiece does not move with respect to its fixturing during a single setup. It does move from one setup to the next. Some parts can be handled in a single setup. More commonly, two or more setups are required.

3.7.1.2 Using the Modeler

While planning or executing, all three controllers repeatedly use the Modeler to build, manipulate, and display solid models of parts, features, access volumes, and other geometric data. The Modeler is used also to verify plan correctness at the end of cell-level and work-level stage-one planning by checking that the part_now is the same shape as the part_out. The interface between the controllers and the Modeler includes both messages (see Section 5.4) and data files (see Section 5.5).

3.7.1.3 Display of Objects

Display of objects is handled by the Modeler and Graphic Display jointly. For each type of object to display, the Modeler facets the object, writes a modeling file and sends a message to the Graphic Display, telling Graphic Display to read the file and display it as directed by the user. The display serves to let the user see how planning or execution is progressing.

3.7.1.4 Interactions Between Modules

Both the controller body and the planner of each controller are involved during hierarchical planning. The planners do not communicate directly with each other via messages. The planners communicate indirectly via files; a file written by one planner will be read by another. The controller bodies communicate via messages and do not use files.

3.7.2 FBICS Initialization

FBICS is brought to a warm state when the user gives the shell command “fbics” in a terminal window. As mentioned earlier, this runs a shell script that starts the various independent processes and establishes communications among them.

FBICS is initialized when the user gives an initialization command to the Cell Controller (see Section 7.2.5). This causes initialization command messages to be sent down hierarchically, with each controller in the hierarchy sending an initialize command to its subordinate, and each

subordinate sending an OK message to its superior when it has completed its initialization. When all controllers report successful initialization, the Cell Controller is ready to accept other commands. Each of the three controllers will not accept other commands until it has been initialized.

Each of the planners maintains its own world model structure. The Modeler and Graphic Display do not have world model structures in the FBICS source code, except for a specification of length units in the Modeler. The commercial software used in the Modeler and Graphic Display do, of course, have extensive internal models. During initialization, each planner cleans up its world model, sets default values for various world model parameters, and reads various files, which may reset some of the world model parameters. Initialization of the Cell, Work, and Task planners is described in Section 7.3.4, Section 8.4.4, and Section 9.4.5, respectively.

3.7.3 Planning Three Levels Deep for Machining with Inspection

This section gives an example of a typical scenario for FBICS planning for machining with inspection. In addition, some description is given of alternatives to the scenario.

The most complex form of FBICS planning occurs when the Cell Controller is told to plan three levels deep including both machining and inspection, so that type of planning is described here. It is assumed that this type of processing will be performed by a machining center equipped with cutting tools and a touch probe. Other types of planning for a machining center are similar but simpler, usually including a subset of the three-level planning activities.

The scenario begins when the user gives the Cell Controller a command to make plans to a depth of three control levels for machining (with in-process inspection) a finished part from a specific starting workpiece. This scenario supposes also that the user has set inspection options so that features with any tolerance on any feature parameter are to be inspected. The scenario supposes further that no feature has a tolerance considered tight under the current setting of the meaning of tight.

In planning for machining with inspection, the major alternatives to these assumptions include:

1. Control levels to plan for: The number of control levels to plan for may be 1 or 2.
2. Starting workpiece: The system may be told there is no design for the starting workpiece.
3. Inspection: Different rules for deciding what to inspect may be put in place as described in Section 3.7.4.

3.7.3.1 Cell Controller Planning

When the user gives a plan_part command to the Cell Controller, the user provides the file name of the part_out, the file name of the part_in, a flag indicating whether the part_in file already exists or should be written, the base name for setup files, the base name for plan files, and the number of levels to plan. We are assuming here that the number of levels is three. The general plan of attack is to make a stage-one cell-level plan first, then make a stage-two plan from the stage-one plan, sending planning command messages to the Work Controller at the same time the stage-two cell-level plan is made. Details of how the Cell Controller does this are in Section 7.2.6. Since the Cell Controller is planning three levels deep, the planning commands it sends to the Work Controller tell the Work Controller to plan two levels deep.

3.7.3.2 Work Controller Planning

Each time the Cell Controller commands the Work Controller to plan a setup two levels deep, the Work Controller first makes a stage-one plan for the setup. Then the Work Controller makes a stage-two plan from the stage-one plan. While it is making the stage-two plan, the Work Controller generates command files to be used later by the Task Controller. Finally, the Work Controller traverses its stage-two plan. As it does so, it first sends a `TASK_OPEN_MSG` to the Task Controller and then a number of `TASK_GEN_NC_MSG`s and `TASK_GEN_DMIS_MSG`s. Details of how the Work Controller does this are in Section 8.2.7.

3.7.3.3 Task Controller Planning

When the Task Controller receives a `TASK_OPEN_MSG`, it reads the setup file whose name is received with the message, reads the files describing the part_in and fixture, and calls the Modeler to model the part_in (but not the fixture).

A `TASK_GEN_NC_MSG` includes (1) the name of an executable operation file describing the machining operation and feature to make and (2) the name of an NC code file to write. When the Task Controller receives a `TASK_GEN_NC_MSG`, it reads the input file, calls the Modeler to model the feature and the workpiece as it will be when that feature is subtracted from it, and calls an NC code generator to write the output file.

A `TASK_GEN_DMIS_MSG` includes (1) the name of an executable operation file describing the inspection operation and feature to inspect and (2) the name of a DMIS code file to write. When the Task Controller receives a `TASK_GEN_DMIS_MSG`, it calls a DMIS code generator to write the output file. As compared with NC code generation, the Task Planner has the extra job of extracting DMIS features from AP 224 features; there is no counterpart in generating NC code.

Further details of NC code generation are in Section 9.5. DMIS generation details are in Section 9.6.

3.7.3.4 An Alternative

If any of the features to be machined had been a pocket with a tight tolerance on a parameter (length, width, or corner radius), several aspects of the planning scenario would have been different. A tolerance might become tight either by being made smaller or by the definition of tight being made looser.

Most significantly, it would not have been appropriate to plan three levels deep, because the NC code generator in the Task Planner would have tried to use inspection results which would not have been available. The way inspection results are used in NC code generation is explained in Section 3.7.7. It would have been OK to plan two levels deep because there is no feedback (other than error status) from Work Controller to Cell Controller that affects cell-level planning for machining with inspection.

3.7.4 Planning Three Levels Deep for Pure Inspection

The behavior of the system when planning for pure inspection is much the same as in machining planning. One factor makes inspection planning easier: the workpiece does not continually change shape during processing. Other factors make inspection planning harder. In machining planning, deciding what to do is simple — every bit of material in all the features not already on

the workpiece has to be removed. In inspection planning, deciding what to do is not so simple. Which AP 224 features should be inspected? How thoroughly should they be inspected? When should they be inspected? Until it is decided which features are to be inspected, it is not known whether any proposed setup will be required. These questions are addressed in detail in Section 4.4. The general approach taken by FBICS is to have the user establish policies and/or make decisions on a case-by-case basis.

When the user gives a plan_inspect_part command to the Cell Controller, the user provides the file name of the design of the part, the base name for setup files, the base name for plan files, and the number of levels to plan. We are assuming the number of levels is three. The general plan of attack is the same as for machining with inspection: make a stage-one plan, then make a stage-two plan concurrently with work-level planning. How this is done is described in Section 7.2.7. The Cell Planner decides which features are to be inspected and defines only those setups in which features to be inspected are accessible.

Inspection planning in the Work Planner and the Task Planner is much the same for pure inspection as was described in Section 3.7.3 for inspection planning done concurrently with machining planning.

3.7.5 Executing a Stage-one Cell-level Plan

This section describes what happens when the user tells FBICS to execute the stage-one cell-level plan made in Section 3.7.3. Exactly the same stage-one cell-level plan would have been made if the user had told the Cell Controller to plan one or two levels deep, rather than three levels (as was the case). When a stage-one plan is executed, any previous stage-two or lower level planning is ignored. Planning below the cell level is done from scratch. No stage-two plans are made at any level.

3.7.5.1 Cell Controller Execution

During execution of a stage-one cell-level plan, the Cell Controller traverses the plan one step at a time. For each plan node requiring action (they are all run_setup nodes), the Cell Controller first sends a `WORK_PLAN_MSG` or a `WORK_PLAN_INSP_MSG` to the Work Controller, and then a `WORK_RUN1_MSG` (indicating a stage-one plan should be used) immediately after. The way in which the Cell Controller traverses its stage-one plan and disambiguates data is identical to the way it is done in preparing a stage-two plan from a stage-one plan (described in Section 3.7.3.1). A slightly different naming convention is used for the disambiguated data, which is intended to be short-lived in the case of executing a stage-one plan. Details of how the Cell Controller executes stage-one plans are given in Section 7.2.8.

3.7.5.2 Work Controller Execution

For each `WORK_PLAN_MSG` or `WORK_PLAN_INSP_MSG` it receives, the Work Controller makes a stage-one plan exactly as described in Section 3.7.3.2.

For each `WORK_RUN1_MSG` it receives, the Work Controller reads the stage-one plan and traverses the stage-one plan exactly as described in Section 3.7.3.2, writing executable operation files as described in that section. During execution, however, as soon as each operation is selected, the Work Controller first sends a `TASK_GEN_NC_MSG` or a `TASK_GEN_DMIS_MSG` to the Task Controller then sends a `TASK_EXEC_NC_MSG` or `TASK_EXEC_DMIS_MSG` to the Task

Controller immediately afterward.

3.7.5.3 Task Controller Execution

Generating NC code and DMIS code is the same during execution as during planning. Executing the code is done during execution (of course) but not during planning. As shown on Figure 1, the Task Controller includes two separate processes. Code generation is performed entirely by the Fbics_Task process. Code execution is performed entirely by the Fbics_Task2 process.

For each TASK_GEN_NC_MSG it receives specifying that an AP224 feature should be cut, the Task Controller generates NC code as described in Section 9.5. TASK_GEN_NC_MSGs are not all for cutting. They may also be for starting a program, ending a program, coolant control, or changing a tool.

For each TASK_GEN_DMIS_MSG it receives specifying that an AP224 feature should be inspected, the Task Controller generates DMIS code as described in Section 9.6. TASK_GEN_DMIS_MSGs are not all for inspecting a feature. They may also be for starting a program, ending a program, or changing a tool.

For each TASK_EXEC_NC_MSG or TASK_EXEC_DMIS_MSG it receives, the Task Controller (by call to either the RS274/NGC interpreter or the DMIS interpreter) interprets the file named in the command, adjusts its internal model appropriately, and makes calls to canonical machining or inspection functions. In stand-alone FBICS these canonical calls simply print themselves and update a dummy external world model required by the interpreter. In integrated FBICS, the canonical calls are used for machine control and actual cutting or inspection.

3.7.6 Executing a Stage-two Cell Plan

This section describes what happens when the user tells FBICS to execute the stage-two cell-level plan made in Section 3.7.3.

To start execution, the user gives the Cell Controller a run_part_plan2 command (see Section 7.2.9), providing the base name of the cell-level process plans previously written. The Cell Controller selects the correct plan to read and reads the plan. The plan is a list of RUN_PLAN2 one_operations. For each one_operation on the list, the Cell Controller sends a WORK_RUN2_MSG to the Work Controller naming the stage-two plan to use.

For each WORK_RUN2_MSG the Work Controller receives, the Work Planner reads the plan. The plan is a list of one_operations. The Work Controller generates one message to the Task Controller for each operation.

The first one_operation on the Work Controller's list is of type OPEN_SETUP, so the Work Controller sends a TASK_OPEN_MSG to the Task Controller. Since a stage-two work-level plan is being executed, the setup file name in the TASK_OPEN_MSG is set to the empty string. When the Task Controller observes the empty string, it gets ready to run for executing NC or DMIS code files but does not go through the elaborate preparations necessary for generating code. The rest but last of the one_operations are MACHINING or INSPECTION operations, for which the Work Controller sends TASK_EXEC_NC_MSGs or TASK_EXEC_DMIS_MSGs to the Task Controller. These are executed in the Fbics_Task2 process of the Task Controller. The last work-level one_operation on the list is of type CLOSE_SETUP, for which a TASK_CLOSE_MSG is sent to the Task Controller.

If the Cell Controller had planned only two levels deep, the Cell stage-two plan would consist of RUN_PLAN1 one_operations, the Work Controller would be executing stage-one plans as the Cell controller executed its stage-two plan, and the Task Controller would go through exactly the same procedures as described in Section 3.7.5 (both generating and executing code files).

3.7.7 Feedback from Inspection to Machining Planning

FBICS includes using feedback from inspection to machining planning, as follows. If there is a tight tolerance on any of the length, width, or corner radius of a pocket to be finish-milled, the Work Planner puts a finish_mill_adaptive node for milling the pocket in the stage-one work-level plan. Both a cutting tool and a probe tool are used in the step. Tool-use parameters for both tools are also included. When executable operations are created for carrying out that node during execution of the plan by the Work Controller, three primary executable operations are defined (in addition to tool changing and coolant control, if needed). The first primary executable operation is to finish mill a pocket slightly smaller than the specified pocket. The second is to inspect the smaller pocket, and the third is to finish mill the final pocket. When the Task Controller executes the DMIS file for inspecting the smaller pocket, the DMIS interpreter (because the inspection program says to do so) writes the results of the inspection to a file. Before generating code for the final cut on the pocket, the Task Controller reads this file. A correction factor for the final cut is calculated as the average error in the radii of the four corners of the pocket. If the radii are too big on average, NC code is written so the final cut is made inside the nominal tool path by the correction factor. If the radii are too small, NC code is written so the final cut is outside the nominal path by the correction factor.

To simplify the implementation of adaptive machining, the DMIS interpreter has been modified so that it prints two output files, a short one containing the output resulting from interpretation of the most recent DMIS file, and a longer one containing the output resulting from the interpretation of all DMIS files interpreted since the DMIS program (which may be spread across many files) was begun. For adaptive machining, the short file is used.

3.8 Length Units

FBICS handles millimeters and inches length units. FBICS is able to deal with parts using length units different from the units used for cutting tools and probes. Handling length units correctly is much more difficult than would be expected. To deal consistently and correctly with length units, a number of principles were adopted. The principles were then implemented in the software and followed in preparing manually prepared data.

3.8.1 Principles for Length Units

3.8.1.1 Allowed Length Units

Only inches and millimeters may be used as length units.

3.8.1.2 Module Awareness of Length Units

Each module that is aware of length units (Fbics_Cell, Fbics_Work, Fbics_Task, and Fbics_Model) has the notion of current length units.

3.8.1.3 Consistency of Length Units Between Modules

At any one time, the four FBICS modules that need to be aware of length units should use the same length units.

3.8.1.4 Changeability of FBICS Length Units

A user can change the current length units of a module while the module is running. If length units are changed for one module, they should be changed for all four.

3.8.1.5 Length Units in AP224 Files

AP224 files may use either inches or millimeters, as long as the same length units are used throughout any one file.

The reason for allowing AP224 files to use either inches or millimeters is that it is expected that some AP224 files will come from outside FBICS, and we want FBICS to be able to handle these, regardless of which length units they use.

The reason for allowing only one type of length unit in an AP224 file is that in the AP, the use of Numeric_parameter vs. REAL is inconsistent. For example, in Target_rectangle, target_length is a REAL and there is no way to determine what units are intended, while in Linear_profile, profile_length is a Numeric_parameter. If all the Numeric_parameters for length are required to use the same length units, it is reasonable to assume that the REALs representing length implicitly use the same unit. If there were different types of length units in different Numeric_parameters, there would be no way to infer what length unit a REAL representing length is using.

3.8.1.6 Length Units in Tool Catalog Files

Tool catalog files may use different units for different tools. The type of length unit is given with every number representing a length, so there is no ambiguity.

It is likely that a shop will have some tools measured in inches and some measured in millimeters, so it is very desirable to allow both types of unit in data read by FBICS.

Tool inventory files do not use length units.

3.8.1.7 Length Units in Files Written by FBICS

Executable operation files, stage_one process plan files, and setup files must use a single length unit throughout, and that must be the same unit as currently in use by a module that reads or writes such files.

3.8.1.8 Convert Length Data on Reading

In each of the four modules aware of length units, all input length data is converted to the current units when it is read, so that no conversions are needed during subsequent processing.

3.8.1.9 Assume Current Length Units if No Explicit Unit Given

The current length units are assumed for any quantity whose measure uses length units but is not explicitly described. For example, feed rate is implicitly in current length units per minute.

3.8.2 Implementation Details

3.8.2.1 Specifying length units

For AP224 data, only the strings “in” and “mm” are allowed as length unit type for Numeric_parameters. To be sure a length unit is specified, part_in, part_out, features, and executable operations files must all have at least one Numeric_parameter.

For options, there is an EXPRESS TYPE length_unit_rule_type in the shop_options schema with values use_in and use_mm. The task_options and work_options schemas USE the shop options schema so that they can use the same TYPE.

For the tool catalog (and, hence, the tool inventory), there is an EXPRESS TYPE, length_unit_type with values millimeters and inches.

For stage_one process plans, an internal string with name “length_units” and value “in” or “mm” is used.

For executable_operations, a dummy Numeric_parameter whose units are “mm” or “in” (as determined by the current length units setting) is inserted by Fbics_Work whenever an executable_operation is built.

For setups, a setup_spec has the attribute length_units, a string whose value must be “in” or “mm”.

3.8.2.2 Length units in the options

Fbics_Cell, Fbics_Work, Fbics_Task, and Fbics_Model all extract the length units to use from the shop options file, which is read before any other options file is read. The read_shop_options function is called whenever any of the four processes executes init_XXX.

1. In Fbics_Cell, init_cellpl is called whenever cellpl_init, close_plan1, close_plan2, plan_part1_inspect, or plan_part1_machine is called. Cellpl_init is called at startup and whenever the user wants to re-initialize.
2. In Fbics_Work, init_workpl is called whenever workpl_init, close_plan1, close_plan2, plan_inspect_setup1, plan_setup1, or plan_setup2 is called. Workpl_init is called at startup and whenever the user wants to re-initialize.
3. In Fbics_Task, init_taskpl is called whenever taskpl_init, taskpl_open_setup, or taskpl_close_setup is called. Taskpl_init is called at startup and whenever the user wants to re-initialize.
4. In Fbics_Model, init_model is called whenever one of the other three processes attaches when no process was already attached. All processes detach between rounds of planning or execution, so all are detached whenever Fbics_Cell is waiting for a user command.

Work and task options also have length units, but that only indicates the units for the file. When a work options or task options file is read, it must use the same length units as already set when the shop options were read. Read_work_options and read_task_options are called only immediately after read_shop_options.

3.8.2.3 Length units in world models

The Fbics_Cell, Fbics_Work, Fbics_Task, and Fbics_Model world models each have the attribute length_units_factor, which is set to -1.0 (meaning unset) on initialization, and to 1.0 for millimeters or 25.4 for inches when the shop options are read.

To change the length_units_factor in a world model, it is necessary to change the length units specified in the shop_options file and then reinitialize.

3.8.2.4 Checking length units

When executable operation files are read, it is checked that all Numeric_parameters whose units are not “degrees” have either “mm” or “in” for parameter_units, and that this matches the current world model setting.

When stage_one process plans are read in Fbics_Cell or Fbics_Work, the “length_units” plan variable is checked. If the current length_units_factor is 1, the value of the variable must be “mm”; if 25.4, the value of the variable must be “in”.

When AP224 files are read (part_in, part_out, features, fixture), it is checked that all Numeric_parameters whose units are not “degrees” have either “mm” or “in” for parameter_units. If the parameter units do not match the current length units (as indicated by the length_units_factor), the values of lengths and coordinates are changed and the names of the units are changed.

When setup files are read in Fbics_Cell, Fbics_Work and Fbics_Task, it is checked that the length_units attribute of the setup_spec is either “mm” or “in” and matches the length_units_factor setting.

When the tool catalog file is read in Fbics_Work and Fbics_Task, tool length units are checked and converted to current units if they differ, and tool attributes that are lengths are changed if needed. The tool inventory has no length units, so it needs no conversion.

3.9 Speed of Execution

In stand-alone FBICS, executing a stage-two cell-level plan for which planning has been done three levels deep goes about 26 times as fast as executing the corresponding stage-one plan. In tests using a Sun SPARCstation 20, a simple part (11 features, one setup) took 4 seconds using a stage-two plan, versus 105 seconds for a stage-one plan. For a complex part (51 features, 8 setups), the corresponding times were 45 seconds and 1180 seconds. The largest part of this time differential is caused by the Modeler and Graphics Display being unused during stage-two plan execution.

When the Modeler and Graphics Display are being used, they use over 90% of all the processing time used by FBICS.

4 FBICS Planning

This section discusses planning issues and algorithms.

4.1 Two-stage Planning

As noted earlier, FBICS employs two-stage planning at the cell and work levels. The use of two-stage planning is driven by the desirability of using flexible plans, combined with the requirement of doing hierarchical planning.

A flexible plan is a plan with variables, alternatives, and (possibly) omissions. Omissions might include omitting tool-changing and coolant commands from stage-one plans; FBICS makes these omissions. Using flexible plans solves the common problem that in the discrete parts and other domains where the formulation of plans is costly in time or money, it is desirable that process plans be reusable. If conditions under which plans are executed may change in a way that affects the utility of plans, however, completely fixed process plans may not be reusable. At least four types of data may change:

1. sensory data — e.g., temperature, probed position, vibration,
2. resource data — e.g., tool catalog, tool inventory (the tools in a machine carousel),
3. job data — e.g., the number of widgets to make now,
4. planning system behavior data — e.g., which rule to use to set spindle speeds or feed rates, which strategy to use to decide what features to inspect.

If any of these types of data affects planning, a fixed plan will become obsolete when the data changes.

In FBICS, the stage-one plans prepared in the Cell and Work planners are flexible plans written in the ALPS process planning language. A flexible plan is expected to be usable for an extended time under a variety of conditions whose variability is embodied in the variables and omissions of the plan. In addition to providing for changeable conditions, a stage-one plan may leave alternatives open because the planning system recognized alternatives but elected not to choose among them.

A stage-one plan may be executed directly. When the plan is executed, it is traversed one step at a time and one or more operations are executed (by physical action or by sending commands to a subordinate) for each step of the plan. By waiting for each command to be executed before selecting the next step to run, the executor allows for using up-to-date values of plan variables. In FBICS, each operation is executed by first telling the subordinate to make a plan to carry out the operation and then telling the subordinate to execute the plan it just made. If either the subordinate's planning or its execution is unsuccessful at any point, execution of the superior's plan fails and is stopped.

A disadvantage of executing a flexible plan directly is that, for each command sent to a subordinate, it may be necessary for the subordinate to make a plan from scratch for carrying out a command received from its superior. If the subordinate planner cannot make such a plan, execution of the superior plan fails. A second disadvantage of executing a flexible plan directly, if it is executed more than once under the same conditions, is that after the first execution, the subordinate will be needlessly making the same plans over again, entailing needless cost.

If conditions will not change for a period of time during which the superior's plan is to be

executed one or more times, it is, therefore, useful to prepare a stage-two plan for the superior and a set of stage-one plans in the subordinate. To do this, the superior's stage-one plan is traversed, producing a set of executable operations by selecting a specific sequence of tasks from among alternatives in the stage-one plan, selecting specific resources where the stage-one plan has generic resources, and filling in blanks where the stage-one plan has omitted items. The subordinate makes a plan for carrying out each of the superior's executable operations. The superior also makes an ordered list of one_operations that is the stage-two plan for the superior.

Another reason for having stage-two plans is to support optimizing plans across a hierarchy. To perform optimization over a hierarchy via standard search methods, a stage-one plan allowing for all cases to be tested is prepared for the top controller. Partial search paths (plan traversals) are constructed through this top level plan. For each node (plan step) on a partial search path at the top level, the next level down is told to find an optimum execution. The total cost of any one search path at the top level is the direct cost of the path to the top level plus the cost of each node on the path to the subordinate(s) that handles it. The subordinate controllers follow exactly the same procedure in determining their own optimums. To support this type of search, it is necessary to have a format for recording the optimum path. The stage-two plan provides that format. The stage-two format could also be used to record partial search paths. No form of optimizing search has yet been implemented in FBICS.

In principle, a set of plans for a subordinate might be made for each allowable traversal of a superior's plan (i.e., for every possible stage-two plan of the superior under current conditions). In FBICS, this is not generally feasible because of combinatorial explosion. For one standard test part, for example, the automatically generated cell-level stage-one process plan could be traversed seven factorial (5040) ways. Generating sets of work-level plans for all of these ways would require several days. Thus, generating a single stage-two plan for a superior plus one set of plans for a subordinate has been implemented in FBICS.

4.2 ALPS

The background of ALPS is given in Section 2.5.5. A few details of the use of ALPS in FBICS are given in Section 12.1.9.

ALPS is a general-purpose discrete event planning language.

The fundamental object of ALPS is the plan. A plan is a recipe for performing a specific task. A plan in ALPS is a one-level breakdown of a task into subtasks, expressing the requirements of each subtask and its interrelation with other subtasks in the plan. A plan contains a set of nodes (or steps) which provide sequencing information and detail how to perform the task in terms of individual subtasks. Every plan has an associated target system which may execute the plan.

ALPS provides for parallel operations, alternatives, synchronization of events, parameters, and resource allocation. To help in specifying how an ALPS plan may be traversed at execution time, each node in a plan has a list of successors and a list of predecessors. The way in which predecessors and successors are used in determining plan traversal varies according to the type of node.

The version of the ALPS schema used in FBICS has been extensively modified from its starting point, which was a schema (the "MSID schema") provided by the Manufacturing Systems Integration Division that had been developed in a project of the same name. The schema has been named FBICS_ALPS, in recognition of the changes. Descriptions of the major changes to the

MSID schema made to produce FBICS_ALPS follow.

4.2.1 Planning Stages

The MSID schema provided for three stages in the evolution of a plan: process_plan, production_managed_plan, and production_plan. Roughly speaking the first stage is characterized by having (possibly) generic resources and no specific scheduling of events. The second stage is characterized by having only specific resources but still no scheduling, and the third stage is characterized by having specific resources and specific scheduling. FBICS includes the notion of generic resources (primarily in its handling of cutting tools) but makes no attempt to deal with scheduling. In addition to the three stages, the MSID schema was based on the premise that many decisions would be made at execution time. The execution time traversal of a plan might be regarded as a fourth stage. FBICS uses only two of the four stages: the original plan and the execution time traversal of the plan. Thus, the MSID schema was simplified by removing all the items (entities, types, rules, etc.) required only for scheduling.

4.2.2 Preconditions

Primitive process planning languages provide for plan traversal at execution time simply by putting the steps of the plan in a list and requiring that all steps be performed in order. More advanced process planning languages provide for traversal by including branch points and partial ordering schemes; usually some method of testing conditions is provided for use at the branch points.

One common method for providing conditions to use in determining traversal of a plan is to associate a list of evaluable conditions with each step and require that all conditions must evaluate to true before the step may be executed. A common type of condition on a plan step is that some specific list of other steps must be executed earlier. The process plan protocol used with the VWS2 system (see Section 2.5.1), for example, used only this type of condition. With only this condition, it is easy to specify either linear traversal (for each step N after the first, just specify that step N-1 must be executed earlier) or allow for many alternative traversals.

The MSID ALPS schema (and earlier versions of ALPS) provided for traversal by the use of split_nodes and join_nodes. Two types of split_node (predicated and parameterized) were provided. But neither type provided the capability to use conditions as described in the previous paragraph. This was recognized as a shortcoming early in the development of ALPS.

To provide for the use in plan traversal of lists of steps that must be executed earlier, a new type of split_node, the “precondition_split_node” has been added to FBICS_ALPS. The branches that split at a precondition_split_node must all rejoin at a matching join_node. During plan traversal, if a precondition_split_node is encountered, until after the matching join_node is executed or another type of split_node is executed, the predecessors of a node are taken to be those nodes which must be executed earlier.

4.2.3 Method of Providing Extensions to the ALPS Schema

The developers of ALPS realized that two types of extensions to ALPS would be required when using ALPS to do work. First, to provide for conditions that could be evaluated, expressions would be required. Second, for specific application domains, domain-specific tasks and their attributes would be required.

The ALPS developers, realizing that a fairly powerful expression model would be desirable and that many such models already existed, did not want to pick one and model it. Rather, they provided that an expression would be a string. That way, different ALPS users could use different expression models. Whatever expression model was chosen, the user would need software to handle strings corresponding to the model.

The ALPS developers also realized that the number of potential application areas of ALPS was huge, so there would be no hope of their defining enough types of tasks to cover all needs. The solution they chose was to define "primitive_task_node" and to provide that it would include (1) a string identifying a work_element to be performed and (2) a set of attributes, each with a name (string) and a value (expression). Whatever set of primitive tasks an application chose to define, the application would need to have software to recognize the names, recognize the attribute names for each task, and evaluate the attributes.

The methods chosen for these two extensions did not take advantage of the capabilities of the EXPRESS language or the availability of tools for automatically generating computer software from EXPRESS. In the case of primitive_task_node, this is particularly unfortunate, since one of the primary capabilities of EXPRESS is to define attributes of entities.

The method of providing these extensions used by FBICS_ALPS does take advantage of those capabilities. Rather than using strings, both expressions and subtypes of primitive_task_node are defined in EXPRESS. In neither case was the capability put directly into the FBICS_ALPS schema, so it still defines a completely generic language. To make the FBICS_ALPS schema from the MSID schema, the changes therefore included: (1) the old definition of expression was deleted, (2) a USE statement was added to incorporate the separate expressions schema, (3) the definition of primitive_task_node was modified by removing its work element attribute, and (4) the list of attributes was deleted from the definition of node. The two sets of specific subtypes of primitive_task_node used in FBICS (for the cell and work levels) are defined in the fbics_combo schema, and the fbics_combo schema USEs FBICS_ALPS. Software for accessing all the data of the additional sorts is generated automatically. The only additional hand-written software that was needed was the expression evaluation and variable handling software.

4.2.4 Use of FBICS_ALPS in FBICS

FBICS uses FBICS_ALPS for cell-level and work-level stage-one process plans. FBICS uses only a subset of the entities defined in the FBICS_ALPS schema. Those used by FBICS are shown in Figure 4. In addition to the entities shown in Figure 4, which are general-purpose, subtypes of primitive_task_node are defined for tasks at the cell level and work level. Figure 19 shows the subtypes of primitive_task_node used in work-level plans. Cell-level plans use only one subtype of primitive_task_node, run_setup, as described in Section 12.1.12.

For parameterized_split_nodes in FBICS, the *m_number* is always zero, and the *timing* is always serial.

The *predecessors* and *successors* lists of the nodes of an ALPS plan contain duplicate information, since a predecessor is always the inverse of some successor. To prevent errors in entering duplicate information, FBICS requires that the *predecessors* list of every node be empty in the file representation of an ALPS plan. When an ALPS plan is read by FBICS, the *predecessors* list of every node is built in the in-memory plan representation from the *successors* lists of other nodes.

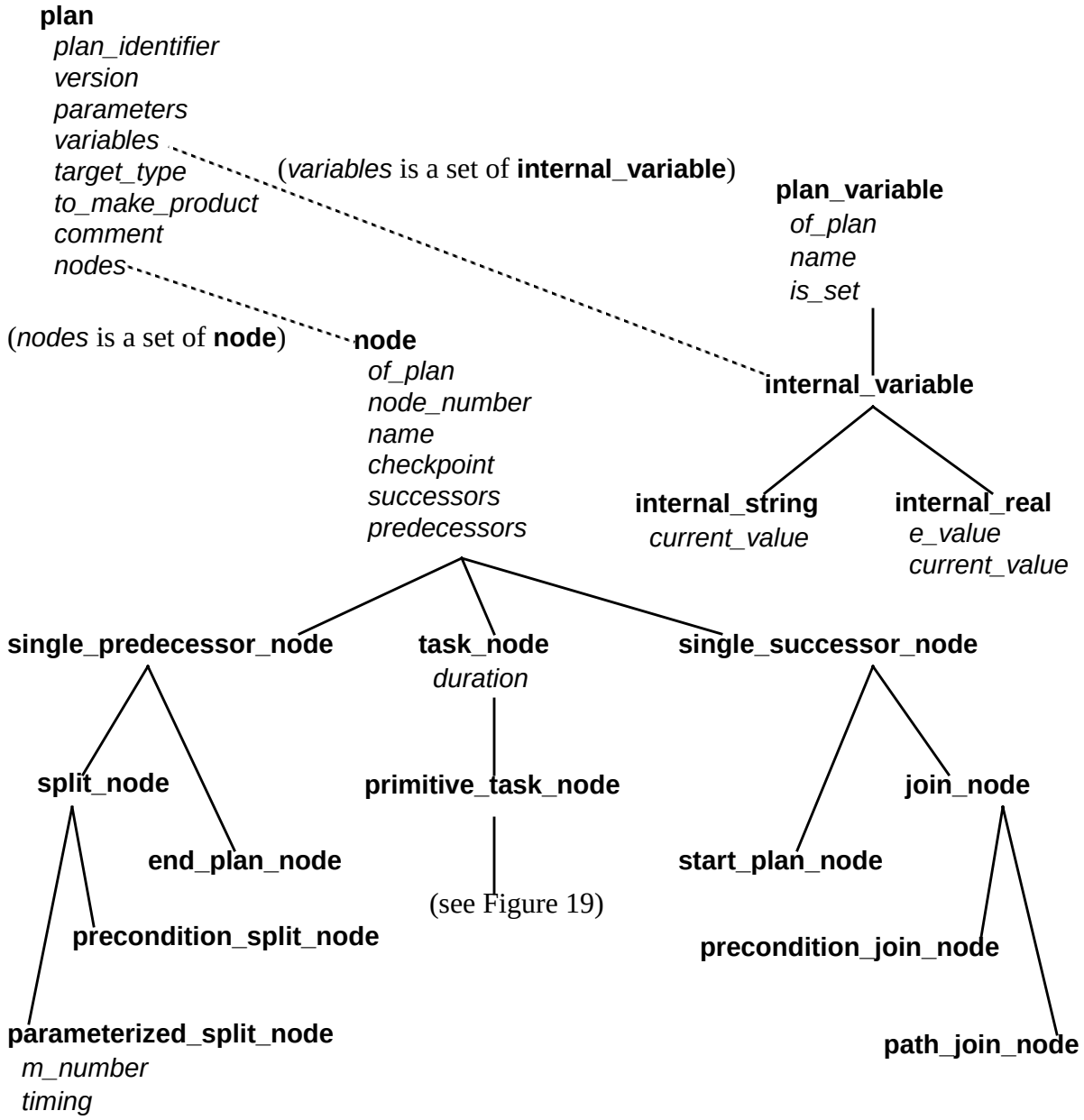


Figure 4. FBICS Use of FBICS_ALPS

ALPS entities used in FBICS are shown in **boldface helvetica** type.
Supertypes are connected to subtypes by solid lines, with the supertype higher on the page.
Attributes names are shown in *italic* type.
Except for the *nodes* and *variables* attributes of **plan**, data types of attributes are not shown.
Only leaf entity types may be instantiated.
To find all the attributes of a subtype of **node** or **plan_variable**, trace down the tree from **node** or **plan_variable** to the subtype, and include the attributes of every entity on the path, in order.

4.2.5 Split-Join Complexes in an ALPS Plan

An ALPS plan may be viewed as a directed graph, where the successor-predecessor relationship of nodes provides the arcs between the nodes. On the directed graphs shown in this section, the arcs are straight lines, and the predecessor is higher up on the page than the successor. Imagine that gravity pulls from predecessor to successor. An example of an ALPS plan is shown in Figure 5.

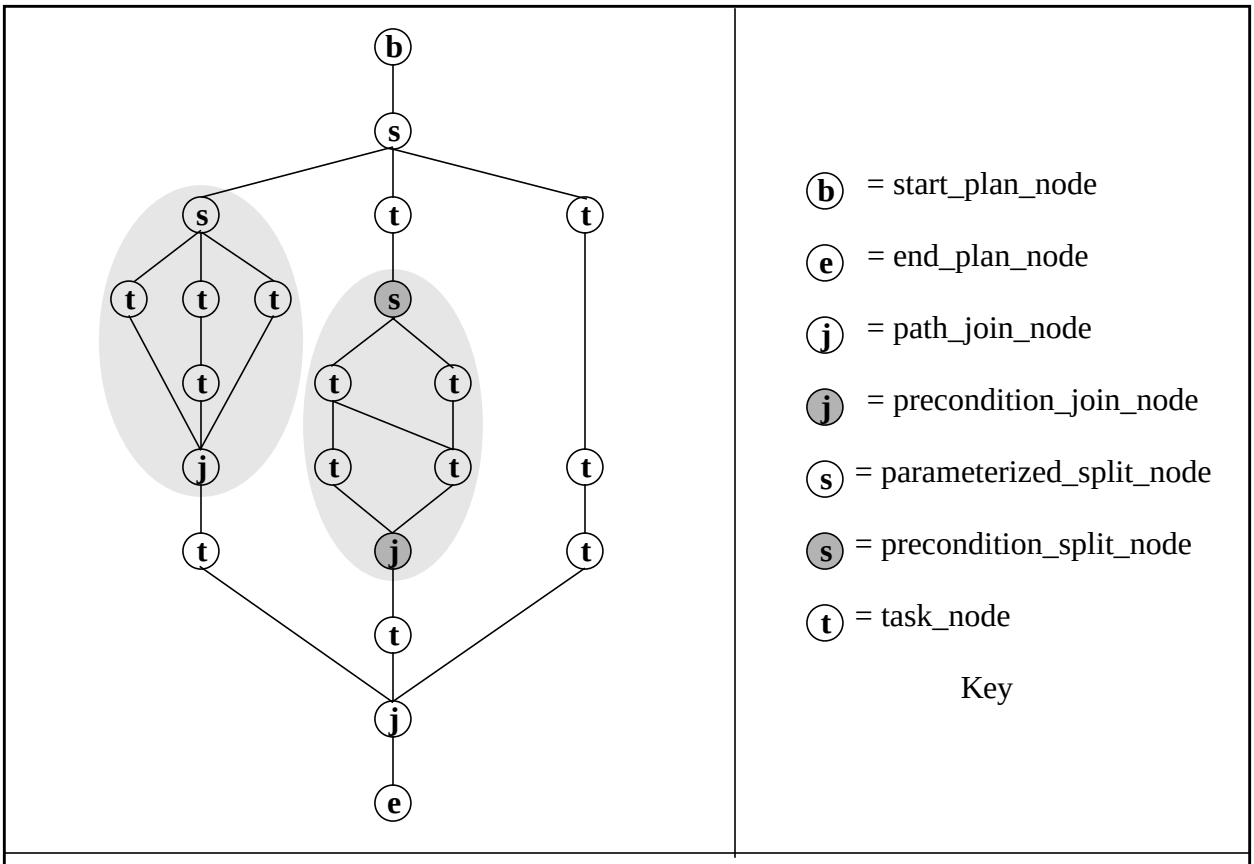


Figure 5. ALPS Plan Example

A key substructure, “split-join complex”, may be defined. A split-join complex is a split_node and its matching join_node plus all the nodes between them. Two business rules of ALPS enable the notion. The first rule is that all the paths leaving a split_node must eventually rejoin at the same join_node. The second rule is that if split_node B is on a path starting from split_node A, and B is before the join_node of A, the join_node of B must occur before the join_node of A. The second rule ensures that split-join complexes do not overlap, i.e., things like the path splitA—splitB—joinA—joinB do not occur. There are two types of split-join complex, precondition or parameterized, depending upon whether the split is a precondition_split_node or a parameterized_split_node. A precondition_join_node matches a precondition_split_node, and a path_join_node matches a parameterized_split_node. The two shaded areas on Figure 5 are each split-join complexes. Two split-join complexes are shown in Figure 6. A parameterized split-join complex is shown on the left in Figure 6A, and a precondition split-join complex is shown in the middle in Figure 6B. Figure 6A has the same structure as Figure 5 with the start_plan_node and

end_plan_node of Figure 5 removed and the two contained split-join complexes collapsed into single symbols.

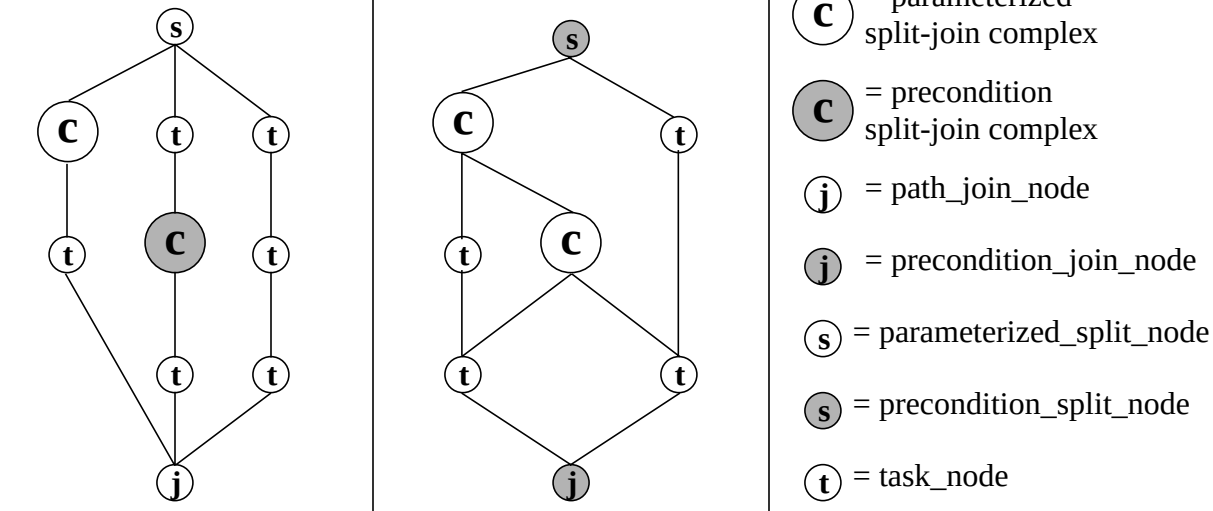


Figure 6. Split-join Complexes

For a parameterized split-join complex, an example of which is shown in Figure 6A, the complex must consist of single threads between the split and the join. No cross-linking between threads is allowed. The items on the threads are either task_nodes or split-join complexes. The split-join complexes could be replaced by suitable sub-diagrams. The threads may be of different lengths. Either parameterized or precondition split-join complexes may occur inside a parameterized split-join complex.

For a precondition split-join complex, an example of which is shown in Figure 6B, cross-linking is allowed (i.e., each task_node or split-join complex may have multiple predecessors and/or multiple successors). The items inside the complex are either task_nodes or parameterized split-join complexes. The parameterized split-join complexes could be replaced by suitable sub-diagrams. Precondition split-join complexes may not be nested, i.e., one may not occur immediately inside another. Nested precondition split-join complexes can be un-nested by erasing the inner split and join and connecting the path across the erasures in an obvious way, so not allowing their nesting does not impose a major limitation[1].

Inside a precondition split-join complex a path_join_node may have multiple successors, and a parameterized_split_node may have multiple predecessors[2].

1. It might be a good idea to get rid of the no-nesting rule, since nesting does not interfere with the operation of the algorithm, and allowing nesting would give the user a method of limiting the ways in which a plan could be traversed, a desirable functionality.
2. The EXPRESS model should be modified to reflect this, but has not yet been so modified.

4.2.6 Traversing an ALPS Plan

This section describes the algorithm used in FBICS for traversing an ALPS plan. This algorithm is used by both the Cell Planner and the Work Planner. They both use it when executing a stage-one plan and when making a stage-two plan from a stage-one plan. It is the most complex algorithm used in any part of FBICS. The plan traversal algorithm described here handles the type of plans FBICS generates. This algorithm cannot traverse all ALPS plans. The ALPS language can express plans that the FBICS plan traversal algorithm cannot traverse.

The traversal algorithm deals with split-join complexes. Once the algorithm starts processing a split-join complex, it finishes processing that complex before continuing processing anything outside the complex. Processing a complex requires processing any complexes nested inside it, of course.

The way FBICS generates ALPS plans, each task_node in a plan should be executed exactly once. A split_node and its matching join are considered executed when all the nodes inside the split-join complex headed by the split_node are executed.

The split_plus class is defined to help with traversal. A split_plus contains information specific to the split-join complex of one split_node. A split_plus object has the following data members.

1. affectables: the set of nodes in the split-join complex whose readiness to execute may be affected by executing the current node.
2. done_nodes: the set of nodes in the split-join complex that have been executed.
3. next_splits: the set of split_plus's built on the split_nodes that occur (at the top level) in the split-join complex.
4. previous_split: the split_plus in whose next_splits this split_plus occurs.
5. ready_nodes: a list of nodes in the split-join complex that are ready to be executed.
6. the_join: the join_node that matches the split_node on which this split_plus is built.
7. the_split: the split_node on which the split_plus is built.
8. waiting_nodes: the set of nodes in the split-join complex that have not been executed and are not ready to be executed.

The Cell Planner and Work Planner world models each include three attributes used for plan traversal. These are:

1. current_plus: the split_plus for the split-join complex currently being processed.
2. first_plus: the first split_plus (i.e., the one at which processing starts).
3. plan_status: one of NOT_STARTED, STARTED, or ENDED.

The nodes of an entire ALPS plan do not comprise a split-join complex, since the first must be a start_plan_node and the last must be an end_plan_node. We would like to treat the entire ALPS plan as a split-join complex so that special functions to handle the whole plan need not be written. Hence, after the plan is read but before traversal, a parameterized_split_node is placed before the start_plan_node, and a matching path_join_node created. This makes the entire plan structure be a split-join complex. The added split is made the split_node of the first_plus in the planner's world model. Unlike most other split_nodes, the added split_node has only one successor, which is the start_plan_node. This is contrary to the ALPS business rules for split_nodes, but the added split_node is never tested for conformance to that rule.

To support traversal during execution, before execution starts, the entire graph is traversed, a

split_plus is built for each split_node in the plan, and the data members of each split_plus are initialized. The initial settings of the data members are obvious except for affectables, ready_nodes, and waiting_nodes. The affectables and ready_nodes are initialized to be empty for both types of split_node. The waiting_nodes are initialized differently for precondition_split_nodes and parameterized_split_nodes. For a precondition_split_node, the waiting_nodes are all the nodes in the split-join complex other than the split_node on which the complex is built. For a parameterized_split_node, the waiting_nodes are the successors of the split_node.

The pre-processing just described is itself somewhat complex. The details are not given here but may be found in the documentation of the functions that do the preprocessing: build_split_next, build_split_param, and build_split_precon. These functions call themselves and each other recursively.

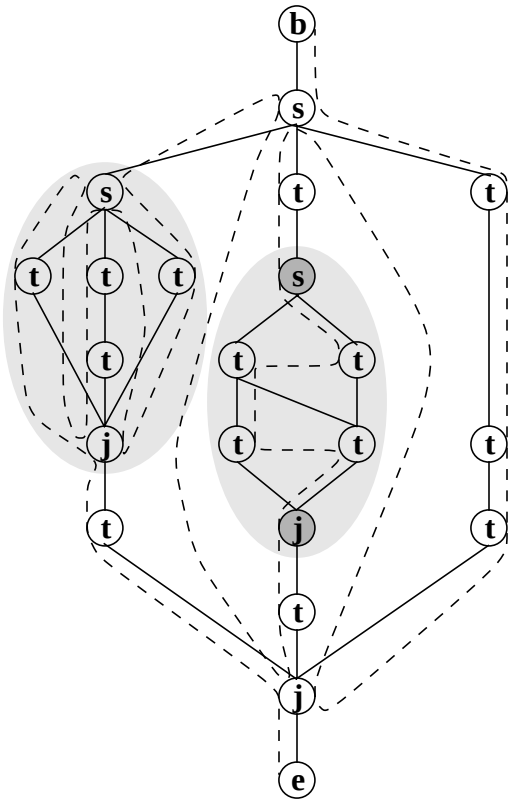
To envision traversal during execution, it is useful to imagine that the traverser is a thing (a bee, perhaps) that flies from node to node executing nodes and often updating the split_plus of the split-join complex for the node just executed. One possible traversal of the plan of Figure 5 is shown in Figure 7 below. The plan contains an outer parameterized split-join complex which has one parameterized split-join complex, one precondition split-join complex, and six primitive_task_nodes inside it. It is instructive to follow the traversal shown in Figure 7 and compare it with the description below.

An ALPS plan is traversed during execution when repeated calls are made to the cellpl_next_op1 interface function (for a cell-level plan, see Section 7.3.6) or to the workpl_next_op1 interface function (for a work-level plan, see Section 8.4.6). An ALPS plan may also be traversed during planning. In all cases, the get_next_node function is eventually called to find the next node requiring action. Appropriate actions are taken when the next node requiring action is found. The split_nodes and join_nodes are purely navigation nodes and require no action. The get_next_node function calls get_next_aux repeatedly to find the next node of any sort, which we will call the “next_node”. Get_next_node returns when the next_node is not a split or join. Get_next_aux calls one of (1) get_next_start (if the world model plan status is NOT_STARTED), (2) get_next_precon (if status is STARTED and the current_plus in the world model is built on a precondition_split_node) or (3) get_next_param (if status is STARTED and the current_plus in the world model is built on a parameterized_split_node).

The procedures used by these functions are described in the immediately following subsections. These functions all perform half a dozen or so checks (not described here) in addition to behaving as described. Each of these functions chooses the next_node. Each function prints the node number of the next_node in the user interface window so the user can see how the traversal is progressing.

4.2.6.1 Get_next_start

When get_next_start is called, the current_plus is first_plus (recall that current_plus and first_plus are kept in the planner’s world model). The first (and only) successor of the_split of the first_plus is a start_plan_node. The next_node is set to this start_plan_node. Then the function changes plan_status to STARTED and updates the data in the first_plus. In particular, it adds the start_plan_node to the ready_nodes of the first_plus. The current_plus is left set to the first_plus.



- - - - - = traversal path

b = start_plan_node

e = end_plan_node

j = path_join_node

j = precondition_join_node

s = parameterized_split_node

s = precondition_split_node

t = task_node

Key

Figure 7. ALPS Plan Traversal Example

The dashed line shows one possible traversal of the ALPS plan of Figure 5. The path starts at the top and ends at the bottom. The nodes are processed in the order shown by the path.

4.2.6.2 Get_next_precon

The data in current_plus is updated by checking the affectables and moving any whose predecessors are all in the done_nodes from the waiting_nodes to the ready_nodes. Then one of the ready nodes is selected to be the next_node. FBICS is currently just taking the first of the ready_nodes (i.e., not trying to optimize the selection in any way). The affectables are set to the successors of the next_node[1], and the next_node is moved from the ready_nodes to the done_nodes. The rest of what happens depends on what sort of node the next_node is.

A. If the next_node is a parameterized_split_node, the split_plus built on it is found (by searching through the next_splits of the current_plus) and the affectables of the current_plus are reset to be the successors of the join matching the next_node. Then the current_plus is reset to be the split_plus that was just found; this has the effect of starting the traversal on the split-join complex headed by the next_node, where the traversal will stay until that complex is entirely traversed.

B. If the next_node is a precondition_join_node, that indicates traversal of the current split-join complex is completed and it is time to go back to the split-join complex containing the one just

1. Resetting affectables here does not appear to be necessary since they are reset below, if needed.

traversed. So the current_plus is reset to be the previous_split of the current_plus. The next_node is added to the done_nodes of the new current_plus, and the affectables of the new current_plus are set to be the successors of the next_node.

C. If the next_node is not one of the two types above, it must be a machining or inspection node. In this case the affectables of the current_plus are set to be the successors of the next_node.

Some of the activities just described are performed by the subordinate functions update_precon and find_plus_match.

4.2.6.3 Get_next_param

For a split_plus built on a parameterized_split_node, the ready_nodes list is used differently. The ready_nodes are either empty (to signal that the traversal is at the split_node heading the current split-join complex) or contain just one node: the last node done.

If the ready_nodes are empty, the traversal is at the split_node. Pick any of the waiting_nodes (which, you will recall, contains successors of the split_node) to be the next_node, remove it from the waiting_nodes, add it to the ready_nodes, and add it to the done_nodes. As with get_next_precon, FBICS is just picking the first of the waiting_nodes, not optimizing its selection. This choice has the effect of picking a thread from the split_node to the join_node for execution.

If there is one node in the ready_nodes, it must have only one successor. Set the next_node to be the successor, replace the one node in the ready_nodes with the next_node, and add the next_node to the done_nodes.

The rest of what happens depends on what sort of node the next_node is.

A. If the next_node is a split_node, the split_plus built on it is found (by searching through the next_splits of the current_plus). The ready_nodes of the current_plus are set to contain the single successor of the join matching the split. If the next_node is a precondition_split_node, the affectables of the split_plus built on the next_node are reset to be the successors of the next_node. Then the current_plus is reset to be the split_plus that was just found; this has the effect of starting the traversal on the split-join complex headed by the next_node, where the traversal will stay until that complex is entirely traversed.

B. If the next_node is a path_join_node, it must be the one at the end of the split-join complex being processed. In this case:

1. If there are no more waiting_nodes of the split_plus for the complex, traversal of the complex is completed, and it is time to go back to the split-join complex containing the one just traversed. So the current_plus is reset to be the previous_split of the current_plus. The next_node is added to the done_nodes of the new current_plus, and the affectables of the new current_plus are set to be the successors of the next_node.
2. If there is still at least one node in the waiting_nodes of the split_plus for the complex, empty the ready_nodes of the current_plus, and print the node number of the split_node at the head of the complex to indicated that it has been revisited.

C. If the next_node is an end_plan_node, set the plan_status to ENDED. The functions that use the ALPS plan traverser know from getting an end_plan_node that the plan traversal is finished.

D. If the next_node is not one of the three types above, it must be a machining or inspection node. In this case nothing further is done in this function.

Some of the activities just described are performed by the subordinate functions update_param and find_plus_match.

4.3 Shape Representation

4.3.1 Types of Shape Representation in FBICS

As has already been discussed, FBICS is feature-based and uses STEP AP 224 features as the stored representation for part shapes and as the basis for process planning for both machining and inspection at three hierarchical planning levels. Section 4.3.2 provides information about AP 224.

For representing parts, features, fixtures, and access volumes in the Graphic Display, FBICS uses a simple facetted shape representation (see Section 12.4).

For dealing with shape during operation, FBICS includes software that drives the Parasolid solid modeler from AP 224 features and calls on Parasolid for answering solid model queries. Outside of the Modeler, FBICS does make use of any knowledge of Parasolid. Parasolid could be replaced by some other solid modeler without requiring any change in code outside the Modeler. The Parasolid boundary representation is not discussed further in this document.

The DMIS language for inspection programs includes shape features (see Section 12.2.5) such as plane and cylinder. These are defined from AP 224 features while generating DMIS programs in the Fbics_Task process and are used by the DMIS interpreter in the Fbics_Task2 process.

It would be desirable to be able to use a boundary representation with tolerances as the original input to FBICS and to be able during process planning to define AP 224 features giving the same shape. FBICS does not currently have this capability. There is currently no standard for a boundary representation with tolerances. It would be a good idea to modify AP 203 to include tolerances to meet the need for such a standard.

4.3.2 STEP AP 224

STEP AP 224, the ISO standard for “Mechanical Product Definition for Process Planning Using Machining Features” [ISO3] is being used in FBICS. AP 224 is largely a specification of a library of parametric machining features, but also provides for defining machining features in terms of a boundary representation and provides for related data, such as design_exceptions and requisitions. FBICS uses the library but does not use the boundary representation method. The library defined in AP 224 is quite similar to that defined in [Kramer9].

The AP 224 feature library provides 51 parametric “manufacturing_features” including, for example: boss, chamfer, circular_pattern, compound_feature, counterbore_hole, countersunk_hole, edge_round, fillet, general_pattern, general_pocket, groove, hole, pocket, rectangular_pattern, rectangular_pocket, slot, spherical cap, thread. The AP 224 feature types implemented in FBICS are round hole, counterbore hole, and rectangular pocket. Implementations of many other feature types from AP 224 are expected to be feasible.

The AP 224 feature library provides rich methods of describing details of features. Thirteen types of (planar) feature_profile are provided, for example. Feature_profiles may be swept in various ways to produce manufacturing_features. Twenty-nine types of feature_definition_item are

provided. This includes feature details such as hole_bottom_condition, path, and taper.

AP 224 provides two general types of tolerance: (1) tolerances on numeric parameters and (2) geometric and dimensional tolerances. In the first case, the tolerance is an attribute of a parameter which is an attribute of a feature, so that it is easy to trace through the data structure from the feature to the tolerance. In the second case, each tolerance is a self-standing entity which may be applied to more than one feature, and several tolerances may apply to the same feature, so that examining the data structure to determine exactly which tolerances apply to a given feature may be a major undertaking.

The tolerances on numeric parameters appear to be a relatively new concept. Many attributes of AP 224 features are of type numeric parameter, and any numeric parameter may be a numeric parameter with tolerance. The meaning of the tolerance in a numeric parameter with tolerance (in terms of what measurements indicate an in-tolerance or out-of-tolerance condition) is not explicitly described. Presumably, the meaning varies according to the type of feature and the attribute of the feature. In many cases, a compelling interpretation will probably be evident in terms of tolerance zones.

The geometric and dimensional tolerances are a full suite of tolerances (from ASME Y14.5). They appear to be enough to cover all or almost all expected needs for representing tolerances on piece parts.

AP 224 STEP Part 21 files are used in FBICS for designs of piece parts (starting, finished, or partially finished) and for features. Each file is expected to define a single (AP 224) Part. The shape of a piece part is determined by its base shape as modified by a list of features. When an AP 224 file is used to describe a design in FBICS, the features are assumed to be closed solids which are boolean subtracted from the base shape to determine the Part shape. When an AP 224 file is used to contain features to be removed or inspected, the base shape is simply ignored.

AP 224 also provides one additive feature, boss. Additive features are not usable as primary features in a machining-feature model because an additive feature cannot be created by machining. Additive features may be usable as secondary features which are subtracted from machining features before the machining features are subtracted from the workpiece. FBICS does not deal with the AP 224 boss feature.

4.3.3 Part_out Shape

The part_out shape (i.e., the design of the finished part) is given in a STEP Part 21 file based on AP 224.

The part_out shape description may include explicit tolerance information. Tolerances on parameters, but not geometric and dimensional tolerances are used in FBICS.

4.3.4 Part_in Shape

The part_in shape (i.e., the design of the incoming part) is given in a STEP Part 21 file based on AP 224. At the cell level, for machining planning, the name of a file describing the part_in shape must be given, but a flag indicates whether to read the file or write it. If the Cell Planner writes the part_in file, the shape described by the file is the base shape of the part_out.

In a real shop, actual incoming parts may have rough or significantly misaligned surfaces — the

cut surfaces of blocks sawed from bar stock, for example. In addition, the surface of stock is often unsuitable as the surface of the actual outgoing part (because it is rough, oxidized, heat hardened, etc.) The FBICS implementation described here does not try to deal with these problems. We assume that the incoming workpiece is nominal and its surfaces are fine. If any tolerance information is provided in the design of the incoming part, it is ignored.

4.3.5 Feature Shape

Feature shapes are given in STEP Part 21 files based on AP 224. Features are modeled as 3D solids in the Modeler.

4.3.6 Access Volume

An access volume is a volume of space related to a feature. The access volume of a feature is the volume through which a cutting tool or probe must pass in order to get access to the feature from the +Z direction for machining or inspection. The access volume for a feature is built by sweeping the top face of the feature (the planar face bounding the feature in the +Z direction) in the +Z direction of the native coordinate system of the feature far enough to be clear of the part. As implemented, “far enough” is 1.01 times the length of the diagonal of the bounding box of the part. Access volumes are built and manipulated by the Modeler and displayed by the Graphic Display while FBICS is running, but access volumes are not saved. The usual check FBICS makes is that the access volume must not intersect the part_now before the feature can be machined or inspected.

4.3.7 Block Bodies

Block bodies is a set of one or more volumes of material related to a feature. The block bodies of a feature contain the volume of material that is actually present in the feature and might block access to other features. The block bodies of a feature are found by taking the intersection of the feature with the part_in. The block bodies may be one solid or several disconnected solids.

4.4 Inspection Planning

Inspection planning is quite different from machining planning. Deciding which features to machine is relatively easy; any feature not on the part_in needs to be machined. Deciding which features to inspect (and when to inspect, or even why to inspect) is harder. This section discusses the issues considered in determining how FBICS implements inspection planning.

The approach to these issues taken in FBICS is that there is no “right” way to make the decisions, so let the user decide. This is implemented by having options files that are read in at initialization time and (if options specify it) by asking the user on a case-by-case basis. There are no national or international standards applicable to these decisions. It is common for a company or shop to have guidelines for making them, however, so having user options is appropriate.

4.4.1 Reasons for Inspection

Inspection (of a finished part or workpiece) might be performed for any of several reasons, including at least:

1. to see if the part was (or the workpiece is being) made to specification.
2. to control the process in progress.
3. to measure tool wear.

4. to determine if manufacturing procedures are working.
5. to determine if process planning methods are working.

The second of these, to control the process in progress, requires that feedback from inspection be obtained and used for process control while the workpiece is being manufactured. FBICS implements this as described in Section 3.7.7.

Different inspection plans might, in principle, be made depending on which of the other reasons obtains, but FBICS does not implement this.

4.4.2 Deciding Which Features to Inspect and How Thoroughly

Selecting features to inspect and deciding how thoroughly to inspect each one is a deep problem.

The primary considerations in determining whether to inspect a feature are the cost of inspection, the cost (production cost and downstream cost) of making a part incorrectly, and the likelihood of making the part incorrectly. The track record of the system may be important in estimating these things.

To determine in how much detail to inspect a feature, the machining process as well as the tolerance specifications must be considered. For example, if a drill is used to make a hole and both position and straightness of the hole are important, it will often be a good idea to inspect a lot of points at various depths in the hole. If the hole is made by a first drilling a slightly smaller hole and then reaming, the straightness of the hole will be much less likely to be out-of-spec, so it may suffice to inspect fewer points, and all the points might be at the same depth.

FBICS does not go deeply into deciding what to inspect or how thoroughly, but provides options for the user. Thus, FBICS allows the user to do the deep thinking and provides a way for the user to get FBICS to do what the user wants.

The FBICS choices for which features to inspect are (a) inspect all features, (b) inspect no features, (c) inspect a feature if any attribute has any tolerance, (d) inspect a feature if any attribute has a tight tolerance, or (e) let the user decide which features to inspect on a case-by-case basis.

Two options are used to specify how thoroughly to inspect. FBICS uses the qualitative idea of high, medium, and low inspection intensity. This is given in the inspecting_level option in Section 12.1.6.4. The user must specify in the task options file how many points to inspect at each of these three levels for each DMIS feature type, as provided in Section 12.1.7.11 through Section 12.1.7.16.

4.4.3 Deciding When to Inspect, During Machining

The two extremes of when to inspect features while machining a part are to alternate cutting and measuring features and to cut all the features before measuring any. Between the extremes are many choices.

The advantages of cutting a lot before inspecting are that fewer tool changes swapping probe and cutter are required and that less time will be needed to clean the part and/or the probe (or other sensor). Cleaning the portion of the part to be inspected before inspecting it is almost always necessary. Having metal chips between the sensor and the part is sure to make the inspection very inaccurate. Getting chips or coolant on the sensor will foul up attempts to inspect a clean part. The process plan might be written to include further machining if the inspection shows a correctable

out-of-spec condition. Or, the system might be given the capability to plan further cutting after an out-of-spec condition is detected. The main disadvantage of postponing inspecting until after a lot of cutting is that an uncorrectable out-of-spec condition might have been created early in the cutting, so that the time and cost of the subsequent cutting is wasted.

Conversely, the main advantage of inspecting more frequently is avoiding wasting time after an uncorrectable condition has been created. A second advantage is that performing a correcting cutting operation may be easier (or only feasible) before further machining occurs. For example reaming a whole hole will produce a better surface than attempting to ream the hole after part of its surface has been cut away (by making an intersecting feature).

FBICS currently inspects each feature immediately after cutting. This wastes time, since doing it requires that the tool be changed and the coolant turned on or off before and after every cutting operation. An option for how many features to queue up for inspection before switching from machining to inspection has been defined and must be included in shop options files (see Section 12.1.6.3), but the information is not used in the current implementation.

4.4.4 Selecting Inspection Points

Each face of a feature to be inspected is inspected separately, since each face has different geometry. Selecting points for probing a face is not a trivial problem. We assume in this section that the number of points for inspecting a face has been decided.

FBICS regards a feature as a volume of space that is empty of material. FBICS makes no assumptions about the appearance on the workpiece of faces of the feature. In the simplest cases, all the faces of the feature except the top (entry) face will appear on the workpiece. In general, however, (1) if a feature extends beyond the initial stock, any face on the portion that extends out will be reduced or not appear, (2) if feature A intersects feature B, any face of A that intersects B will either not appear at all or have holes in it.

The general approach in FBICS for inspecting a face is to pick candidate probe points one by one, test each candidate point as it is picked to be sure it can be probed safely, keep the points that can be probed, and discard the ones that cannot be probed until the desired number of probe points has been picked (giving up if too small a proportion of the candidate points are safe to probe). The algorithm that picks candidate points picks them roughly uniformly distributed all over the face, regardless of how many candidate points must be picked (see Section 4.4.5).

It is not safe to attempt to inspect points that might not lie on material. FBICS checks that candidate probe points are (relatively) sure to be present on the workpiece with the help of the Modeler. It is not enough to know that a point is theoretically where it should be. Rather, it is necessary to check that an entire patch of surface around the point is all there. The patch should be big enough that it is fairly certain the probe will hit within the patch when it is aiming at the point.

Depending on the inspection situation, the patch may be a circle or a rectangle. A circle is appropriate if the workpiece being inspected is likely be incorrect (out of spec or out of position) equally in both directions perpendicular to the probing direction. A rectangle is appropriate if error is likely to be greater in one direction than another. This will happen, for example, if a workpiece is lying flat on a table but is otherwise inexactly located. In that case, for any point on the workpiece, the expected error in its position parallel to the table is much larger than the expected error in its position perpendicular to the table (assuming position errors dominate

manufacturing errors). Messages to the Modeler telling it to check that a circle or rectangle lies on the workpiece are described in Section 10.2.3 and Section 10.2.20, respectively.

4.4.5 The Inspection Point Selection Algorithm

The following algorithm was devised for selecting candidate inspection points on a rectangular portion of a plane so that the points are roughly uniformly distributed over the rectangle. Since a cylinder may be formed by rolling up a rectangle, the same algorithm can be used (followed by the mapping from a rectangle to a cylinder) for selecting points on a cylinder. The points will be roughly uniformly distributed on the cylinder.

If portions of the rectangle or cylinder for which candidate points are being found are missing, so that many candidate points are rejected, no significant problem arises. Additional candidate points are easily generated, and they will be roughly uniformly distributed on whatever portion of the surface is present. Thus, the algorithm may be used, for example, for finding points on any planar face of a part. A rectangle is defined that fits around the face, candidate points are found in the rectangle, candidate points are tested (using the Modeler) for actually being on the face and discarded if not there, and additional candidate points are generated until the desired number have been found.

For the first 4 points, the rectangle is divided by vertical and horizontal lines through the center into four half-size rectangles and points are placed in the centers of the half-size rectangles. For the next 16 points the four half-size rectangles are each divided into four quarter-size rectangles and points are placed in the centers of those rectangles. And so on. Each subsequent subdivision yields four times as many points, so the next subdivision, for example, yields 64 points.

The order in which the points are placed in the smallest rectangles at each stage is determined as follows. Imagine a base-four counter, with as many places as needed, that starts at zero and has the upper places blanked out. It has the unusual property that when a blank place should change to 1, it changes to 0 instead. Otherwise, the counter behaves normally. The appearance of the counter is shown in Figure 8 for the first 92 counts. The surprises are at 5 (which would ordinarily be `10`, since it follows `3` on a base-4 counter), 21 (which would ordinarily be `100`, since it follows `33` on a base-4 counter), and 85 (which would ordinarily be `1000`, since it follows `333` on a base-4 counter).

The counter digits for the nth point give the point location as follows. Imagine the four half-size rectangles are numbered 0, 1, 2, 3, starting from the lower right and going counterclockwise. The units digit on the counter for the nth point gives the number of the half-size rectangle in which the point is found. If the fours digit for that point is blank, the point is at the middle of the half-size rectangle. If the fours digit is not blank, the half-size rectangle is divided into four quarter-size rectangles that are numbered the same way, and the point is found in the quarter-size rectangle whose number is the fours digit. If the sixteens digit for that point is blank, the point is at the middle of the quarter-size rectangle. If the fours digit is not blank, the quarter-size rectangle is divided into four eighth-size rectangles that are numbered the same way, and so on.

For example, take the 49th point. Its counter digits (as seen in Figure 8) are `130`. The units digit is `0`, so the point is in the lower right half-size rectangle. The sixteens digit is `3`, so the point is in the lower left quarter-size rectangle (of the half-size rectangle). The sixty-fours digit is `1` and there is no 256s digit, so the point is in the center of the upper right eighth-size rectangle (of the quarter-size rectangle). This description matches the location of 49 on Figure 9.

```
 1.     0        24.  003        47.  122        70.  301
 2.     1        25.  010        48.  123        71.  302
 3.     2        26.  011        49.  130        72.  303
 4.     3        27.  012        50.  131        73.  310
 5.    00        28.  013        51.  132        74.  311
 6.    01        29.  020        52.  133        75.  312
 7.    02        30.  021        53.  200        76.  313
 8.    03        31.  022        54.  201        77.  320
 9.    10        32.  023        55.  202        78.  321
10.    11        33.  030        56.  203        79.  322
11.    12        34.  031        57.  210        80.  323
12.    13        35.  032        58.  211        81.  330
13.    20        36.  033        59.  212        82.  331
14.    21        37.  100        60.  213        83.  332
15.    22        38.  101        61.  220        84.  333
16.    23        39.  102        62.  221        85. 0000
17.    30        40.  103        63.  222        86. 0001
18.    31        41.  110        64.  223        87. 0002
19.    32        42.  111        65.  230        88. 0003
20.    33        43.  112        66.  231        89. 0010
21.   000        44.  113        67.  232        90. 0011
22.   001        45.  120        68.  233        91. 0012
23.   002        46.  121        69.  300        92. 0013
```

Figure 8. Point Counter

The placement of the points for the first 84 (= 4 + 16 + 64) points is shown on the rectangle in Figure 9. On the figure, the three sets of points are shown in successively smaller fonts.

A C language implementation of the algorithm, much simpler than one might expect, is shown in Figure 10. After the function runs, the x-value of the point is in `answer[0]` and the y-value in `answer[1]`. The implementation is iterative. A recursive implementation has been built (with three functions rather than one). The functions are even simpler, but net result is more obscure and less efficient.

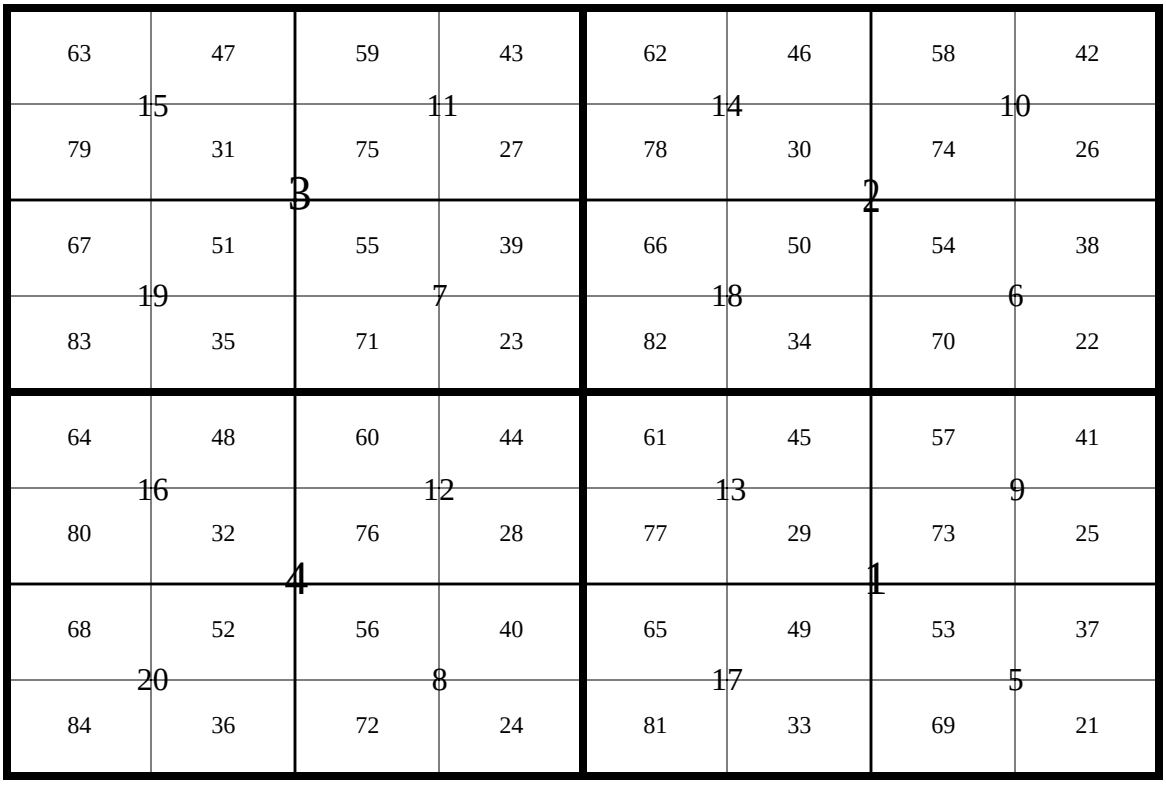


Figure 9. Point Placement

```
void find_inspection_point(/*  ARGUMENTS                     */
 double center_x,          /* rectangle X center            */
 double center_y,          /* rectangle Y center            */
 double x_size,            /* rectangle length              */
 double y_size,            /* rectangle width               */
 int nth,                  /* the ordinal of the point      */
 double * answer)          /* array of two doubles to be set*/
{
  int quad;

  answer[0] SET_TO center_x;
  answer[1] SET_TO center_y;
  x_size SET_TO (x_size / 2.0);
  y_size SET_TO (y_size / 2.0);
  for (; nth > 0; nth SET_TO (nth / 4))
    {
      nth SET_TO (nth-1);
      x_size SET_TO (x_size / 2.0);
      y_size SET_TO (y_size / 2.0);
      quad SET_TO (nth % 4);
      if (quad IS 0) OR (quad IS 1)
         answer[0] SET_TO (answer[0] + x_size);
      else
         answer[0] SET_TO (answer[0] - x_size);
      if ((quad IS 1) OR (quad IS 2))
         answer[1] SET_TO (answer[1] + y_size);
      else
         answer[1] SET_TO (answer[1] - y_size);
    }
}
```

Figure 10. Inspection Point Selection Algorithm Implemented in C

This algorithm has the following advantages:

1. It is easily implemented in half a page of simple code, as shown.
2. The location of the nth point is found without needing to know the location of any previously found points.
3. No three of any four consecutive points are collinear.

This algorithm has the disadvantage of picking points in a regular pattern. This is a disadvantage because machining errors are quite likely to occur in regular patterns, such as lobing in drilling or waviness in milling. The interaction of regular patterns may lead to misleading inspection results. This problem could be ameliorated by adding random noise to the point location on a scale about the size of the rectangle centered on the point.

4.4.6 Using AP 224 Features for Inspection

FBICS uses AP 224 machining features for inspection. Each inspection node in a process plan refers to an AP 224 feature. To “inspect a feature of the part” we inspect those faces of the part identified with the feature. The way this is done in FBICS is described below. Briefly, DMIS features are defined which coincide with the identified faces, and touch points for those DMIS features are selected with the help of the Modeler.

The tolerance information which may be carried in AP 224 features is also used in FBICS during process planning for inspection. How this is done is described in Section 8 and Section 9.

For each AP 224 feature type, it is necessary to decide what DMIS features should be defined to represent the AP 224 feature. In addition, it is necessary to decide how many points to measure when inspecting each DMIS feature.

The DMIS language defines the surface features plane, cylinder, cone, and sphere, among others. These are simpler than features defined in AP 224, which has, for example, pocket and hole. To generate DMIS code from AP 224 features, it is necessary to decompose the surface of each AP 224 feature into a set of DMIS features. The decomposition of AP224 feature types used in FBICS is discussed in the subsections below.

4.4.6.1 Partial Surfaces

The DMIS features that are present on an AP 224 feature may be completely or partially missing from the surface of the part, since AP 224 features may project outside of parts or intersect with other features.

For each DMIS feature present on an AP 224 feature, two types of check are made that the DMIS feature is actually present on the part and is inspectable.

1. A test is made using the Modeler that the bounded portion of the DMIS feature which might be present has a non-empty intersection with the surface of the part. This is done by making a thin shell on the bounded portion of the feature and intersecting it with the part. If the intersection is empty, the DMIS feature is discarded.
2. If the first test is passed, an attempt is made to find the number of required inspection points for the feature on the surface of the part. Candidate points are generated (see Section 4.4.5) and, using the Modeler, are tested. If a candidate point is not on the part, a different candidate is generated and tested. Candidate points will fail to be on the part in cases where a feature extends beyond the base shape of the workpiece or where another (already made) feature on the workpiece intersects the feature being inspected and has removed part of it. The generate and test procedure continues until either the required number of points has been found on the part or ten times the required number of points have been tested. In the latter case, the DMIS feature is discarded.

For each DMIS feature actually present and inspectable, as determined by the methods just described, DMIS code is written (1) to define the feature, (2) to measure the feature (using the points already found), and (3) to report the results of the measurement. If the feature is a cylinder with a diameter tolerance (inherited from a tolerance on the diameter of an AP 224 hole or on the corner radius of an AP 224 pocket), DMIS code is also written to define the tolerance, calculate the error in the diameter, and report whether the cylinder diameter is in-tolerance or not.

4.4.6.2 AP 224 Round_hole

An AP 224 round_hole is represented by a single DMIS cylinder. The general AP 224 round_hole may be tapered, so that its side surface is conical rather than cylindrical, but FBICS is not handling that case. The bottom of a round_hole is not inspected by FBICS.

4.4.6.3 AP 224 Counterbore

An AP 224 counterbore consists of two co-axial holes, one of larger radius and one of smaller radius. As used in FBICS, making a counterbore hole includes cutting only the larger hole. The smaller hole must already exist before cutting the larger hole is permitted. Extending this notion to inspection, only the larger hole is inspected. The larger hole is treated like a round_hole with no bottom, as described in Section 4.4.6.2.

4.4.6.4 AP 224 Rectangular_closed_pocket

A prototypical AP 224 rectangular_closed_pocket with no frills has, in general, a plane at the bottom, four planes for sides, and four quarter cylinders blending the intersections of the sides. There is a DMIS feature of each of those three types, so the general case of such a pocket will be represented by nine DMIS features. The general AP 224 pocket may have fillets at the bottom, but FBICS is not handling bottom fillets. Various degenerate geometries that occur for specific relations among the parameters of the pocket are not barred by AP 224. If the corner radius of the pocket is half its width, the pocket looks like what is usually called a slot. There are only two side planes and there are two half cylinders at the ends. If, in addition, the corner radius is half the length of the pocket, the pocket becomes a round hole with a flat bottom and is represented by one full cylinder and one plane. The DMIS generator in the Task Planner handles all of these cases.

4.5 Human Input

An observed (by the authors) phenomenon of trying to build automatic manufacturing systems is that if humans are able to participate (so that the system is not fully automatic), the range of parts that the system can handle satisfactorily is about ten times the range that can be handled fully automatically. To have the most effective system, therefore, human participation should be integrated.

Two strategies for facilitating human input are implemented in FBICS. The first strategy is to have a large number of system options in options files that users can modify. The files can be changed to suit a whole shop, to suit one user, or for a single part. Options are discussed in Section 12.1.6, Section 12.1.7, and Section 12.1.8.

The second strategy is to modularize the system in such a way that data files for all key data pass from module to module. These files provide a medium for facilitating human participation. Decisions made automatically by a module and embodied in a data file can be readily reviewed and revised. If a large amount of data were transferred by the messaging system, it would be more difficult to provide for suitable review by humans.

In addition to having a large number of types and instances of data files, an effort has been made to include all data a human might want to change in a file of some sort. Consider tool changes and coolant control, for example. These do not appear in stage-one work-level plans. During stage-two work-level planning, a tool change or coolant control is defined as a one_operation with a corresponding executable operation file. This makes it visible in files a human can edit. Tool

changes and coolant control could have been handled within the Task Planner without making them visible. If that had been done, there would have been no way for a human to deal with them at the work level, and they could only have been addressed in RS274 or DMIS code.

For the kinds of data files used in FBICS (STEP Part 21, RS274, DMIS, and graphics), it is feasible (but usually tedious and error-prone) for a knowledgeable user to edit the files with a text editor. It would be much better to have a user-friendly editor for each type of data. The current implementation of FBICS includes no such editors. In a commercial implementation, these editors would be essential.

4.6 Planning Issues

This section discusses some planning issues that are too interesting to omit. The reader concerned only with how FBICS works may skip it. Those interested in why FBICS works the way it does (or how it might be improved) should read it.

4.6.1 Plan Off-line or On-line

In general, planning (at any controller level) might be done off-line or on-line. In off-line planning, an entire plan for getting from the starting state to the end state is made before execution starts. In on-line planning, one step is planned at a time, with each step being executed before the next step is planned. Machining is an irreversible process, so on-line planning is a reasonable thing to do only if there is a high probability of being able to get to the end state, i.e., a low probability that it will be impossible to proceed from any intermediate state.

If the planning method being used includes backtracking (as is the case with many methods), it does not make sense to machine while planning, since the machining will prevent backtracking. For the lower levels of machine control, including generating a tool path to cut a feature, and planning how to keep a tool on a path, backtracking is not needed, so interleaving planning with execution is a reasonable thing to do. For the upper levels of control, including determining which features are to be cut in which setups, and in what order features may be cut, backtracking is needed, and for optimization, it is essential.

If off-line planning is done, in order to deal with the variability of conditions at execution time, it is useful to include both alternatives and variables in off-line plans. For an off-line plan at execution time, the system must determine the order of execution, assign values to variables, and select among alternatives. These are usually viewed as planning functions, so the planner may have substantial work to do at execution time executing an off-line plan.

When planning is being done on a computer, it may be completed very quickly. Thus, the practical difference between doing off-line and on-line planning may disappear. In both cases, the user may not have to start the planning process until it is time to make the part, and the total time from the start of planning to the end of machining may be about the same either way.

One big advantage of off-line planning is that the plan may be tested and improved. And, with off-line planning the plan is saved, so that planning need be done only once and many parts of the same design may be made using the plan.

4.6.2 Maximum Off-Line Planning Depth

The maximum planning depth in FBICS is down to (and including) the task level for stage-one

plans and down to (and including) the work level for stage-two plans. It is not generally feasible to plan off-line deeper than this, as discussed in the remainder of this section.

A stage-two task-level plan might be a sequential list of canonical machining commands (see [Proctor3]) or canonical inspection commands (see [Kramer17]). The RS274/NGC and DMIS interpreters in stand-alone FBICS make calls to these commands (implemented as functions), and the functions print themselves. Printing may easily be redirected to a file, so FBICS could be used to generate command files.

A command file for machining could be used as a stage-two task-level plan as long as it did not include feedback from inspection or operator data input (or any other feedback source). This has not been tried. A task-level command file reader would be required but none has been built. Any case of machining with feedback at the NC code execution level would not work using a command file because the feedback would not get used in the command file.

A command file for inspection could be used as a stage-two task-level plan only in unusual cases where measurement results do not feed forward. Once again, a command file reader would be needed.

The key difference between machining and inspection when dealing with stage-two task-level plans is that in machining, feedback from lower levels to the interpreter is generally infrequent and may not occur at all, whereas in inspection, such feedback is very frequent.

4.6.3 Distributed Planning Alternatives

An important choice in distributed off-line planning is the following. Consider any superior-subordinate pair in the hierarchy. When the system is planning and the superior is making a stage-two plan, the following two alternatives are available.

1. The superior might first plan fully for itself, assuming that the subordinate will be successful in creating whatever plans it needs. After the superior has completed its stage-two plan, the superior goes through that plan and issues planning commands to the subordinate. If any of the subordinate's attempts to make a plan fails, the superior must redo its stage-two plan and try again to have the subordinate do all its required planning.

2. The superior might interleave planning stage-two plan steps for itself with issuing planning commands to the subordinate. If a subordinate's attempt to make a plan fails, the superior (depending on the alternatives available in the stage-one plan) may be able to select a different alternative from the stage-one plan in making its stage-two plan step and issue a different planning command to the subordinate.

The pattern of planning over an entire hierarchy will be quite different depending on which alternative is chosen. Under the first alternative, all controllers in the chain of superiors extending up from any controller will have completed their stage-two plans before the controller plans for a given activity. Under the second alternative, all controllers in that chain of superiors will be in the process of making stage-two plans when the controller is planning for the same activity.

The first alternative is somewhat easier to implement. The second alternative seems better. It helps prevent wasting large amounts of planning effort. The backbone of the second alternative is implemented in the FBICS Cell Planner, but no method has been implemented of backtracking in the superior when the subordinate's attempt to plan fails. The first alternative is implemented in

the FBICS Work Planner, where it is appropriate because NC code generation and DMIS code generation, the planning performed at the next lower (task) level, do not often fail.

Other modes of planning might be considered that have not been implemented in FBICS. Two examples follow, both of which could be implemented readily at the user interface level. Other similar planning activities could also be implemented readily.

1. Start with an existing cell-level stage-one (or stage-two) plan and make a set of work-level plans to go with it.
2. Start with a design and make a cell-level stage-one plan and stage-two plan, but no lower-level plans.

4.6.4 Plan Validity

If the contents of a plan differ according to conditions at planning time, then, if the plan was made under one set of conditions, it may not be valid if executed under different conditions. A plan made off-line may have this problem.

The user is in control of some planning time conditions, such as the settings of options. Currently, this information is not recorded. It would be useful to modify FBICS to provide a place in plans for recording options settings; the names of the options files used would suffice.

Other planning time conditions are not controlled by the user. As an example, in current FBICS, during adaptive machining, the tool path for the final cut is hard-coded differently, depending on measurement data taken after the semi-final cut. This machining plan (NC program) would not generally be valid to use for cutting another similar part.

It would be useful to record whether or not it is valid to re-use a plan made on-line.

Observe that if a variable whose value is determined at execution-time is used in a plan, the plan is valid (with respect to that variable). In the adaptive machining case just mentioned, if the final tool path were expressed in the NC code in terms of an NC code variable depending on measurement data whose value is determined at execution-time, the plan (NC program) would be valid.

A method of extending the validity of a plan whose contents depend on conditions at planning time would be to record the value of any such condition in the plan. Then the plan would be valid whenever the conditions at execution-time match the conditions at planning time. In the case of adaptive machining, the measured data (corner radius, perhaps) would have its value recorded in the NC code, and the code could be reused whenever the measured data matched the saved data. That is a poor example because it would be silly to start machining taking a chance like that. A better example would be where the tool path depended on the ambient temperature. If the ambient temperature at execution-time were close to the ambient temperature at planning time, the plan would be valid.

5 FBICS Interfaces

5.1 Modules and Processes

A FBICS module is a set of related computer code (source code or object code) for performing functions of the system. In stand-alone FBICS, each of the three controllers includes a driver module (everything a controller needs except the planner). The Fbics_Cell and Fbics_Work processes each have a complete planner module in addition to the driver module. The Fbics_Task process has half of a planner, the off-line part. The other half of the Task Planner is in the Fbics_Task2 process, which has a DMIS interpreter module and an RS274/NGC interpreter module. The Fbics_Task2 process also has a driver, but it is merely a message-handling wrapper. The Fbics_Task2 driver reads messages from Fbics_Task, makes calls to the interpreters and returns status messages. The Fbics_Model and Fbics_Draw processes are each single modules. All of these modules are visible as areas inside boxes on Figure 1.

The planners and stand-alone drivers are kept in separate modules so that it will be easy to integrate the planners into other, more fully functional, controllers.

5.2 Types of Interface

FBICS uses three types of interfaces between modules: application programming interfaces (APIs), messaging interfaces, and file interfaces. An API is a set of functions calls available for use between modules. A messaging interface is a set of messages that may be sent between processes by interprocess communication methods. A file interface is a set of file types which may be written by one process and read by another. Stand-alone FBICS uses all three types of interface. An API is used when two functional modules are in the same process (a planner and the rest of its controller in stand-alone FBICS, for example), while a messaging interface or file interface is used when modules are in different processes.

For real-time operation with short cycle times (a characteristic of controllers in integrated FBICS), it is necessary that messages be kept short. Also, the programming overhead is high for defining and handling longer messages. For these two reasons, the messages used in stand-alone FBICS are all relatively short.

For communication between processes, stand-alone FBICS uses messaging interfaces supplemented by file interfaces. If content is simple (as in status messages from one process back to another, for example), the message contains all the information that needs to pass between processes. If content is complex (a machining operation plus the shape of a removal volume, for example), the sending process writes a file, then sends a message to the receiving process telling it to perform an operation and giving the name of the file. Upon receiving the message, the receiving process reads the file.

APIs typically allow a function call argument to be a reference to a complex type of data. Within a process, once a complex data structure is built, it is easy to use by reference. With a messaging or file interface, if the sender wants the receiver to do something with a complex data structure in the sender’s process, the data structure must first be packed into a message and sent or written to a file.

It is feasible to change an API into a messaging interface, in the event functional modules are put into separate processes. As just observed, however, if the API includes references to complex data, the time and effort required for communication between the caller/sender and the called/

receiver increases greatly when the change is made. For this reason, none of the function arguments of the APIs to the planners in FBICS is a complex data type. Where pointers (to simple data types) are used in an API interface and the called function sets the value of the data pointed at, in the equivalent messaging interface, the values are put into the status message returned by the called/receiver.

It is anticipated that, in advanced integrated FBICS, the planners may be in separate processes from the remaining parts of their controllers. Keeping the arguments simple allows for easy conversion from API interface to messaging interface.

5.3 FBICS APIs

The Cell, Work, and Task planners in FBICS each have an API for calls to the planner (see sections Section 7.3, Section 8.4, and Section 9.4, respectively). Functions names in these APIs start with “cellpl_”, “workpl_”, and “taskpl_”, respectively. All of the functions in the APIs return an integer indicating either OK, EXIT, or ERROR. In the event of an error, before returning ERROR, the planner will have printed a message describing the error.

The NIST DMIS interpreter and the three-axis NIST RS274/NGC interpreter included in the Fbics_Task2 process have their own APIs and the driver uses them. It is expected that those APIs will be used in advanced integrated FBICS, also.

In the section discussing the APIs, some function arguments are used to return data from the function called to the caller. Such arguments are always pointers and what they point at is set by the function. These arguments are underlined. Where arguments are file names, the use of path names (in the usual hierarchical directory system sense) is possible to some extent, but the extent is system-dependent. It is expected that relative path names starting in the directory from which a FBICS process is running will work in most operating systems. Using relative path names of this sort has been tested extensively in the current implementation of FBICS under the (UNIX-like) Solaris operating system. Using wild-card characters in path names does not work in the current implementation and is not expected to work under other operating systems.

5.4 Message Interfaces

5.4.1 Introduction

FBICS messaging interfaces are interfaces between processes. The interfaces are implemented by interprocess communication. All of them use the Neutral Message Language (NML) messaging capabilities built in the NIST EMC project [Shackleford]. A message interface may connect a module within one process to a module within another process without being known by other modules in those processes.

There are three aspects to the FBICS messaging interfaces: (1) what the messages are, (2) which FBICS modules are connected via messages, and (3) what the message protocols are (i.e., which messages are sent between which parties in which circumstances). Comparing stand-alone FBICS and integrated FBICS, there will be a subset of modules for which these aspects are the same in both, and a subset that differ. The subset for which these will be the same includes the three planners, the Modeler, and the Graphic Display. The subset for which they will differ is the controller bodies (the parts outside the planners).

In stand-alone FBICS, the user interfaces to the controllers are implemented by direct connection

between the controller process and the keyboard, as described in Section 7.2 (Cell), Section 8.2 (Work), and Section 9.2 (Task). If the user interface were implemented as a process separate from the rest of the controller, as is commonly done, additional messages would be required. Connecting the user interface by messages is superior in several ways to direct keyboard input and would be a good idea for future versions of FBICS.

5.4.2 NML Messaging

The NML messaging system provides an interprocess communications API in the C++ language that is the same regardless of how far separated the processes are. Processes may, for example, be in a single processor, on separate processors sharing a single memory board, or on different computers attached to the same net. The applications programmer uses the same communication function calls, regardless.

The principal NML abstract notions include: process, mailbox, message type, and message instance. Some number of mailboxes and message types are defined for a system involving multiple processes. For each mailbox, there is a list of processes, identified by name, which can use the mailbox. Each mailbox can hold (at least) one message instance at any time. Each process may be allowed to read a message instance in the mailbox, allowed to write a new message instance and put it in the mailbox, or allowed to do both. Each process exercises its access to the mailbox whenever it wants to. A message instance of any defined message type may be put into any mailbox. There is an option for being able to put several message instances in a mailbox in a queue.

The applications programmer writes three files to enable the use of NML:

1. A configuration file that names the mailboxes, specifies which processes may read or write the contents of each mailbox, and specifies how each process is connected to the mailbox.
2. A C++ header (.hh) file defining the message types that may be used by the application.
3. A C++ code (.cc) file defining an update function for each message type, plus a single format function.

The configuration file is used at run time by the NML system. The other two files are compiled and linked into the application at compile time, along with the NML library. If the system is reconfigured by moving a process from one computer to another, only the configuration file needs to be changed. Recompiling is not necessary.

NML has been implemented for many computers and operating systems. For FBICS, it has run on Sun Computers running Solaris, PC’s running Windows NT, and Silicon Graphics computers running IRIX.

For more information on NML, see [Shackleford].

5.4.3 Module Connections

The connection of modules is along the solid straight lines shown on Figure 1. The connections are directional, as shown on the figure, in the sense that the process on the upstream end of the line is giving a command to the process on the downstream end (where the arrowhead is). The upstream process sends its command message down the line, and the downstream process

executes the command then sends a status message back up the line. Each line shown on the figure is implemented by two mailboxes, one for command messages (written by the upstream process and read by the downstream) and one for status messages (written by downstream and read by upstream).

As shown on Figure 1, the Modeler is connected directly to the planner inside each controller. The other portions of those controllers do not communicate with the Modeler. The Graphic Display communicates only with the Modeler. These connections will be the same in integrated versions of FBICS as in the stand-alone.

5.4.4 Message Protocols

A message protocol specifies which messages are sent between particular modules under what conditions. A message protocol may specify standard message sequences and timing. FBICS message protocols are not explicit anywhere in the software, and are documented here only.

Although the code files for messaging are not divided into parts, conceptually, there are three sets of message protocols: those between controller bodies, those between parts of a controller, and those between the planners, Modeler, and Graphic Display. The first set is expected to differ greatly between stand-alone FBICS and integrated versions of FBICS. The second set will vary according to how a controller is divided, but will always consist of a wrapper for one or more module APIs. The third set is expected to be the same in all forms of FBICS.

The type of message protocol appropriate to two processes depends on the nature of the processes, so some information about the processes is provided here.

All the messaging protocols used in stand-alone FBICS involve two parties. One party sends commands (and does not repeat any command) and the other party returns one status message for each command message received. There is no queuing. The command sender never sends another command until status is received for the preceding command, so there is no chance of commands being overwritten and no chance of status being overwritten. As an extra check, each command message includes a sequence number, the status message responding to the command echoes that number, and the command sender always checks to make sure the status message has the same number as the previous command.

5.4.4.1 FBICS Stand-Alone Controller Message Protocols

The FBICS stand-alone controllers run cyclically, with no specified upper limit on cycle time. A stand-alone FBICS controller cycle lasts as long as it takes to handle one or both of a user command and a command from a superior controller; if there is neither, a cycle lasts a tenth of a second.

As described earlier, each FBICS stand-alone controller with a superior has a command-from-superior mailbox and a status-to-superior mailbox. Similarly, each FBICS stand-alone controller with a subordinate has a command-to subordinate mailbox and a status-from-subordinate mailbox. Thus, the top-level (Cell) controller (which has no superior) and the bottom-level (Task) controller (which has no subordinate) each have two mailboxes for control messages, while the in-between-level controller (Work) has four.

Stand-alone FBICS controllers read their command-from-superior mailbox each cycle. They write status-to-superior only when done with executing a command, which always occurs during

the same cycle in which the command was read. Stand-alone FBICS controllers write in their command-to-subordinate mailbox when ready to give a command, which may be many times or no times during one cycle. Immediately after giving a command, they read status-from-subordinate repeatedly at tenth-of-a-second intervals until the subordinate responds, or 1000 reads have been made. If 1000 reads are made with no status message being received, a total time of about 100 seconds, the user is asked whether to wait for (up to) another 1000 reads. If the user decides not to wait, an error is generated. The status reported in a status message may have only two values: OK (the command was executed successfully) or ERROR.

It would be an improvement to make the waiting-for-status time longer at higher hierarchical levels, but, in the current stand-alone implementation, waiting times are the same in all three controllers and the Modeler (which waits on the Graphics Display).

In the case of exit messages to the lower-level controllers, the protocol just described is changed. After sending an exit message, the sender process exits without trying to get a status message back.

5.4.4.2 FBICS Integrated Controller Message Protocols

Controllers used in integrated versions of FBICS are expected to be RCS controllers. This section describes some aspects of RCS controllers to suggest what message protocols may be used in integrated versions of FBICS.

Most RCS controllers run a cyclic process with either a fixed cycle time or a known upper limit on cycle time. On every cycle, most RCS controllers also read from a command-from-superior mailbox, write into a status-to-superior mailbox, write into zero to many command-to-subordinate mailboxes, and read from zero to many status-from-subordinate mailboxes. The status reported in a status message may have more than two values: IN_PROGRESS, OK, or ERROR, for example. Different RCS implementations use different sets of status values.

In one version of integrated FBICS, the stand-alone Task Controller is connected to an RCS-type controller which the Task Controller treats like the Fbics_Task2 process. The message protocol used in this case is that the Task Controller writes a command to the RCS controller only once, but then expects to read several status messages from the RCS controller reporting on the status of the previous command before getting a status report on the newly sent command.

5.4.4.3 FBICS Controller Internal Message Protocols

When a controller is divided into two or more processes, as the stand-alone Task Controller is and all controllers are expected to be in an advanced integrated FBICS, each message protocol used between modules in the two processes is expected to be nothing more than a wrapper for an API. The module sending a command message selects the message tailored for a specific API function call and stuffs the message with parameter values (if there are any). The message handlers for the two processes exchange the message. The receiving module carries out the API call. The return values (if any) from the function call are wrapped in a status message which is exchanged. If returned values are expected, the module that sent the command extracts the returned values from the status message.

The stand-alone Task Controller uses two mailboxes for internal communications. The mailboxes connect the Fbics_Task process with the Fbics_Task2 process.

5.4.4.4 Other FBICS Message Protocols.

The Modeler and the Graphic Display are cyclic processes with unbounded cycle time. The Modeler services one of the three planners on each cycle and goes on to the next planner in the next cycle. The Graphic Display serves only the Modeler.

The message protocols used between the planners and the Modeler are structurally similar to one another and conform to the description given above. Each time a planner sends a command message to the Modeler, it waits to get a status message responding to that before proceeding with other activities.

One of the message types which may be sent to the Modeler is the `MODEL_FUNCTION_MSG`. This differs from all other FBICS messages in being generic and serving many purposes. See Section 10.1.3 for details. One of the fields in a `MODEL_FUNCTION_MSG` is a function identifier that identifies one of 26 possible function types for the modeler to execute.

5.4.5 Message Types and Names

FBICS messages fall in two types: command messages and status messages.

The status messages are: `WORK_READY_MSG`, `TASK_READY_MSG`, `TASK2_READY_MSG`, `MODEL_READY_MSG`, and `DRAW_READY_MSG`. Cell has no status message. Except for the `MODEL_READY_MSG`, all status messages include only a sequence number and a status value (which may be only OK or ERROR, as noted earlier). The `MODEL_READY_MSG` has fields for additional information, as described in Section 10.1.4.

All command messages include a sequence number. Additional fields are noted in the message descriptions given in the following places:

Work Controller — Section 8.3,
Task Controller — Section 9.3,
Task2 process — Section 9.7,
Modeler — Section 10.1 and Section 10.2,
Graphic Display — Section 11.1.

There are no command messages to the Cell Controller because it sits at the top of the command hierarchy.

Except for in the Modeler, the receivers of commands do not have explicit knowledge of who sent command message (that is explicit only in the communications configuration file). The Modeler may get the same type of message from three different senders. The Modeler infers who the sender was from knowing which mailbox the message was in.

5.5 Data Interfaces

This section describes the interfaces between modules or processes that are implemented by transfer of files through the Data Repository. In most cases, the module getting the data is notified of the file name in a message. In some cases, the same file name is used repeatedly, and the name is not transferred in a message. The nature of the data is described in Section 12, and the descriptions are not repeated here. This section is arranged by interface.

Wherever a file name is passed to the Modeler in a `MODEL_FUNCTION_MSG`/function call in this section, it is passed from the first party in the subsection title to the Modeler, and the Modeler

reads the file when it executes the function.

Where this section refers to work in progress, this means the part_in and part_out in the case of machining (possibly) with inspection and part_now in the case of pure inspection.

Where a type of data is written at one time and read later by the same module, this has not been treated as a data interface and is not included in this section. The most important instances of this are the cell-level process plans (see Section 12.1.12) and work-level process plans (see Section 12.1.13).

5.5.1 Cell Planner to Work Planner

AP 224 STEP Part 21 files (see Section 12.1.4) representing work in progress and features are written by the Cell Planner during planning. These files are read by the Work Planner during planning when a `WORK_PLAN_MSG` or `WORK_PLAN_INSP_MSG` command is received and during execution when a `WORK_RUN1_MSG` is received. The file names are included in the command messages.

Setup STEP Part 21 files (see Section 12.1.16) are written by the Cell Planner during planning. These files are read by the Work Planner during planning when a `WORK_PLAN_MSG` or `WORK_PLAN_INSP_MSG` is received and during execution when a `WORK_RUN1_MSG` is received. The file names are included in the command messages.

5.5.2 Cell Planner to Modeler

The name of an AP 224 STEP Part 21 file for the design of the part_in (or part_out) is passed when the `MODEL_FUNCTION_MSG`/model_part_in (or model_part_out) function is called. The file itself may have existed before the Cell Planner started work, or the Cell Planner may have written the file.

5.5.3 Work Planner to Task Planner

When sending a `TASK_GEN_DMIS_MSG` or `TASK_GEN_NC_MSG` to the Task Planner, the Work Planner includes the name of a STEP Part 21 work-level executable operation file (see Section 12.1.15). This file is written by the Work Planner. The Task Planner reads the file while executing the command.

When sending a `TASK_OPEN_MSG` to the Task Planner, the Work Planner includes the name of a STEP Part 21 setup file (see Section 12.1.16). The setup file was written by the Cell Planner. The setup file identifies other files used by the Task Planner: work in progress and fixture. The Task Planner reads all of these files while executing the command.

5.5.4 Work Planner to Modeler

The name of an AP 224 file for a fixture is passed when the `MODEL_FUNCTION_MSG`/ model_fixture function is called. The file must be prepared outside of FBICS.

The name of an AP 224 STEP Part 21 features file is passed when the `MODEL_FUNCTION_MSG`/ model_part_features function is called.

The name of an AP 224 STEP Part 21 file for the design of the part_in (or part_out) is passed when the `MODEL_FUNCTION_MSG`/model_part_in (or model_part_out) function is called. The

file itself may have existed before the Cell Planner started work, or the Cell Planner may have written the file.

The name of an AP 224 STEP Part 21 file for the design of a feature is passed when the MODEL_FUNCTION_MSG/model_part_now_file function is called. This file is normally a file written by the Work Planner describing an executable operation, but it contains a feature description.

5.5.5 Task Planner to Modeler

The name of an AP 224 STEP Part 21 file for the design of the part_in is passed when the MODEL_FUNCTION_MSG/model_part_in function is called. The file itself may have existed before the Cell Planner started work, or the Cell Planner may have written the file.

The name of an AP 224 STEP Part 21 file for the design of a feature is passed when the MODEL_FUNCTION_MSG/model_part_now_file or show_volume_file function is called. This file is normally a file written by the Work Planner describing an executable operation, but it contains a feature description.

5.5.6 Modeler to Graphic Display

Six of the seven command messages (see Section 11.1) that may be sent from the Modeler to the Graphic Display each refer implicitly to a specific graphics file the Graphic Display should read. These files are all written by the Modeler and read by the Graphic Display. The format of graphics files is described in Section 12.4. The DRAW_FLUSH_MSG requires no file.

5.5.7 Fbics_Task to Fbics_Task2

The TASK2_EXEC_DMIS_MSG and TASK2_EXEC_NC_MSG command messages that may be sent from Fbics_Task to Fbics_Task2 name a DMIS (see Section 12.2) or RS274/NGC (see Section 12.3) file that Fbics_Task2 should execute. These files are written by Fbics_Task.

5.6 FBICS User Interfaces

FBICS has similar text-based interfaces for the three stand-alone controllers, described in Section 7.2 (Cell), Section 8.2 (Work), Section 9.2 (Task) and a graphical interface for the Graphic Display, described in Section 11.2.

The Modeler has no user interface but may ask for the user to decide what to do about a time-out.

5.6.1 Interface Modes

The user interfaces for the Work Controller, Task Controller, and Graphic Display each have a MANUAL mode and an AUTO (automatic) mode. In AUTO mode, the system processes command messages immediately. In MANUAL mode, the system waits for a user signal before processing a waiting command message. For the controllers, the user signal is pressing the <enter> key. For the Graphic Display, the user signal is putting the mouse cursor on a red bar and pressing a mouse button.

For the Work and Task controllers, if the controller is in MANUAL mode, it will accept user input, but if it is in AUTO mode, it will not accept user input.

The Cell Controller does not receive command messages, so it does not have a mode, but it

behaves as though it is always in MANUAL mode.

5.6.2 Time-outs

Four of the terminal windows (for the three controllers and the Modeler) may have time-out messages printed to them by their processes. This happens if a subordinate process has taken too long (about 100 seconds) to respond to an NML message. The time-out message, always the same, is: “Waited long for message. Wait more? (y/n) >”. When faced with this message, the user must enter “y” to continue with processing as planned. Entering anything else puts the waiting process into a state from which it may or may not be possible to proceed. If the user enters “y”, the waiting system continues its work with no harm done.

In the use that has been made of FBICS, time-outs caused by a subordinate dying non-obviously (what the time-outs were designed to detect) have been very rare. The two normally observed reasons for time-outs are, (1) the user has put the Graphics Display into MANUAL update mode and an update has been waiting for the user, or (2) some planning activity (most often at the work level) took a long time. If reason (1) obtains, all three controllers and the Modeler are likely to have timed out and will need to get the go-ahead “y”.

6 FBICS Verification

This section deals with error prevention and error detection. For FBICS to be effective, it must use every opportunity to avoid errors while it runs, or stop before it makes one. A single crash can put a machining center or coordinate measuring machine out of action for weeks and cost thousands of dollars to fix. Extreme efforts to avoid errors are, therefore, required. The source code for the three FBICs planners and the Modeler includes a combined total of over 500 error messages for conditions that are checked. The RS274/NGC and DMIS interpreters in the Fbics_Task2 process include several hundred more.

An extensive discussion of verification is given in [Kramer3]. Only topics specific to FBICS are covered here.

6.1 Verification Methods and Tools

The usual tools for ensuring software correctness were used in developing FBICS: helpful editor for writing code, compiler for flagging many errors, coverage checker, and code checker. Standard software correctness methods are also used: a set of test cases has been defined and regression testing is performed. The code itself is written with many cross-checks.

In addition to these generic things, several less generic methods and tools have been used for developing FBICS. The most important of these are a solid modeler (Parasolid), a set of tools for handling STEP models and data (STEP Tools software), and a graphics engine (HOOPS). Each of these tools includes a large set of checks on the data it processes.

The division of FBICS into several processes (most of which follows from using a hierarchical architecture) forces a great deal of verification that might not be done in a more centralized system. At each controller level, more highly aggregated data comes in (from the user or a superior controller) from one type of file, and less aggregated, more detailed data goes out in another type of file. In addition, data files are passed from the planners to the Modeler, and from the Modeler to the Graphic Display. In every case, the act of reading or writing the data includes many checks on the data, usually more during reading.

6.2 Shape Verification Using Modeler

As mentioned earlier, many types of shape are used in FBICS: part_out, part_in, part_now, features, access_volumes, etc. The Modeler is used to perform a number of checks on shapes. Each of the FBICS planners may do workpiece shape verification appropriate to the shapes with which it deals. The following checks have been implemented using the Modeler.

1. Check that the part_now at the end of each setup is the same shape as the part_out.
2. Check that the part_out fits entirely within the part_in in each setup.
3. Check that a feature to be inspected has no material in it.
4. Check that no feature which should be machined or inspected has its access irremediably blocked by another feature.

In addition to those checks, which are explicitly programmed, the Modeler automatically makes many checks while it runs, such as checking that the part is not mathematically non-manifold, or that the union of two bodies is a single body.

6.3 Other Part and Feature Checking

Many other checks of parts and features are made without the use of the Modeler. A large subset of these are to check that each machining feature is a realizable machining feature. This includes checking that each parameter is within a range appropriate for it, and that rules regarding the relations between parameters are not violated. Here is a sampling of checks of parts and features.

1. A check is made that the part_out and part_in are made of the same material.
2. A check is made that the native Z-axis of each feature is parallel to the setup Z-axis before machining or inspecting the feature.
3. A check is made that the depth of a pocket is positive.
4. A check is made that the bottom of a pocket is parallel to the top.
5. For a plus_minus_value tolerance on a feature parameter, a check is made that the upper bound is larger than the lower bound.

6.4 Machining Operation Verification

Machining operation descriptions are also intensively checked. Checks are made that the tool and the operation are compatible. For example:

1. The diameter of a drill making a hole should be the same as the diameter of the hole.
2. The diameter of an endmill making a pocket should not be greater than the width of the pocket.
3. The tool used to finish mill a pocket must be an endmill.

There is a verification function in the Task Planner software for each type of machining operation, but most of the functions are not complete. In particular,

1. Checks should be added that feeds and speeds are in a reasonable range.
2. Checks should be added that pass depths and stepovers are in a reasonable range.

6.5 Other Checks

When the tool catalog and tool inventory are read, checks are made on individual tools and on the correspondence between the inventory and the catalog. Every tool in the inventory, for example, must be of a type described in the catalog.

When options files are read, checks are made that all required options are present in the file and that all options values are valid.

Whenever an expression is read, it is checked for being a valid expression. Whenever an expression is evaluated, checks are made that function arguments are in a valid range and that any variables reference have been defined and assigned values.

7 Cell Controller

This section describes the Cell Controller in moderate detail.

Throughout this section, the following typesetting conventions are used.

1. User commands and their arguments are set in plain `courier font`.
2. API functions and their arguments are set in *`courier italic font`*.

Messages continue to be set in `courier font`. Since they all end in `_MSG`, they should not be confused with user commands or their arguments.

7.1 Cell Controller Architecture

The Cell Controller architecture is shown in Figure 11. This is a more detailed view than shown in Figure 1, but has the same connections. The user gives commands to the Cell Controller Body via a simple text-based user interface, in response to which the Cell Controller Body makes function calls to functions in the Cell Planner API and/or sends messages to the Work Controller.

To carry out the API function calls made to it, the Cell Planner (1) gets data from the Data Repository and/or sends data to it, (2) sends messages to the Modeler and gets answers back, (3) makes or interprets a plan, according to what the API function call says to do, (4) sends file names back to the Cell Controller Body.

The Cell Controller Body interacts directly with the user and with the Work Controller, but does not interact directly with the Modeler or the Data Repository.

The Cell Planner interacts directly with the Modeler and the Data Repository, but does not interact directly with the user or the Work Controller.

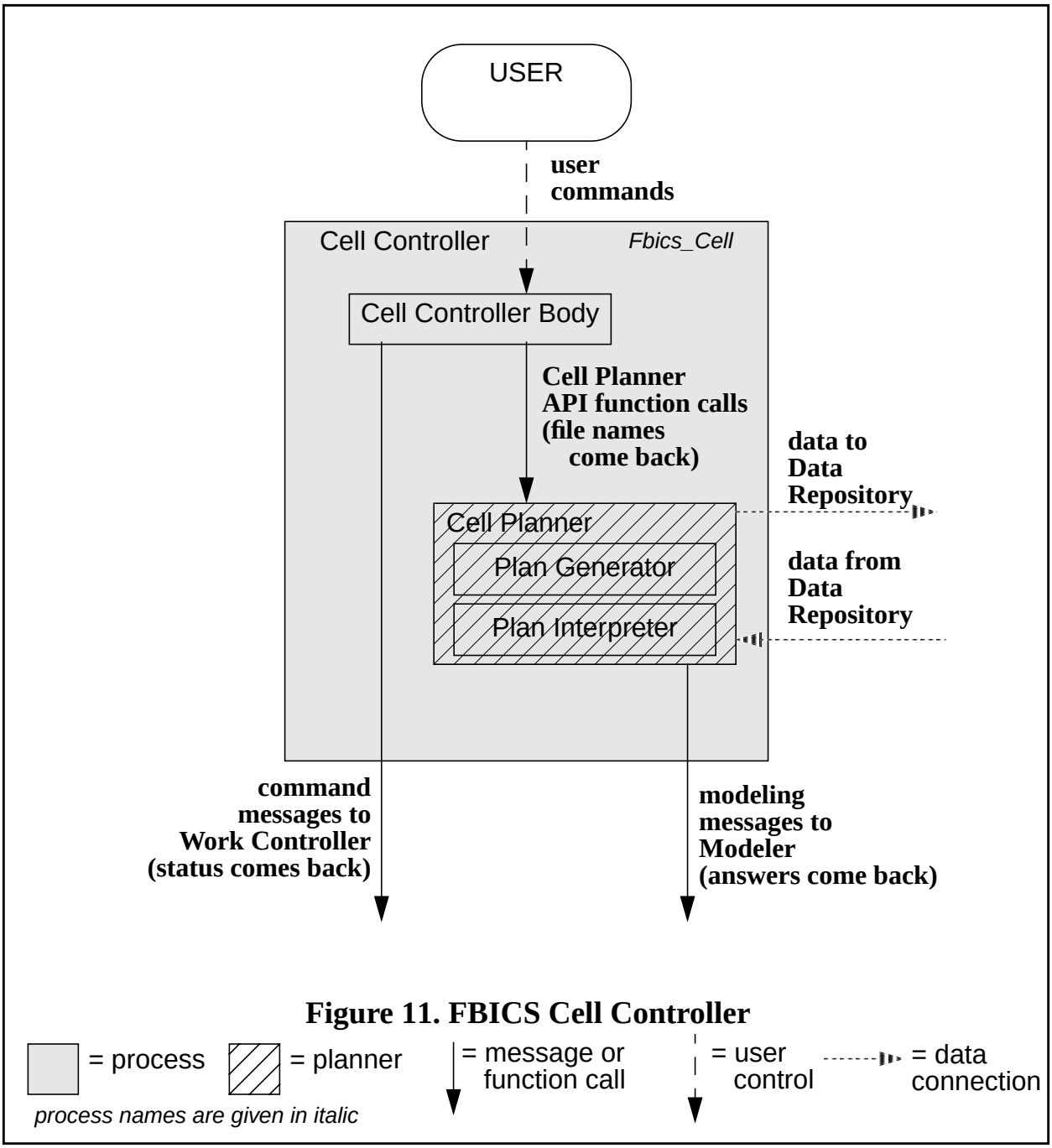


Figure 11. FBICS Cell Controller

7.2 User Interface to the Cell Controller

This section discusses the user interface to the Cell Controller, including a description of the terminal window, an overview of the user commands, and details of each user command.

7.2.1 Cell Controller Terminal Window

The Cell Controller terminal window is always enabled as a user interface. The user can give any of the user commands described in Section 7.2.2 when given the prompt `Cell =>` for user

input.

When the Cell Controller is started, the name of a command file may, optionally, be included as a command line argument. If the file name is used, the file must exist. Each line of the file should contain a command in the same form as if a user had typed it at the user interface. If the file name is used, each time the Cell Controller is ready to execute another command, it displays the next line from the file and asks the user whether it should be executed. If the user enters “y” the line is executed; otherwise the user is prompted to enter a command or just press the return key. If the user enters a command, it is executed, and then the Cell Controller goes on to the next line of the file. If the user just hits the return key, the Cell Controller goes on to the next line of the file. Once the end of the command file is reached, the `Cell =>` prompt appears after every user command.

Any improperly formatted command from the user or the file is detected and executing the command is not attempted. If an error occurs while executing a command, the Cell Controller is re-initialized.

7.2.2 User Commands to the Cell Controller

The user commands to the Cell Controller are displayed as shown in Figure 12 if the user types `help`. The figure shows the name, arguments, and meaning of each command.

When user commands are executed by the Cell Controller, functions from the Cell Planner API (described in Section 7.3) are called and command messages (described in Section 8.3) may be sent to the Work Controller. Whenever a command message is sent to the Work Controller, the Cell Controller waits for a response before proceeding. During execution of any user command, in addition to the operation of the command as described here, a number of checks are made. The checks are not described here.

In this section, some user commands are described as though they were function calls. They are not function calls, but the more meaty ones are carried out by making function calls to functions not explicitly described here. For example, the `init` command is carried out by the user_init function defined in source code for the Cell Controller Body.

```
help = print this list of commands
quit = quit cell controller
init() = initialize cell controller
plan_part_machine(part_out_name, part_in_exists (ON or OFF),
   part_in_name, plan_name, feature_name, setup_name, levels)
   = plan for machining with inspection
plan_part_inspect(part_name, plan_name, feature_name,
   setup_name, levels) = plan for inspection
run_part_plan1(plan_name) = run a stage1 cell-level plan
run_part_plan2(plan_name) = run a stage2 cell-level plan
exit() = exit cell controller
work_manual()
```

Figure 12. Cell Controller Help

7.2.3 Help

The user command `help` causes the Cell Controller to print the list of commands shown in Figure 12. This command has no arguments and is not followed by parentheses.

7.2.4 Quit

The user command `quit` causes the Cell Controller to quit immediately without tidying up the way `exit` does. The terminal window in which the controller was running disappears. No other processes exit. This command has no arguments and is not followed by parentheses.

7.2.5 Init

The user command `init()` initializes the Cell Controller. First, *`cellpl_init`* is called, causing the Cell Planner to initialize itself as described in Section 7.3.4. Then a `WORK_INIT_MSG` is sent to the Work Controller. The Cell Controller Body waits for and checks the reply from the Work Controller.

If the Cell Controller is already initialized when the `init()` command is given, the user is prompted to decide whether to re-initialize. If the user chooses to re-initialize, only the call to *`cellpl_init`* is made. No `WORK_INIT_MSG` is sent.

7.2.6 Plan_part_machine

The user command `plan_part_machine(part_out_name, part_in_exists (ON or OFF), part_in_name, plan_name, feature_name, setup_name, levels)` causes the Cell Controller to plan for machining an entire part (possibly) with inspection. The `levels` argument may be 1, 2, or 3.

Regardless of the value of `levels`, the first thing that happens is that a call is made to *`cellpl_plan_part1_machine`*. The first six command arguments are passed on as the six arguments to *`cellpl_plan_part1_machine`*. That function makes a stage-one plan and associated data, as described in Section 7.3.11. If `levels` is 1, that is all that happens.

If `levels` is 2 or 3, `plan_part_machine` traverses the stage-one plan just made, building a stage-two plan as the traversal progresses, as follows.

First, *`cellpl_open_plan1`* (see Section 7.3.8) is called to open the stage-one plan just made, and *`cellpl_open_plan2`* (see Section 7.3.9) is called to open a stage-two plan for writing. Then repeatedly:

1. *`cellpl_next_op1`* (see Section 7.3.6) is called to get the next operation. Until there are no more operations to do, the next operation is always of type RUN_SETUP. Getting the next operation includes traversing the stage-one plan when needed (see Section 4.2.6).
2. *`cellpl_make_op2`* (see Section 7.3.5) is called to make a corresponding one_operation and the one_operation is inserted in the stage-two plan. If `levels` is 2, the operation type is RUN_PLAN1 (since the Work Controller will be executing a stage-one plan when the operation is carried out). If `levels` is 3, the operation type is RUN_PLAN2 (since the Work Controller will be executing a stage-two plan when the operation is carried out).
3. The Work Controller is sent a WORK_PLAN_MSG (see Section 8.3.5) to cause it to plan to carry out the operation. If `levels` is 2, the `levels` field of the message is set to 1, so that each WORK_PLAN_MSG causes the Work Controller to make a stage-one plan for itself. If `levels` is 3, the `levels` field of the message is set to 2, so that each WORK_PLAN_MSG causes the Work Controller to make a stage-one plan for itself, to make a stage-two plan for itself, and to tell the Task Controller to write NC code and/or DMIS code files.

The repetition just described is stopped when *`cellpl_next_op1`* returns the operation type NONE, indicating that the stage-one plan has been completely traversed.

Each time around the loop above, the Cell Controller determines the shape of the part_in and part_out for the setup. If either shape was unknown when the setup file for the setup was written, the setup file is rewritten and a file (or two) describing the previously unknown shape(s) is written.

Finally, `plan_part_machine` calls *`cellpl_close_plan2`* (see Section 7.3.2) to write out the stage-two plan.

7.2.7 Plan_part_inspect

The user command `plan_part_inspect(part_name, plan_name, feature_name, setup_name, levels)` causes the Cell Controller to plan for inspecting an entire part. The `levels` argument may be 1, 2, or 3.

Regardless of the value of `levels`, the first thing that happens is that a call is made to *`cellpl_plan_part1_inspect`*. The first four command arguments are passed on as the first four arguments to *`cellpl_plan_part1_inspect`*. That function makes a stage-one plan and associated data, as described in Section 7.3.10. If `levels` is 1, that is all that happens.

What plan_part_inspect does if levels is 2 or 3 is identical to what plan_part_machine does in that case, as described immediately above, except that WORK_PLAN_INSP_MSGs (see Section 8.3.6) are sent rather than WORK_PLAN_MSGs.

7.2.8 Run_part_plan1

The user command run_part_plan1(plan_name) causes the Cell Controller to execute a stage-one cell-level plan. The plan_name argument is the base name of a cell-level plan. If the base name is "npl", for instance, the plan file named "npl_1cell.stp" is used. The "_1" portion of the name indicates it is a stage-one plan, the "cell" portion indicates it is a cell-level plan, and the ".stp" portion indicates it is a STEP Part 21 file. The plan which is executed may be either for machining (possibly) with inspection or for pure inspection.

To start executing a run_part_plan1 command, *cellpl_open_plan1* is called and opens the plan as described in Section 7.3.8.

Then, repeatedly:

1. *cellpl_next_op1* is called to get the next operation type and write any required data, as described in Section 7.3.6.
2. If the operation type is RUN_SETUP, a WORK_PLAN_MSG is sent to the Work Controller. If the operation type is RUN_SETUP_INSPECT, a WORK_PLAN_INSP_MSG is sent. This causes the Work Controller to make a stage-one plan for doing the work of the setup.
3. A WORK_RUN1_MSG is sent to the Work Controller, causing it to execute the plan it just made.

The repetition ends when the operation type is NONE, indicating the plan has been completely executed.

Finally, *cellpl_close_plan1* is called and closes the plan as described in Section 7.3.1.

7.2.9 Run_part_plan2

The user command run_part_plan2(plan_name) causes the Cell Controller to execute a stage-two cell-level plan. The plan which is executed may be either for machining (possibly) with inspection or for pure inspection. The plan_name argument is the base name of a cell-level plan. If the base name is "npl", for instance, the plan file named "npl_2cell.stp" is used. The "_2" portion of the name indicates it is a stage-two plan, the "cell" portion indicates it is a cell-level plan, and the ".stp" portion indicates it is a STEP Part 21 file.

To start executing a run_part_plan2 command, *cellpl_open_plan2* is called and opens the plan as described in Section 7.3.9.

Then, repeatedly:

1. *cellpl_next_op2* (see Section 7.3.7) is called to get the operation type and plan file name from the next one_operation on the list of one_operations in the plan.
2. If the operation type is RUN_PLAN1, a WORK_RUN1_MSG is sent to the Work Controller. If the operation type is RUN_PLAN2, a WORK_RUN2_MSG is sent. This causes the Work Controller to run the plan named in the message.

The repetition ends when the operation type is NONE, indicating the plan has been completely

executed.

Finally, *cellpl_close_plan2* is called and closes the plan as described in Section 7.3.2.

7.2.10 Exit

The user command `exit()` causes the Cell Controller to send a `WORK_EXIT_MSG` to the Work Controller. Then *cellpl_exit* is called and works as described in Section 7.3.3. The Cell Controller stops running (without waiting for a reply from the Work controller about the exit message). The terminal window in which the controller was running disappears. The exit messages cause a chain reaction of exit messages so that a total of six processes exit along with their terminal windows. Only the NML server process is left running.

7.2.11 Work_manual

The user command `work_manual()` causes the Cell Controller to send a `WORK_MANUAL_MSG` to the Work Controller telling it to switch into MANUAL mode (so that it executes commands typed at the keyboard). If the Work Controller is already in MANUAL mode, the message is sent but has no effect. If the user wants to switch the Work Controller from AUTO mode to MANUAL mode, this must be done by giving a `work_manual()` command to the Cell Controller. There is no user command to the Work Controller for that purpose, since the Work Controller is not listening to user commands while it is in AUTO mode.

7.3 Cell Planner API

This section describes the eleven Cell Planner API functions. These functions are called as described in Section 7.2 when the Cell Controller executes user interface commands.

In the interface function descriptions that follow, an argument is underlined if it is a pointer and what it points to is (or may be) set when the function executes. In other words, arguments that are returned values are underlined.

Several of the Cell Planner interface functions deal with names used by the Work Planner, but none of the functions tells the Work Planner to do anything.

Naming conventions used by the functions are described where they apply.

All these API functions return OK if they work without error. Return values in case of error are not discussed in this document but are given in the source code documentation. During execution of any API function, in addition to the operation of the function as described here, a number of checks are made. The checks are not described here.

7.3.1 Cellpl_close_plan1

The *cellpl_close_plan1()* function takes no arguments.

The general meaning of *cellpl_close_plan1* is: complete work on the currently open stage-one plan and get ready to do something else. If the plan is for machining, a check is made by the Modeler that the shape of the workpiece at the end of processing (the final part-now) is identical to the shape of the part as designed (the part-out). Data associated with the current plan are deleted from memory. The Cell Planner world model is re-initialized.

7.3.2 Cellpl_close_plan2

The *cellpl_close_plan2(char read_write)* function takes one argument:

1. *read_write* — a single character, either “r” if the file was read or “w” if the file was written.

The general meaning of *cellpl_close_plan2* is: complete work on the currently open stage-two plan and get ready to do something else. If a stage-two plan file was being written, it is saved. Data associated with the current plan are deleted from memory. The Cell Planner world model is re-initialized.

7.3.3 Cellpl_exit

The *cellpl_exit()* function takes no arguments.

The general meaning of *cellpl_exit* is: stop running. *Cellpl_exit* cleans the Cell Planner world model, frees memory allocated by STEP Tools functions, and sends a MODEL_EXIT_MSG to the Modeler. The function returns without waiting for a reply from the Modeler about the exit message.

7.3.4 Cellpl_init

The *cellpl_init()* function takes no arguments.

The general meaning of *cellpl_init* is: get ready to run. The Cell Planner world model is initialized. The Cell Planner reads the shop options file and transcribes relevant data into its world model as described in Section 7.4.

Other interface functions (except *cellpl_exit*) will return an error code if they are called before *cellpl_init*.

7.3.5 Cellpl_make_op2

The *cellpl_make_op2(int operation_class, char * plan_file_name)* function takes two arguments:

1. *operation_class* — an integer to put into the one_operation representing the type of operation,
2. *plan_file_name* — a string to put into the one_operation giving the name of the work-level plan file.

The general meaning of *cellpl_make_op2* is: make one one_operation and put it into the list of one_operations being built for a stage-two plan.

7.3.6 Cellpl_next_op1

The *cellpl_next_op1(int * operation class, char * setup file name, char * work plan name, char * suffix)* function takes four arguments:

1. *operation_class* — a pointer to an integer representing the type of operation; the value is set by the function to the integer code for one of: NONE, RUN_SETUP, or RUN_INSPECT_SETUP (indicating a setup for pure inspection). If the

operation_class is set to NONE, the function does nothing else.

2. *setup_file_name* — a string the planner writes into giving the name of a setup file.
3. *work_plan_name* — a string the planner writes into if the *operation_class* is RUN_SETUP or RUN_INSPECT_SETUP. The string gives the base name of the work-level process plan to be run when the operation is executed.
4. *suffix* — a string (either "temp" or "keep") to be added to the base of the *setup_file_name* to make a new name to use if a revised setup file is written.

The general meaning of *cellpl_next_op1* is: examine the stage-one process plan being executed, determine what the next operation to do should be, and write any files needed for the execution of the operation.

The function first gets the next plan node to execute by traversing the plan further (see Section 4.2.6). If the end of the plan has not been reached, the next node to execute will be a run_setup node. The function reads the setup file identified in the run_setup node. The plan name identified in the setup file is copied into the *work_plan_name* argument.

If the setup file identifies itself as being for pure inspection, the name of the setup file is copied into the *setup_file_name* argument, and the *operation_class* argument is set to RUN_INSPECT_SETUP.

If the setup file does not identify itself as being for pure inspection, it is for machining (possibly) with inspection. In this case, the name of the setup file is copied into the *setup_file_name* argument with the *suffix* added to the name, the *operation_class* argument is set to RUN_SETUP, the part_now is updated to show that all the features made by the setup have been made, part_in and part_out files are written, and the setup file is rewritten to put the names of the part_in and part_out files in place of "not_set".

7.3.7 Cellpl_next_op2

The *cellpl_next_op2(int * operation_class, char * plan_file_name)* function takes two arguments:

1. *operation_class* — a pointer to an integer representing the type of operation. The value is set by the function to the integer code for one of: NONE, RUN_PLAN1, or RUN_PLAN2.
2. *plan_file_name* — a string the planner writes into. If the type of operation is RUN_PLAN1 or RUN_PLAN2, the string gives a plan file name. The file will normally exist when this function is called, but that is not required by the function. If the type of operation is NONE, the *plan_file_name* is not written.

The general meaning of *cellpl_next_op2* is: examine the stage-two process plan being executed and determine what the next operation to do should be.

7.3.8 Cellpl_open_plan1

The *cellpl_open_plan1(char * plan_file_name)* function takes one argument:

1. *plan_file_name* — the name of an existing stage-one plan file to open. This should be a STEP Part 21 ALPS cell-level plan. It is an error if the file does not exist

when this function is called.

The general meaning of *cellpl_open_plan1* is: get ready to use an existing stage-one plan. The Cell Planner reads the process plan into a working form.

7.3.9 Cellpl_open_plan2

The *cellpl_open_plan2(char * plan_file_name, char read_write)* function takes two arguments:

1. *plan_file_name* — the name of a stage-two plan file to open.
2. *read_write* — "r" for reading, "w" for writing.

The general meaning of *cellpl_open_plan2* is: If *read_write* is "w", get ready to create a new stage-two plan; if *read_write* is "r" get ready to execute an existing stage-two plan. If *read_write* is "r", it is an error if the file does not exist when this function is called.

7.3.10 Cellpl_plan_part1_inspect

The *cellpl_plan_part1_inspect(char * part_file_name, char * plan_file_name, char * feature_file_name, char * setup_file_name, int * setups)* function takes five arguments:

1. *part_file_name* — the name of a STEP Part 21 AP 224 file for the part to inspect. The name will usually have a ".stp" suffix. It is an error if this file does not exist when this function is called.
2. *plan_file_name* — the base name of the STEP Part 21 cell-level ALPS plan to write. The actual cell-level plan name is the base name followed by the suffix "_1cell.stp".
3. *feature_file_name* — the base name of the STEP Part 21 AP 224 feature file(s) to write. One feature file is written for each setup. The actual name used for each feature file is the base name followed by the suffix "_N.stp", where N is 1 or 2 or 3, etc. and represents the setup number.
4. *setup_file_name* — the base name of the STEP Part 21 setup file(s) to write. The actual name used for each setup file is the base name followed by the suffix "_N_keep.stp" or "_N_temp.stp", where N is 1 or 2 or 3, etc. and represents the setup number and "keep" or "temp" indicates whether the setup file should be kept for later execution or is only temporary for immediate execution.
5. *setups* — a pointer to an integer giving the number of setups, set by the function.

The general meaning of *cellpl_plan_part1_inspect* is: make a stage-one cell-level process plan and associated data for inspecting a part. When the function executes, it writes a cell-level process plan and determines how many setups are required. For each setup, it writes a setup file and a feature file. Details follow.

First, the function sends a MODEL_ATTACH_MSG to the Modeler to get connected with the Modeler. After this, the function calls on the Modeler frequently using various types of MODEL_FUNCTION_MSG. These will not be described in detail here. See Section 10.2 for details regarding types of MODEL_FUNCTION_MSG.

Next, the function reads the file named by the *part_file_name* describing the design of the

part to be inspected, which is in terms of a base shape with AP 224 features. The Modeler is told to model the entire part and (as separate solids) every feature of the part.

For each feature of the part, a “feature_plus” is built. The feature_plus contains additional information about the feature, including a pointer to Modeler’s model of the feature. The access volume (see Section 4.3.6) of each feature is found by the Modeler and, using the access volumes, it is checked that it is possible to inspect the part by approaching each feature from the direction of the feature’s native Z-axis.

The set of all part features is divided into subsets called direction-sets, all of whose native z-axes point in the same direction (since these are all inspectable from the same direction). The features in each direction set are examined to see which should be inspected, and that subset of the direction set is designated as the setup-set. If the setup-set is empty, no setup is needed for that setup-set.

If there is no non-empty setup-set, planning is finished; no files are written.

Otherwise, a trivial partial ordering for inspecting non-empty setup-sets is captured in a cell-level stage-one ALPS process plan; the setups may be run in any order. The plan nodes that are partially ordered are all run_setup nodes, one for each setup-set.

For each run_setup plan node, a setup file is made and its name is recorded in the run_setup node. A file describing the features to be inspected in the setup is made. The name of this file is recorded in the setup file along with (1) the name of the design, (2) the base name of a work-level plan (not yet in existence) for inspecting the features in the setup-set, (3) the name of the fixture file to use for the setup and (4) the direction of the setup-set.

The string given by the *`plan_file_name`* argument is used in the setup files this writes as the base name for work-level process plans. The full work-level plan name used in the setup files is the plan file name followed by the suffix “_N_1work.stp” or “_N_2work.stp”, where the “_1” or “_2” means to use a stage-one or stage-two work-level plan and N is 1 or 2 or 3, etc. and represents the setup number.

When its work is done, the function sends a `MODEL_DETACH_MSG` to the Modeler to disconnect from the Modeler and re-initializes the Cell Planner world model.

7.3.11 Cellpl_plan_part1_machine

The *`cellpl_plan_part1_machine(char * part_out_file_name, ON_OFF part_in_exists, char * part_in_file_name, char * plan_file_name, char * feature_file_name, char * setup_file_name)`* function takes six arguments:

1. *`part_out_file_name`* — the name of a STEP Part 21 AP 224 file for the part to make. The name will usually have a “.stp” suffix. It is an error if this file does not exist when this function is called.
2. *`part_in_exists`* — an integer (0=no, 1=yes) indicating whether the part_in file exists. It is an error for this argument to be set to 1 if the file does not exist. If the argument is set to 0 and the file already exists, the file will be overwritten.

3. *part_in_file_name* — the name of a STEP Part 21 AP 224 file for the part to start with. The name will usually have a ".stp" suffix. If the file exists, it is taken to be a file describing the workpiece as it is before processing starts. This may be a featureless base shape, or it may be a base shape with some features already made. If the file does not exist, the function creates a base shape to use for the part_in and writes a file of the given name.
4. *plan_file_name* — the base name of the STEP Part 21 ALPS cell-level plan to write and the base name for work-level plans used in the setup file(s) written by this function. The actual cell-level plan name is the base name followed by the suffix "_cell.stp". The actual work-level plan name used in the setup files is the base name followed by the suffix "_N.stp", where N is 1 or 2 or 3, etc. and represents the setup number.
5. *feature_file_name* — the base name of the STEP Part 21 AP 224 feature file(s) to write. One feature file is written for each setup. The actual name used for each feature file is the base name followed by the suffix "_N.stp", where N is 1 or 2 or 3, etc. and represents the setup number.
6. *setup_file_name* — the base name of the STEP Part 21 setup file(s) to write. The actual name used for each setup file is the base name followed by the suffix _N.stp, where N is 1 or 2 or 3, etc. and represents the setup number.

The general meaning of *cellpl_plan_part1_machine* is: make a stage-one cell-level process plan and associated data for machining a part, possibly interleaved with periodic inspection of the part (depending on the design and the shop options settings). The function determines how many setups are required. For each setup, it selects a fixture and writes a setup file and a features file. Details follow.

First, the function sends a MODEL_ATTACH_MSG to the Modeler to get connected with the Modeler. After this, the function calls on the Modeler frequently using various types of MODEL_FUNCTION_MSG. These will not be described in detail here. See Section 10.2 for details regarding types of MODEL_FUNCTION_MSG.

Next, the function reads the file named by the *part_out_file_name* describing the design of the part to be made, which is in terms of a base shape with AP 224 features. The Modeler is told to model the entire part and (as separate solids) every feature of the part.

If *part_in_exists* is 1, indicating a description of the shape of the part_in is also provided, the function reads the file named by the *part_in_file_name*. If *part_in_exists* is zero, the function defines a part_in to have the same shape as the base shape of the part_out but with no features; then it writes a STEP AP 224 file named by the *part_in_file_name* containing the part_in description. The Modeler is told to make a model of the part_in. A copy of the model is made to represent the part_now. The part_now model is constantly updated as the function proceeds, while the part_in model is unchanging.

For each feature of the part_out, a "feature_plus" is built with a lot of help from the Modeler. The feature_plus contains additional information about the feature, including but not limited to:

1. a pointer to the Modeler's model of the feature.
2. a pointer to the access volume of the feature (see Section 4.3.6).
3. a pointer to the block bodies of the feature (see Section 4.3.7).

4. the physical block_bys of the feature. This is a list of pointers to other feature_pluses whose parent features physically block access to the feature.
5. the logical block_bys of the feature. This is a list of pointers to other feature_pluses whose parent features block access, logically or physically, to any of the physical block_bys of the feature.
6. the block_tos of the feature. This is a list of pointers to other feature_pluses whose parent features are blocked by the feature.

An initial list of features to be made is built by making a copy of the features on the part_out. Some of the features may already exist on the part_in. Each feature is tested by doing a boolean intersection of a boundary representation of the feature with the boundary representation of the part_in. If the intersection is empty, the feature requires no machining and is deleted from the list of features to be made. By using the boundary representations of the feature and the part_in, it is not necessary to match features on the part_in with features on the part_out.

The function checks that the part_out is contained in the part_in.

The set of features to be made is divided into subsets called direction_sets, all of whose native z-axes point in the same direction (since these are all machinable from the same direction). Each direction_set is separated into two subsets: makeables and unmakeables. The makeables are initially those features whose block_bys of both types are empty. The unmakeables are the rest of the features. Either makeables or unmakeables may be empty. After the initial assignment, any feature in the unmakeables, all of whose block_bys are in the makeables, is transferred from the unmakeables to the makeables (since it will be makeable in the same setup). The transfer procedure is repeated over and over until no feature is transferred.

Setup-sets (sets of features that can be made in a single setup) and a partial ordering for making the setup-sets are then constructed concurrently by the following iterative procedure.

1. If one or more direction_sets has no unmakeables, those direction sets are used as a group of setup-sets representing setups which may be made in any order. Otherwise, if one or more direction sets has non-empty makeables, those makeables are used as a group of setup-sets representing setups which may be made in any order. The features in the setup-sets so identified are henceforth considered to have been made.
2. The part_now is updated and the block bodies, access volume, block_bys, and block_tos of every feature are recomputed. Any direction_set with no remaining features is deleted from the direction sets. The makeables and unmakeables of each direction set are recomputed.

The two steps above are repeated until all features have been considered made. This produces a total ordering of groups of setup-sets in which the setup-sets of each group may be made in any order. This ordering is captured in a cell-level stage-one ALPS process plan. The task_nodes of the plan are all run_setup nodes, one for each setup-set. The ordering is embodied in the ALPS plan using the following rules:

1. If a group of setup-sets has two or more setup-sets, (i) make a parameterized_split_node (with m_number zero and serial timing) and have its successors be the run_setup nodes for the setup-sets, and (ii) make a path_join_node

and have it be the successor of the run_setup node for each of the setup-sets.

2. If a group of setup-sets has two or more setup-sets, the successor of the join_node for the group is: (i) if there are no more groups, an end_plan_node, (ii) if the next group has one setup-set, the run_setup node for the setup-set, or (iii) if the next group has two or more setup-sets, the parameterized_split_node for the group.
3. If a group of setup-sets has one setup-set, the successor of the run_setup node for the setup-set is: (i) if there are no more groups, an end_plan_node, (ii) if the next group has one setup-set, the run_setup node for the setup-set, or (iii) if the next group has two or more setup-sets, the split_node for the group.

For each run_setup plan node, a setup file is made and its name is recorded in the run_setup node. A file describing the features to be made in the setup is also made. The following items are recorded in the setup file (1) the base name of a work-level plan (not yet in existence) for making the features in the setup-set, (2) the name of the fixture file to use for the setup, (3) the direction of the setup-set, (4) the name of the features file, and (5) “not_set” for the names of the part_in file and part_out file for the setup[1].

Finally, the function checks that the shape of the part_now (which has been updated periodically) is the same as the shape of the part_out.

When its work is done, the function sends a `MODEL_DETACH_MSG` to the Modeler to disconnect from the Modeler and re-initializes the Cell Planner world model.

7.4 Cell Planner Options

The options used by the Cell Planner are a subset of the shop options (see Section 12.1.6). These are the inspecting_action, inspecting_decision, inspecting_interval, inspecting_level, milling_tolerance_default, and milling_tolerance_tightest.

1. The shape of the part_in and part_out cannot always be determined because, in general, setups may be made in different orders. It is possible to determine these shapes in many cases (whenever the setups are totally ordered, for example), but it is simpler and saves planning time not to get into analyzing the ordering situation. Of course, time is lost at execution time by waiting until then to determine shapes that could have been determined at planning time.

8 Work Controller

This section describes the Work Controller in moderate detail.

Throughout this section, the following typesetting conventions are used.

1. User commands and their arguments are set in plain `courier font`.
2. API functions and their arguments are set in *`courier italic font`*.

Messages continue to be set in `courier font`, but since they all end in `_MSG`, they should not be confused with user command items.

8.1 Work Controller Architecture

The Work Controller architecture is shown in Figure 13. This is a more detailed view than shown in Figure 1, but has the same connections. The Work Controller may get commands either from the user or from the Cell Controller, or both. The user gives commands via a simple text-based user interface. The Cell Controller gives commands by sending messages. Commands from both sources are handled by the Work Controller Body. In response to these commands, the Work Controller Body makes function calls to functions in the Work Planner API and/or sends messages to the Task Controller.

To carry out the API function calls made to it, the Work Planner (1) gets data from the Data Repository and/or sends data to it, (2) sends messages to the Modeler and gets answers back, (3) makes or interprets a plan, according to what the API function call says to do, (4) sends file names back to the Work Controller Body.

The Work Controller Body interacts directly with the Cell Controller, the Task Controller, and the user, but does not interact directly with the Modeler or the Data Repository.

The Work Planner interacts directly with the Modeler and the Data Repository, but does not interact directly with the Cell Controller, the Task Controller, or the user.

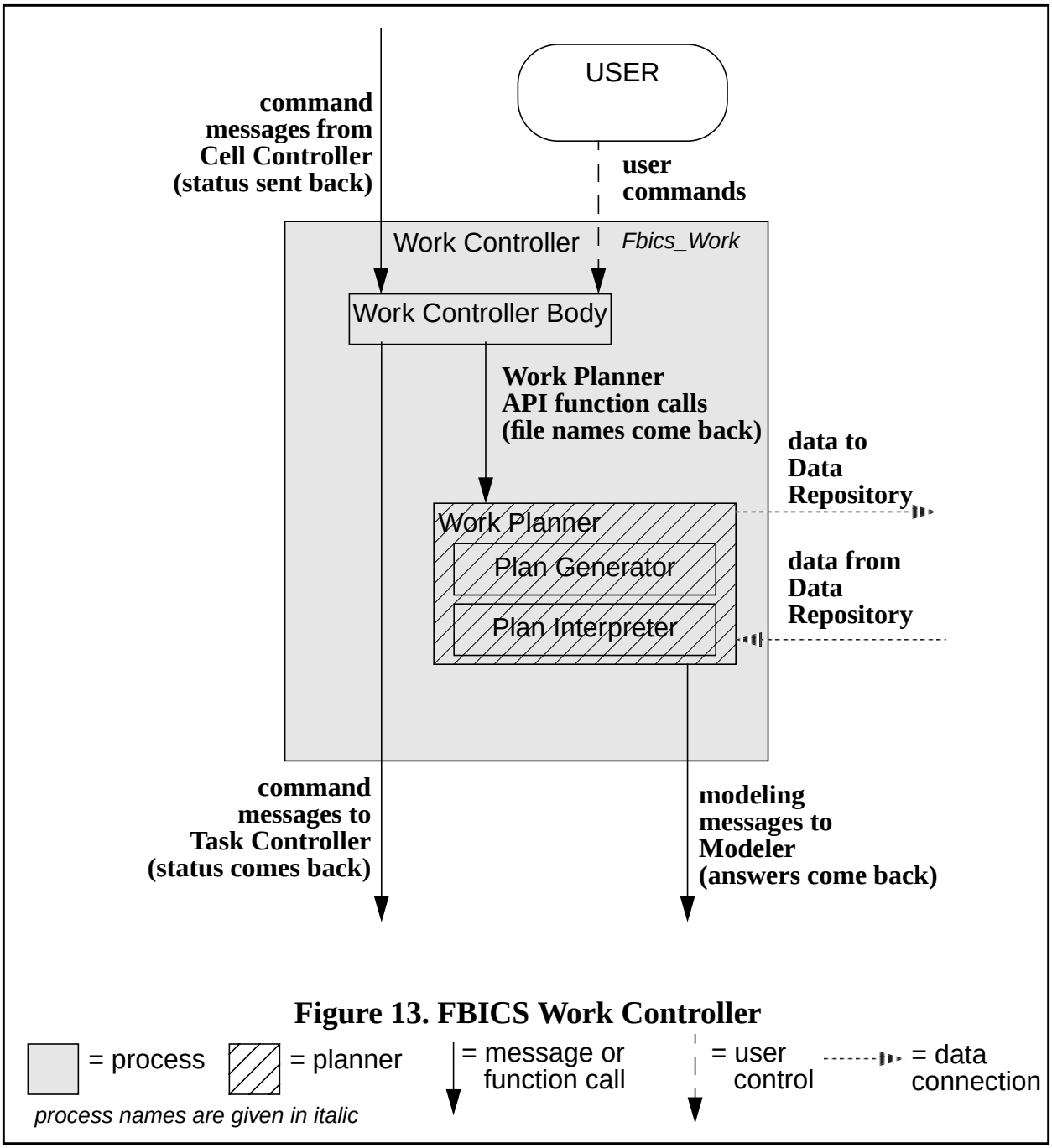


Figure 13. FBICS Work Controller

8.2 User Interface to the Work Controller

This section discusses the user interface to the Work Controller, including a description of the terminal window, an overview of the user commands, and details of each user command.

8.2.1 Work Controller Terminal Window

The Work Controller terminal window may be enabled or disabled as a user interface. The user can give any of the user commands described in Section 8.2.2 when given the prompt `Work =>`

for user input. This happens only when user input is enabled. The user can still type in commands when user input is disabled, but the Work Controller does not read them.

When the Work Controller is started, MANUAL or AUTO may, optionally, be included as a command line argument. If AUTO is included, the Work Controller starts up in AUTO mode with user input disabled. Otherwise, the Work Controller starts up in MANUAL mode and user input is enabled. The “fbics” script used to start stand-alone FBICS starts the Work Controller in AUTO mode.

The Work Controller is always ready to respond to command messages from the Cell Controller. When the Work Controller is in MANUAL mode, however, after a command message is received from the Cell Controller, the Work Controller waits until the user presses the <enter> key before executing the command message. In AUTO mode, the Work Controller executes command messages immediately.

8.2.2 User Commands to the Work Controller

The user commands to the Work Controller are displayed as shown in Figure 14 if the user types `help`. The figure shows the name, arguments, and meaning of each command.

When user commands are executed by the Work Controller, functions from the Work Planner API (described in Section 8.4) are called and command messages (described in Section 9.3) may be sent to the Task Controller. Whenever a command message is sent to the Task Controller, the Work Controller waits for a response before proceeding. During execution of any user command, in addition to the operation of the command as described here, a number of checks are made. The checks are not described here.

In this section, some user commands are described as though they were function calls. They are not function calls, but the more meaty ones are carried out by making function calls to functions not explicitly described here. For example, the `init` command is carried out by the user_init function defined in source code for the Work Controller Body.

```
help = print this list of commands
quit = quit the workstation controller
auto = run automatic (keyboard dead) until reset by CELL
init() = initialize workstation controller
plan_setup(setup_file_name, levels) = plan setup
plan_inspect_setup(setup_file_name, levels) = plan inspect setup
run_setup_plan1(plan_name, base_name) = run stage-one plan
run_setup_plan2(plan_name) = run stage-two plan
exit() = exit workstation controller
task_manual() = return task controller to manual mode
```

Figure 14. Work Controller Help

8.2.3 Help

The user command `help` causes the Work Controller to print the list of commands shown in Figure 14. This command has no arguments and is not followed by parentheses.

8.2.4 Quit

The user command `quit` causes the Work Controller to quit immediately without tidying up the way `exit` does. The terminal window in which the controller was running disappears. No other processes exit. This command has no arguments and is not followed by parentheses.

8.2.5 Auto

The user command `auto` causes the Work Controller to switch into AUTO mode. This command has no arguments and is not followed by parentheses. In AUTO mode, the Work Controller does not pay attention to the keyboard and takes action on future messages as soon as they are received.

Since user input is not processed in AUTO mode, there is no Work Controller user command to switch into MANUAL mode. To switch the Work Controller from AUTO mode to MANUAL mode, the `work_manual()` command must be given to the Cell Controller (see Section 7.2.11).

8.2.6 Init

The user command `init()` initializes the Work Controller. First, *`workpl_init`* is called, causing the Work Planner to initialize itself as described in Section 7.3.4. Then a `TASK_INIT_MSG` is sent to the Task Controller. The Work Controller Body waits for and checks the reply from the Task Controller.

If the Work Controller is already initialized when the `init()` command is given, the user is

prompted to decide whether to re-initialize. If the user chooses to re-initialize, only the call to *workpl_init* is made. No TASK_INIT_MSG is sent.

8.2.7 Plan_setup

The user command plan_setup(setup_file_name, levels) causes the Work Controller to plan for machining the features of one setup (possibly) with inspection. The setup to plan for is as described by the setup file having the given setup_file_name. Setup files are described in Section 12.1.16. The levels argument may be 1 or 2.

Regardless of the value of levels, the first thing that happens is that a call is made to the *workpl_plan_setup1* interface function. The setup_file_name command argument is passed as the argument of the same name to *workpl_plan_setup1*. A pointer to a buffer is passed to *workpl_plan_setup1* as its *plan_name* argument, and that function writes the full plan name into the buffer. *Workpl_plan_setup1* makes a work-level stage-one plan and associated data, as described in Section 8.4.10. If levels is 1, that is all that happens.

If levels is 2, plan_setup then calls *workpl_plan_setup2* (see Section 8.4.11) to build a stage-two plan and calls *workpl_open_plan2* (see Section 8.4.8) to open the plan for reading.

Then repeatedly:

1. *workpl_next_op2* (see Section 8.4.6) is called to get the next stage-two operation, and
2. the Task Controller is sent an appropriate message to cause it to plan to carry out the operation (in response to which the Task Controller will write files of NC code and/or DMIS code). These files will be executed when the Work Controller executes its stage-two plan.

The first operation selected by *workpl_next_op2* should be OPEN_SETUP, for which a TASK_OPEN_MSG will be sent. The last operation selected should be CLOSE_SETUP, for which a TASK_CLOSE_MSG will be sent. All the operations in between should be either MACHINING or INSPECTION for which a TASK_GEN_NC_MSG or a TASK_GEN_DMIS_MSG will be sent.

Finally, *workpl_close_plan2* (see Section 8.4.2) is called to close the stage-two plan.

8.2.8 Plan_inspect_setup

The user command plan_inspect_setup(setup_file_name, levels) causes the Work Controller to plan for inspecting the features of one setup. The setup to plan for is as described by the setup file having the given setup_file_name. Setup files are described in Section 12.1.16. The levels argument may be 1 or 2.

Regardless of the value of levels, the first thing that happens is that a call is made to the *workpl_plan_inspect_setup1* interface function. The setup_file_name command argument is passed as the argument of the same name to *workpl_plan_inspect_setup1*. A pointer to a buffer is passed to *workpl_plan_inspect_setup1* as its *plan_name* argument, and that function writes the full plan name into the buffer. *Workpl_plan_inspect_setup1* makes a work-level stage-one plan and associated data, as described in Section 8.4.9. If levels is 1, nothing else is done.

If `levels` is 2, `plan_setup` then calls *`workpl_plan_setup2`* (see Section 8.4.11) to build a stage-two plan and calls *`workpl_open_plan2`* (see Section 8.4.8) to open the plan for reading.

Then repeatedly:

1. *`workpl_next_op2`* (see Section 8.4.6) is called to get the next stage-two operation, and
2. the Task Controller is sent an appropriate message to cause it to plan to carry out the operation (in response to which the Task Controller will write files of DMIS code). These files will be executed when the Work Controller executes its stage-two plan.

The first operation selected by *`workpl_next_op2`* should be OPEN_SETUP, for which a `TASK_OPEN_MSG` will be sent. The last operation selected should be CLOSE_SETUP, for which a `TASK_CLOSE_MSG` will be sent. Each operation in between should be INSPECTION, for which a `TASK_GEN_DMIS_MSG` will be sent.

Finally, *`workpl_close_plan2`* (see Section 8.4.2) is called to close the stage-two plan.

8.2.9 Run_setup_plan1

The user command `run_setup_plan1(plan_name, base_name)` causes the Work Controller to execute the stage-one work-level process plan of the given `plan_name`. Executing the plan does the work of one setup. While it is executing, the Work Controller tells the Task Controller to write and then execute files of machining and/or inspection code. The given `base_name` is used as the base of the names for those files. The plan which is executed may be either for machining (possibly) with inspection or for pure inspection.

To start executing a `run_setup_plan1` command, *`workpl_open_plan1`* is called and opens the plan as described in Section 8.4.7. In addition, a `TASK_OPEN_MSG` (see Section 9.3.10) is sent to the Task Controller, causing it to open the setup.

Then, repeatedly:

1. *`workpl_next_op1`* is called to get the next operation type and write an executable operation file, as described in Section 8.4.5.
2. If the operation type is INSPECTION, a `TASK_GEN_DMIS_MSG` is sent to the Task Controller, followed by a `TASK_EXEC_DMIS_MSG`. These messages cause the Task Controller to generate then execute a section of DMIS code to perform the operation described in the executable operation file.
3. If the operation type is MACHINING, a `TASK_GEN_NC_MSG` is sent to the Task Controller, followed by a `TASK_EXEC_NC_MSG`. These messages cause the Task Controller to generate then execute a section of RS274/NGC NC code to perform the operation described in the executable operation file.

The repetition ends when the operation type is NONE, indicating the plan has been completely executed.

Finally, *`workpl_close_plan1`* is called and closes the plan as described in Section 8.4.1. Also, a `TASK_CLOSE_MSG` (see Section 9.3.1) is sent to the Task Controller, causing it to close the setup.

8.2.10 Run_setup_plan2

The user command `run_setup_plan2(plan_name)` causes the Work Controller to execute the stage-two work-level process plan of the given plan_name. Executing the plan does the work of one setup. The plan includes the names of files of machining and/or inspection code. Those files must already exist when this command is given. They are executed as the plan is executed.

To execute any stage-two work-level process plan, first *`workpl_open_plan2`* (see Section 8.4.8) is called, then *`workpl_next_op2`* (see Section 8.4.6) is called repeatedly until it returns operation type NONE.

The first operation selected by *`workpl_next_op2`* should be an OPEN_SETUP operation. The Task Controller does not need the information in the setup file, because all it is going to do is execute existing code, so a TASK_OPEN_MSG (see Section 9.3.10) is sent to the Task Controller with a setup name that is the empty string.

The rest of the operations (except the last) selected by *`workpl_next_op2`* will be MACHINING or INSPECTION operations for which the code files have already been written. For each MACHINING operation, a TASK_EXEC_NC_MSG is sent to the Task Controller. For each INSPECTION operation, a TASK_EXEC_DMIS_MSG is sent to the Task Controller.

The last operation selected by *`workpl_next_op2`* before it returns NONE should be a CLOSE_SETUP operation. *`Workpl_close_plan2`* (see Section 8.4.2) is called and a TASK_CLOSE_MSG is sent to the Task Controller.

8.2.11 Exit

The user command `exit()` causes the Work Controller to call *`workpl_exit`* (which works as described in Section 8.4.3) and to send a TASK_EXIT_MSG to the Task Controller. Then the Work Controller stops running (without waiting for a reply from the Task controller about the exit message). The exit messages cause a chain reaction of exit messages so that the Fbics_Work, Fbics_Task, and Fbics_Task2 processes exit and their terminal windows disappear.

8.2.12 Task_manual

The user command `task_manual()` causes the Work Controller to send a TASK_MANUAL_MSG to the Task Controller telling it to switch into MANUAL mode (so that it executes commands typed at the keyboard). If the Task Controller is already in MANUAL mode, the message is sent but has no effect. If the user wants to switch the Task Controller from AUTO mode to MANUAL mode, this must be done by giving a `task_manual()` command to the Work Controller. There is no command to the Task Controller for that purpose, since the Task Controller is not listening to user commands while it is in AUTO mode.

8.3 Command Messages to the Work Controller

Eight command messages to be sent to the Work Controller (by the Cell Controller) are defined, as follows. All messages have an integer-valued `sequence_number` field. This field is a sequence number as described in Section 5.4.4.

If the Work Controller is in AUTO mode, it carries out each message as soon as the message is received. If the Work Controller is in MANUAL mode, it waits for the user to press the <enter>

key before carrying out the message.

8.3.1 WORK_EXIT_MSG

The `WORK_EXIT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `WORK_EXIT_MSG` is identical to the effect of carrying out the user command `exit`, as described in Section 8.2.11.

8.3.2 WORK_IDLE_MSG

The `WORK_IDLE_MSG [int sequence_number]` has no fields other than the `sequence_number`.

This command is not currently used. Carrying out a `WORK_IDLE_MSG` is intended to have no effects.

8.3.3 WORK_INIT_MSG

The `WORK_INIT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `WORK_INIT_MSG` is identical to the effect of carrying out the user command `init`, as described in Section 8.2.6.

8.3.4 WORK_MANUAL_MSG

The `WORK_MANUAL_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `WORK_MANUAL_MSG` is to put the Work Controller into MANUAL mode. In MANUAL mode, the Work Controller pays attention to the keyboard and does not take action on future messages received until the user presses the <enter> key. When the Work Controller is in MANUAL mode, the user may put it into AUTO mode by giving the user command `auto` (see Section 8.2.5).

8.3.5 WORK_PLAN_MSG

The `WORK_PLAN_MSG [int sequence_number, char setup_file_name[80], int levels]` has two fields in addition to the `sequence_number`.

1. `setup_file_name` — a string giving the name of the setup file to use.
2. `levels` — an integer (1 or 2) giving the number of levels to plan for.

The effect of carrying out a `WORK_PLAN_MSG` is identical to the effect of carrying out the user command `plan_setup`, as described in Section 8.2.7 (using command arguments as given by the fields of the message).

8.3.6 WORK_PLAN_INSP_MSG

The `WORK_PLAN_INSP_MSG [int sequence_number, char setup_file_name[80], int levels]` has two fields in addition to the `sequence_number`.

1. `setup_file_name` — a string giving the name of the setup file to use.
2. `levels` — an integer (1 or 2) giving the number of levels to plan for.

The effect of carrying out a `WORK_PLAN_INSPECT_MSG` is identical to the effect of carrying out the user command `plan_inspect_setup`, as described in Section 8.2.8 (using command arguments as given by the fields of the message).

8.3.7 WORK_RUN1_MSG

The `WORK_RUN1_MSG [int sequence_number, char plan_file_name[80], char base_file_name[80]]` has two fields in addition to the `sequence_number`.

1. `plan_file_name` — the name of the stage-one work-level process plan to execute.
2. `base_file_name` — the base name to use for files written when this message is carried out.

The effect of carrying out a `WORK_RUN1_MSG` is identical to the effect of carrying out the user command `run_setup_plan1`, as described in Section 8.2.9 (using command arguments as given by the fields of the message).

8.3.8 WORK_RUN2_MSG

The `WORK_RUN2_MSG [int sequence_number, char plan_file_name[80]]` has one field in addition to the `sequence_number`.

1. `plan_file_name` — the name of the stage-two work-level process plan to execute.

The effect of carrying out a `WORK_RUN2_MSG` is identical to the effect of carrying out the user command `run_setup_plan2`, as described in Section 8.2.10 (using command arguments as given by the fields of the message).

8.4 Work Planner API

The eleven Work Planner API functions described in this section are called when the Work Controller executes the user interface commands described in Section 8.2 or the command messages to the Work Controller described in Section 8.3.

In the interface function descriptions that follow, an argument is underlined if it is a pointer and what it points to is set when the function executes. In other words, arguments that are returned values are underlined.

8.4.1 Workpl_close_plan1

The *`workpl_close_plan1()`* function takes no arguments.

The general meaning of *`workpl_close_plan1`* is: complete work on the currently open stage-one plan and get ready to do something else. Data associated with the current plan are deleted from memory. The Work Planner detaches from the Modeler. The Work Planner world model is re-initialized.

8.4.2 Workpl_close_plan2

The *`workpl_close_plan2()`* function takes no arguments.

The general meaning of *`workpl_close_plan2`* is: complete work on the currently open

stage-two plan and get ready to do something else. Data associated with the current plan are deleted from memory. The Work Planner world model is re-initialized.

8.4.3 Workpl_exit

The *workpl_exit()* function takes no arguments.

The general meaning of *workpl_exit* is: get ready to stop running. Data associated with the current plan are deleted from memory.

8.4.4 Workpl_init

The *workpl_init()* function takes no arguments.

The general meaning of *workpl_init* is: get ready to run. The Work Planner world model is initialized. The Work Planner reads the shop options and work options files and transcribes relevant data into its world model as described in Section 8.5. The Work Planner reads the tool catalog, the tool inventory, and the tool_usage_rules (all from STEP Part 21 files) and puts pointers to them into its world model.

Other interface functions (except *workpl_exit*) will return ERROR if they are called before *workpl_init*.

8.4.5 Workpl_next_op1

The *workpl_next_op1(int * operation_class, char * op_file_name)* function takes two arguments:

1. *operation_class* — a pointer to an integer representing the type of operation; the value is set by the function to the integer code for one of: NONE, MACHINING, or INSPECTION.
2. *op_file_name* — a string containing the name of the STEP Part 21 executable operation file the planner should write.

The general meaning of *workpl_next_op1* is: examine the stage-one process plan being executed and determine what the next operation to do should be. The function writes a file describing the operation, puts the file name into the *op_file_name* string, and puts a code for the type of the operation into the *operation_class*. A single step from the process plan may give rise to several operations.

To support this function, the Work Planner world model maintains a list of executable operations it has already planned to do and keeps track of where it is in the list. If not all the operations on the list have been executed, the function writes an executable operation file for the next operation on the list. If there are no more unexecuted operations on the list, the function gets the next plan node to execute by traversing the plan further (see Section 4.2.6), builds a list of executable operations to carry out that node, and writes an executable operation file for the first operation on the list.

8.4.6 Workpl_next_op2

The *workpl_next_op2(int * operation_class, char * op_file_name)* function takes two arguments:

1. *operation_class* — a pointer to an integer representing the type of

one_operation; the value is set by the function to the integer code for one of: NONE, MACHINING, or INSPECTION.

2. *op_file_name* — a string into which the function writes the name of the STEP Part 21 operation file (which already exists).

The general meaning of *workpl_next_op2* is: examine the stage-two process plan being executed and determine what the next operation to do should be. The function removes the first entry on the list of operations which is the stage-two plan and copies the operation class and the file name from that entry into *operation_class* and *op_file_name*, respectively.

8.4.7 Workpl_open_plan1

The *workpl_open_plan1(char * plan_file_name, char * setup_file_name)* function takes one argument:

1. *plan_file_name* — the name of a STEP Part 21 stage-one work-level process plan file to open. It is an error if the file does not exist when this function is called. The files referenced in the plan file must also exist.
2. *setup_file_name* — the name of a STEP Part 21 setup file to open. It is an error if the file does not exist when this function is called. All the files referenced in the setup file must also exist. The name is copied out of the plan into the *setup_file_name*.

The general meaning of *workpl_open_plan1* is: get ready to make and/or inspect a number of features on a workpiece using the data included or referenced in this plan file.

When this function is called, the Work Planner attaches to the Modeler. Then it reads five or six STEP Part 21 data files, makes internal representations of the data in the files, checks the data, and preprocesses the data. The six files are: process plan, setup, part_out, part_in, fixture, and features. For pure inspection, the part_in and part_out files are the same, so only one of them is read. The function tells the Modeler to model the features and the fixture. The function also tells the Modeler to model the part_in and/or part_out.

8.4.8 Workpl_open_plan2

The *workpl_open_plan2(char * full_plan_name)* function takes one argument:

1. *full_plan_name* — the full name of a STEP Part 21 stage-two work-level plan file to open. It is an error if the file does not exist when this function is called. All the files referenced in the plan file must also exist.

The general meaning of *workpl_open_plan2* is: get ready to make and/or inspect a number of features on a workpiece using the files referenced in this plan file. When this function is called, the Work Planner reads the plan file and finds the list of operations in it.

8.4.9 Workpl_plan_inspect_setup1

The *workpl_plan_inspect_setup1(char * setup_file_name, char * plan_file_name)* function takes two arguments:

1. *setup_file_name* — the name of a STEP Part 21 setup file to read. It is an error if the file does not exist when this function is called. All the files referenced in the

setup file, except for the process plan file (which will be overwritten if it already exists), must also exist.

2. *plan_file_name* — the name of the stage-one STEP Part 21 work-level process plan file that is written. The suffix "_1work" is added to the name given in the setup file.

The general meaning of *workpl_plan_inspect_setup1* is: make a stage-one work-level process plan for inspecting the features given in the features file in the setup of the given *setup_file_name*. When this function runs, it writes a STEP Part 21 ALPS work-level process plan file.

To start, the function attaches to Modeler, reads the setup file, and models the part, fixture, and features.

For each feature to be inspected, a "feature_plus" is built. The feature_plus contains additional information about the feature, including a pointer to Modeler's model of the feature. While building the feature_plus, it is checked that the feature does not intersect the part being inspected. The access volume (see Section 4.3.6) of the feature is found by the Modeler and it is checked that the access volume does not intersect the part. That check insures that it is possible to approach the feature from the direction of the feature's native Z-axis.

An inspect_feature_geometry node is built for each feature to be inspected, a type of probe tool to use to inspect the feature is selected and recorded in the node, and tool use parameters are selected and recorded in the node.

The inspect_feature_geometry nodes are given the trivial ordering that they may be executed in any order, and a stage-one work-level ALPS plan is written. If more than one feature is to be inspected in the setup, "execute in any order" is embodied by (i) making a parameterized_split_node (with m_number zero and serial timing) and having its successors be the inspect_feature_geometry nodes, and (ii) making a path_join_node and having it be the successor of each inspect_feature_geometry node. If the options settings indicate that inspection should start with locating the part precisely, a locate_part node is inserted in the plan before any inspect_feature_geometry nodes.

8.4.10 Workpl_plan_setup1

The *workpl_plan_setup1(char * setup_file_name, char * plan_file_name)* function takes two arguments:

1. *setup_file_name* — the name of a STEP Part 21 setup file to read. It is an error if the file does not exist when this function is called. All the files referenced in the setup file, except for the process plan file (which will be overwritten if it already exists), must also exist.
2. *plan_file_name* — the name of the stage-one STEP Part 21 work-level ALPS process plan file that is written. The function writes the full plan name into the character string at which the argument points. The name is formed by adding the suffix "_1work.stp" to the base plan name given in the setup file.

The general meaning of *workpl_plan_setup1* is: make a stage-one work-level process plan for machining and, possibly, inspecting the features given in the features file in the setup specified by the setup file. What gets inspected depends upon tolerances given in the design of the part_in

and upon settings given in shop options (see Section 8.5.2). When this function runs, it writes a STEP Part 21 ALPS work-level process plan file of the name it writes.

To make a stage-one plan, the function first attaches to Modeler and reads the setup file of the given *setup_file_name*. This tells it what the name of the plan it writes should be, as well as the names of files to read (describing features to make, fixture, part_in, and part_out), and what the orientation of the setup should be. The function reads all the files and calls on the Modeler to set up solid models of all parts and features in the correct position. The part_now is made as a copy of the part_in (and is continually updated as work proceeds). A check is made that the part_out as positioned by the setup is contained in the part_in as positioned by the setup.

For each feature to be made, a "feature_plus" is built with a lot of help from the Modeler. The feature_plus contains additional information about the feature, including but not limited to:

1. a pointer to the Modeler's model of the feature.
2. a pointer to the access volume of the feature (see Section 4.3.6).
3. a pointer to the block bodies of the feature (see Section 4.3.7).
4. the physical block_bys of the feature. This is a list of pointers to other feature_pluses whose parent features physically block access to the feature.

All the features to be made in the setup should be makeable, or the Cell Planner would not have assigned them to the setup. It may be necessary to make some before others. The plan, however, has no information about what order to make them in, so a procedure is needed for putting them in a makeable order.

The procedure makes groups of features. The features in each group may be made in any order. The groups are totally ordered. The procedure is as follows.

1. Put all the features whose physical block_bys are empty in the first group and consider them to have been made.
2. Put all the features in the next group whose physical block_bys parent features have been previously considered made.
3. Repeat step 2 until all the features are considered made.

The function decides which features are to be inspected, a decision depending on Work Planner option settings and tolerances of feature parameters. Any features to be inspected are to be inspected immediately after being machined.

For each feature to be machined, the Work Controller selects (1) a subtype of cutting node appropriate to make the feature and (2) a type of cutting tool with which to perform the operation. The cutting tool type is selected from the tool catalog as a catalog id number. Values are selected for tool use parameters (feed, speed, etc.) by using the tool_usage_rules, as described in Section 12.1.11.

For each feature to be inspected, the Work Controller selects (1) an inspection operation node appropriate to the feature and (2) a touch probe with which to perform the operation. The only tool-use parameter for a probe is feed rate, and that is set arbitrarily; the tool_usage_rules are not used for probes.

Using the following rules, a stage-one work-level ALPS plan is made embodying the feature ordering determined above.

1. For each feature, make an appropriate subtype of cutting node and, if the feature is to be inspected, also make an inspect_feature_geometry node and set it to be the successor of the cutting node.
2. If a group of features has two or more features, (i) make a parameterized_split_node (with m_number zero and serial timing) and have its successors be the cutting nodes for the features, and (ii) make a path_join_node and have it be the successor of either the inspect_feature_geometry node for the feature, if it exists, or the cutting node for the feature, if not.
3. If a group of features has two or more features, the successor of the join_node for the group is: (i) if there are no more groups, an end_plan_node, (ii) if the next group has one feature, the cutting node for the feature, or (iii) if the next group has two or more features, the parameterized_split_node for the group.
4. If a group of features has one feature, the successor of the inspect_feature_geometry node for the feature, if it exists, or the cutting node for the feature, if not, is: (i) if there are no more groups, an end_plan_node, (ii) if the next group has one feature, the cutting node for the feature, or (iii) if the next group has two or more features, the parameterized_split_node for the group.

Finally, the function checks that the shape of the part_now (which has been updated periodically) is the same as the shape of the part_out.

When its work is done, the function sends a `MODEL_DETACH_MSG` to the Modeler to disconnect from the Modeler and re-initializes the Work Planner world model.

8.4.11 Workpl_plan_setup2

The *`workpl_plan_setup2(char * base_plan_name)`* function takes one argument:

1. *`base_plan_name`* — the base name of a STEP Part 21 work-level plan file. The file whose name is the *`base_plan_name`* with the suffix "_1work.stp" must exist and be a stage-one work-level process plan.

The general meaning of *`workpl_plan_setup2`* is: make a stage-two plan corresponding to the existing stage-one plan. The stage-two plan will have the same base name with the suffix "_2work.stp".

To start, the function reads the stage-one plan, the setup it names, and the features file named in the setup.

Then the function makes a list of one_operations (call it the O-list) and an executable operation file for each one_operation on the O-list. To do this, the Work Planner world model maintains a list of executable operations it has already planned to do (call it the E-list) and keeps track of where it is in the E-list. If not all the operations on the E-list have been used to make one_operations, the function writes an executable operation file for the next operation on the E-list and makes a corresponding one_operation and puts it on the O-list. If there are no more unprocessed operations on the E-list, the function makes a new E-list to work from.

To make the new E-list, the function gets the next plan node to execute by traversing the plan further (see Section 4.2.6). Executable operations needed to carry out that node are determined by the function and put on the E-list. Some plan nodes require only one operation, and some require half a dozen. If two successive cutting nodes use different tools, for example, an operation to turn

coolant off, an operation to change the tool, an operation to turn coolant back on, and an operation to do the cutting are needed. If the same tool is used by the two cutting nodes, only the operation to do the cutting is needed.

When there is no next plan node, the work of the function is finished and the stage-two plan is written out with the O-list in it.

8.5 Work Planner Options

The Work Planner uses both work options (options used only by the Work Planner) and shop options. The Fbics_Work process reads the shop options and work options files when it initializes or re-initializes and saves the settings they describe in its world model.

8.5.1 Work Options

Work options are used only by the Work Planner (see Section 12.1.8). They include four options used only during inspection: maximum angle error (in original part location), maximum origin error (in original part location), maximum shape error, and locating method.

8.5.2 Shop Options

The shop options used by the Work Planner are a subset of the shop options (see Section 12.1.6). These are the length_unit_rule, inspecting_decision, inspecting_level, milling_tolerance_default, milling_tolerance_tightest, tool catalog file name, tool inventory file name, and tool usage file name.

9 Task Controller

This section describes the Task Controller in moderate detail.

Throughout this section, the following typesetting conventions are used.

1. User commands and their arguments are set in plain `courier font`.
2. API functions and their arguments are set in *`courier italic font`*.

Messages continue to be set in `courier font`, but since they all end in `_MSG`, they should not be confused with user command items.

9.1 Task Controller Architecture

The stand-alone Task Controller is shown in Figure 15. This is a more detailed view than shown in Figure 1, but has the same connections. The stand-alone Task Controller is the most complex of the three controllers because it includes two processes, the Fbics_Task process and the Fbics_Task2 process. As discussed earlier, there are two processes so that integrated systems using existing controllers (one with a DMIS interpreter built in and one with an RS274 interpreter built in) are easy to build. For an integrated system, the Fbics_Task2 process is removed, and messages that would have gone to Fbics_Task2 are sent instead to an actual machine controller.

This is an oddball architecture used only for convenience. The most unusual feature of Figure 15 is that the Task Planner includes all of the Fbics_Task2 process and all of the Fbics_Task process except for the Task Controller Body. A more sensible architecture for a Task Controller with the same functionality is shown in Figure 2.

The Task Controller may get commands either from the user or from the Work Controller, or both. The user gives commands via a simple text-based user interface. The Work Controller gives commands by sending messages. Commands from both sources are handled by the Task Controller Body. In response to these commands, the Task Controller Body makes function calls to functions in the Task Planner API and/or sends messages to the Fbics_Task2 process.

To carry out the API function calls made to it, the Task Planner (1) gets data from the Data Repository and/or sends data to it, (2) sends messages to the Modeler and gets answers back, and (3) makes or interprets a file of NC code or DMIS code, according to what the API function call says to do.

The Task Controller Body interacts directly with the Work Controller, the Fbics_Task2 process, and the user, but does not interact directly with the Modeler or the Data Repository.

The Task Planner interacts directly with the Modeler and the Data Repository, but does not interact directly with the Work Controller or the user.

In retrospect, the overall architecture might be cleaner if the messages to the Fbics_Task2 process were not sent from the Task Controller Body but were sent instead from the part of the Task Planner inside the Fbics_Task process. With that change, the Task Controller Body would deal with the Task Planner solely by making API function calls, just as the Cell Controller Body and Work Controller Body deal with their planners solely via an API.

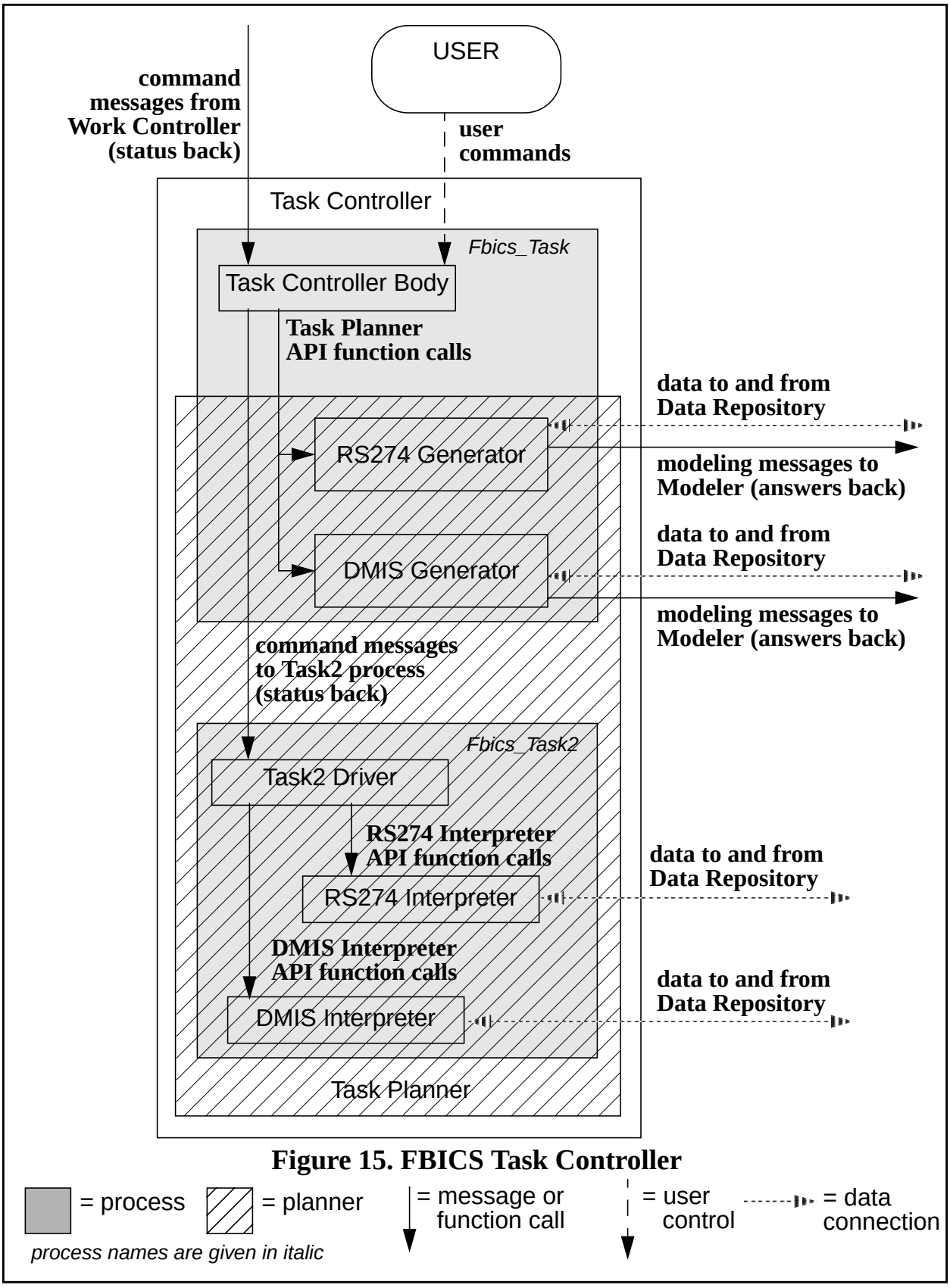


Figure 15. FBICS Task Controller

9.2 User Interface to the Task Controller

This section discusses the user interface to the Task Controller, including a description of the terminal window, an overview of the user commands, and details of each user command.

9.2.1 Task Controller Terminal Window

The Task Controller terminal window may be enabled or disabled as a user interface. The user can give any of the user commands described in Section 9.2.2 when given the prompt `Task =>` for user input. This happens only when user input is enabled. The user can still type in commands when user input is disabled, but the Task Controller does not read them.

When the Task Controller is started, MANUAL or AUTO may, optionally, be included as a command line argument. If AUTO is included, the Task Controller starts up in AUTO mode with user input disabled. Otherwise, the Task Controller starts up in MANUAL mode and user input is enabled. The “fbics” script used to start stand-alone FBICS starts the Task Controller in AUTO mode.

The Task Controller is always ready to respond to command messages from the Work Controller. When the Task Controller is in MANUAL mode, however, after a command message is received from the Work Controller, the Task Controller waits until the user presses the <enter> key before executing the command message. In AUTO mode, the Task Controller executes command messages immediately.

9.2.2 User Commands to the Task Controller

The user commands to the Task Controller are displayed as shown in Figure 16 if the user types `help`. The figure shows the name, arguments, and meaning of each command.

A peculiar aspect of the Task Controller is that, although it focuses on the machining or inspection of single shape features, it needs to be aware of details of what its superior (the Work Controller) focuses on: the setup in which features are being made or inspected. The Work Controller, by contrast, does not need to be aware of details of what its superior (the Cell Controller) focuses on: the entire part. Thus, the Task Controller has (perhaps unexpected) user commands and API functions for opening and closing setups.

When user commands are executed by the Task Controller, functions from the Task Planner API (described in Section 9.4) are called and command messages (described in Section 9.7) may be sent to the Fbics_Task2 process. Whenever a command message is sent to the Fbics_Task2 process, the Task Controller Body waits for a response before proceeding.

```
help = print this list of commands
quit = quit the task controller
auto = run automatic (keyboard dead) until setup is closed
init()
open_setup(setup file name)
generate_nc(operation file name, NC file name)
execute_nc(NC file name)
generate_dmis(operation file name, DMIS file name)
execute_dmis(DMIS file name)
close_setup()
exit()
```

Figure 16. Task Controller Help

9.2.3 Help

The user command `help` causes the Task Controller to print the list of commands shown in Figure 16. This command has no arguments and is not followed by parentheses.

9.2.4 Quit

The user command `quit` causes the Task Controller to quit immediately without tidying up the way `exit` does. The terminal window in which the controller was running disappears. No other processes exit. This command has no arguments and is not followed by parentheses.

9.2.5 Auto

The user command `auto` causes the Task Controller to switch into AUTO mode, with the effects described above.

Since user input is not processed in AUTO mode, there is no Task Controller user command to switch into MANUAL mode. To switch the Task Controller from AUTO mode to MANUAL mode, the `task_manual()` command must be given to the Work Controller (see Section 8.2.12).

9.2.6 Init

The user command `init()` causes the Task Controller to initialize itself as follows. The Task Controller Body calls the *`taskpl_init`* function, causing the Fbics_Task process to initialize itself. Then the Task Controller Body sends a `TASK2_INIT_MSG` to the Fbics_Task2 process, and waits for and checks the reply from the Fbics_Task2 process.

If the Task Controller is already initialized when the `init()` command is given, the user is

prompted to decide whether to re-initialize. If the user chooses to re-initialize, only the call to *taskpl_init* is made. No TASK2_INIT_MSG is sent.

9.2.7 Open_setup

The user command open_setup(setup_file_name) causes the Task Controller to open the setup file named in the command. Setup files are described in Section 12.1.16.

The setup_file_name may be the empty string. In this case, nothing is done. Opening the setup this way is done if DMIS or NC code execution is to be done, but not code generation.

To execute the command when the setup_file_name is not the empty string, a call is made to *taskpl_open_setup*, with the effects described in Section 9.4.6.

9.2.8 Generate_nc

The user command generate_nc(operation_file_name, nc_file_name) causes the Task Controller to write a file with the given nc_file_name containing NC code for carrying out the operation described in the file with the given operation_file_name.

To execute the command, a call is made to *taskpl_generate_nc*, with the effects described in Section 9.4.4.

9.2.9 Execute_nc

The user command execute_nc(nc_file_name) causes the Task Controller to execute the NC code in the file with the given nc_file_name.

To execute the command, a TASK2_EXEC_NC_MSG is sent to the Fbics_Task2 process, with effects as described in Section 9.7.2.

9.2.10 Generate_dmis

The user command generate_dmis(operation_file_name, dmis_file_name) causes the Task Controller to write a file with the given dmis_file_name containing DMIS code for carrying out the operation described in the file with the given operation_file_name.

To execute the command, a call is made to *taskpl_generate_dmis*, with the effects described in Section 9.4.3.

9.2.11 Execute_dmis

The user command execute_dmis(dmis_file_name) causes the Task Controller to execute the DMIS code in the file with the given dmis_file_name.

To execute the command, a TASK2_EXEC_DMIS_MSG is sent to the Fbics_Task2 process, with effects as described in Section 9.7.1.

9.2.12 Close_setup

The user command close_setup() causes the Task Controller to close the currently open setup.

If the setup was opened with an empty string for the setup name, nothing is done to close the setup.

If the setup was opened with a setup name identifying a setup file, the *taskpl_close_setup* function is called with effects as described in Section 9.4.1.

9.2.13 Exit

The user command `exit()` causes the Task Controller Body to call *taskpl_exit* (which works as described in Section 9.4.2) and to send a `TASK2_EXIT_MSG` to the Fbics_Task2 process. Then the Fbics_Task process stops running (without waiting for a reply from the Fbics_Task2 process). The terminal window in which the process was running disappears. The Fbics_Task2 process and its terminal window also disappear (when the Fbics_Task2 process executes the `TASK2_EXIT_MSG`).

9.3 Command Messages to the Task Controller

Ten command messages to be sent to the Task Controller (by the Work Controller) are defined, as follows. These messages are handled by the Fbics_Task process. All messages have an integer-valued `sequence_number` field. This field is a sequence number as described in Section 5.4.4.

If the Task Controller is in AUTO mode, it carries out each message as soon as the message is received. If the Task Controller is in MANUAL mode, it waits for the user to press the <enter> key before carrying out the message.

9.3.1 TASK_CLOSE_MSG

The `TASK_CLOSE_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `TASK_CLOSE_MSG` is identical to the effect of carrying out the user command `close_setup`, as described in Section 9.2.12.

9.3.2 TASK_EXEC_DMIS_MSG

The `TASK_EXEC_DMIS_MSG [int sequence_number, char dmis_file_name[80]]` has one field in addition to the `sequence_number`.

1. `dmis_file_name` — the name of the DMIS code file to execute.

To carry out the message, a `TASK2_EXEC_DMIS_MSG` is sent to the Fbics_Task2 process, with effects as described in Section 9.7.1.

9.3.3 TASK_EXEC_NC_MSG

The `TASK_EXEC_NC_MSG [int sequence_number, char nc_file_name[80]]` has one field in addition to the `sequence_number`.

1. `char nc_file_name` — the name of the NC code file to execute.

To carry out the message, a `TASK2_EXEC_NC_MSG` is sent to the Fbics_Task2 process, with effects as described in Section 9.7.2.

9.3.4 TASK_EXIT_MSG

The `TASK_EXIT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `TASK_EXIT_MSG` is identical to the effect of carrying out the user command `exit`, as described in Section 9.2.13.

9.3.5 TASK_GEN_DMIS_MSG

The `TASK_GEN_DMIS_MSG [int sequence_number, char op_file_name[80], char dmis_file_name[80]]` has two fields in addition to the `sequence_number`.

1. `char op_file_name` — the name of the executable operation file to read.
2. `char dmis_file_name` — the name of the DMIS code file to write.

The effect of carrying out a `TASK_GEN_DMIS_MSG` is identical to the effect of carrying out the user command `generate_dmis`, as described in Section 9.2.10.

9.3.6 TASK_GEN_NC_MSG

The `TASK_GEN_NC_MSG [int sequence_number, char op_file_name[80], char nc_file_name[80]]` has two fields in addition to the `sequence_number`.

1. `char op_file_name` — the name of the executable operation file to read.
2. `char nc_file_name` — the name of the NC code file to write.

The effect of carrying out a `TASK_GEN_NC_MSG` is identical to the effect of carrying out the user command `generate_nc`, as described in Section 9.2.8.

9.3.7 TASK_IDLE_MSG

The `TASK_IDLE_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `TASK_IDLE_MSG` is to do nothing. This message is not currently used.

9.3.8 TASK_INIT_MSG

The `TASK_INIT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `TASK_INIT_MSG` is identical to the effect of carrying out the user command `init`, as described in Section 9.2.6.

9.3.9 TASK_MANUAL_MSG

The `TASK_MANUAL_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `TASK_MANUAL_MSG` is to put the Task Controller into MANUAL mode. In MANUAL mode, the Task Controller pays attention to the keyboard and does not take action on future messages received until the user presses the <enter> key. When the Task Controller is in MANUAL mode, the user may put it into AUTO mode by giving the user

command `auto`, as described in Section 9.2.5. In AUTO mode, the Task Controller does not pay attention to the keyboard and takes action on future messages as soon as they are received.

9.3.10 TASK_OPEN_MSG

The `TASK_OPEN_MSG [int sequence_number, char setup_file_name[80]]` has one field in addition to the `sequence_number`.

1. `setup_file_name` — the name of the setup file to open.

The effect of carrying out a `TASK_OPEN_MSG` is identical to the effect of carrying out the user command `open_setup`, as described in Section 9.2.7.

9.4 Task Planner API Functions

The Task Planner API functions described in this section are called when the Task Controller executes the user interface commands described in Section 9.2 or the command messages from the Work Controller described in Section 9.3.

The Task Controller Body uses the Task Planner by making calls to functions in the Task Planner API or by sending messages to the Fbics_Task2 process. The Task Planner API includes six interface functions.

API functions for executing NC code files or DMIS code files are not included in the Task Planner API because execution is performed in the Fbics_Task2 process and the request for execution does not go through the Fbics_Task process. As discussed at the end of Section 9.1, the architecture would be cleaner if the Task Planner API did include functions for executing NC code files and DMIS code files.

9.4.1 Taskpl_close_setup

The *`taskpl_close_setup()`* function takes no arguments.

The general meaning of *`taskpl_close_setup`* is: complete work on the currently open setup and get ready to do something else. Data associated with the current setup are deleted from memory. The Task Planner sends a `MODEL_DETACH_MSG` to the Modeler to disconnect from the Modeler. The Task Planner world model is re-initialized.

9.4.2 Taskpl_exit

The *`taskpl_exit()`* function takes no arguments.

The general meaning of *`taskpl_exit`* is: stop running. Tool, fixture, and setup data are removed from the Task Planner world model.

9.4.3 Taskpl_generate_dmis

The *`taskpl_generate_dmis(char * op_file_name, char * dmis_file_name)`* function takes two arguments:

1. *`op_file_name`* — the name of a STEP Part 21 operation file to read which describes the operation to be executed as well as any feature used in the operation. It is an error if the file does not exist when this function is called.
2. *`dmis_file_name`* — the name of a DMIS file to write. If the file already exists, it

will be overwritten.

The general meaning of *taskpl_generate_dmis* is: generate a DMIS file. The generated file may have code to inspect an AP 224 feature, or it may have code for ancillary operations, such as changing a tool. Details of DMIS generation are given in Section 9.6.

9.4.4 Taskpl_generate_nc

The *taskpl_generate_nc(char * op_file_name, char * nc_file_name)* function takes two arguments:

1. *op_file_name* — the name of a STEP Part 21 operation file to read which describes the operation to be executed as well as any feature used in the operation. It is an error if the file does not exist when this function is called.
2. *nc_file_name* — the name of an NC file to write. If the file already exists, it will be overwritten.

The general meaning of *taskpl_generate_nc* is: generate an NC code file. The generated file may have code to cut an AP 224 feature, or it may have code for ancillary operations, such as turning coolant off. The dialect of RS274 in which the file is written will be either RS274/NGC, or the Hexapod dialect, depending on current settings of task options. If the operation file describes a feature cutting operation, this function also updates the model of the workpiece by subtracting the feature from the workpiece. Details of NC generation are given in Section 9.5.

9.4.5 Taskpl_init

The *taskpl_init()* function takes no arguments.

The general meaning of *taskpl_init* is: Get ready to run. The Task Planner world model is initialized. The Task Planner reads the shop options and task_options files and transcribes relevant data into its world model. The Task Planner reads the tool catalog and the tool inventory and puts pointers to them into its world model. All files read are STEP Part 21 files.

9.4.6 Taskpl_open_setup

The *taskpl_open_setup (char * setup_file_name)* function takes one argument:

1. *setup_file_name* — the name of a STEP Part 21 setup file to read. It is an error if the file does not exist when this function is called.

The general meaning of *taskpl_open_setup* is: get ready to make and/or inspect a number of features on a workpiece using the data included or referenced in this setup file. When this function is called, the Task Planner attaches to the Modeler. Then it reads three or four STEP Part 21 data files, makes internal representations of the data in the files, checks the data, and preprocesses the data. The four files are: setup, part_in, part_out, and fixture. For pure inspection, the part_out file is not read, since it is the same as the part_in. For machining, the function tells the Modeler to model the part_out, part_in, and part_now (by copying the part_in). For pure inspection, only the part_now is modeled.

9.5 The FBICS RS274 NC Code Generator

The RS274 NC code generator in the Fbics_Task process takes in a machining executable operation description and generates code for performing the operation by 3-axis machining. The

executable operation may require very little code (turn coolant off, for example) or a lot of code (finish mill a large pocket, for example). The code is generated initially as pseudocode. The pseudocode is stored briefly in active memory.

One of two NC code writers translates the pseudocode for an executable operation into a file of NC code in one of two dialects, RS274/NGC or Hexapod, depending on which is selected by the task options. The descriptions of generators that follow describe the pseudocode that is generated. Where the NC code writers differ significantly in what they write from the same pseudocode, the differences are described.

Section 12.1.15 describes 25 executable operations, 19 of which are for machining. The FBICS RS274 code generator can generate code for 7 of the 19: coolant_ex, counterboring_ex, end_nc_ex, finish_mill_ex, nc_change_ex, start_nc_ex, and twist_drilling_ex. The generator for finish_mill_ex is implemented only for rectangular pockets. There is a separate function for generating code for each of the executable operations. Adding the capability to generate code for most of remaining 12 machining operations would not be difficult.

The code generator functions for the 7 executable machining operations are described in the following subsections. The information used by the code generator functions comes from three sources: the arguments of the function, the Task Planner world model, and the task options (via the world model). World model information used by the generators includes such things as where the tool tip is when the function starts, whether flood coolant is on or off, what the current feed and speed rates are, etc. The function descriptions that follow list the world model and options information used by each function, in addition to the arguments. World model information is updated by the functions.

9.5.1 Coolant_ex Generator

The coolant_ex operation deals only with flood coolant. The function that handles coolant_ex takes one argument specifying:

1. whether coolant should be on or off.

The function uses world model information regarding:

1. whether coolant is currently on or off.

If the coolant is already set the way it is supposed to be set, the function writes no code. If the coolant is supposed to be on and it is off, the function writes one line of code to turn the coolant on. If the coolant is supposed to be off and it is on, the function writes one line of code to turn the coolant off.

9.5.2 Counterboring_ex Generator

The function that handles counterboring_ex takes arguments specifying:

1. a description of the feature to be counterbored; it must be a round_hole.
2. whether flood coolant should be on or off.
3. spindle speed.
4. feed rate.

The location and depth of the hole are extracted from the description of the hole being counterbored.

The function uses world model information regarding:

1. current tool tip position.
2. whether flood coolant is currently on or off.
3. current spindle speed.
4. whether and which way the spindle is currently turning.
5. current feed rate.

The procedure is:

1. Traverse parallel to the Z-axis to retract_high if not already at that height.
2. If necessary, turn flood coolant on or off.
3. If necessary, change spindle speed,
4. If necessary, start spindle clockwise.
5. If necessary, change the feed rate.
6. Traverse parallel to the XY-plane to the XY location of the hole.
7. Traverse parallel to the Z-axis to one retract_distance_low (a machining option) above the top of the hole.
8. Feed to the bottom of the hole.

When this finishes, the tool tip is at the bottom of the hole.

9.5.3 End_nc_ex Generator

The function that handles end_nc_ex takes no arguments.

The function uses world model information regarding:

1. whether flood coolant is currently on or off.
2. whether the spindle is currently turning.

The function uses options information regarding:

1. the type of place to move to at the end of a program.
2. the location of a place to move to.

The procedure is:

1. Stop the spindle turning if it is not already stopped.
2. Turn flood coolant off if it is not already off.
3. Write zero, one, or two lines of code to move the machine to its end_location at traverse rate.
4. Write the code needed to end a program.

There are four options (as indicated in the task options file) for end_location: leave the position wherever it happens to be, leave the position wherever it happens to be in XY but retracted in Z, move to home1, move to home2. The locations of home1 and home2 are other task options.

9.5.4 Finish_mill_ex Generator

The function that handles finish_mill_ex takes arguments specifying:

1. a description of the feature to be finish milled; it must be a rectangular_closed_pocket.
2. a description of the specific cutting tool to be used.

3. stepover (radial offset specifying horizontal cut depth, used in zig-zag cutting).
4. spindle speed.
5. feed rate.
6. whether flood coolant should be on or off.

The location, length, width, corner radius, and depth of the pocket are extracted from the description of the pocket.

The function and/or its subordinates use world model information regarding:

1. current tool tip position.
2. whether the cut adjustment (described below) should be set.
3. whether flood coolant is currently on or off.
4. current spindle speed.
5. whether and which way the spindle is currently turning.
6. current feed rate.

The function and/or its subordinates use options information regarding:

1. finish cut thickness.
2. how far to retract between cuts.

The generator function for finish_mill_ex participates in adaptive machining as described in Section 3.7.7. This is implemented by having a cut_adjustment which makes the length, width, and corner radius of the nominal pocket being machined a little bigger or a little smaller. The cut adjustment is always used, but is zero unless adaptive machining is in progress.

The general procedure for a rectangular pocket is:

1. Check the world model to see if the cut adjustment should be set and set it, if so.
2. Traverse parallel to the Z-axis to retract_high if not already at that height.
3. If necessary, turn flood coolant on or off.
4. If necessary, change spindle speed,
5. If necessary, start spindle clockwise.
6. If the width of the pocket is more than two tool diameters, finish mill the bottom of the pocket in a zig-zag pattern (if the width is smaller than two tool diameters, the bottom will be finished while the sides are being finished).
7. Finish mill the sides of the pocket at the full depth of the pocket. The approach and retract paths for this are helical, to avoid plunge or dwell marks on the sides of the pocket.

The general procedure includes several possible changes of feed rate not listed above. If the tool diameter is the same as the width of the pocket, steps 5 to 7 in the general procedure are replaced by one or two motions: feed straight in, then (if necessary) feed straight parallel to the XY plane. In this case the tool is left at the bottom of the pocket.

The finish_mill_ex operation is intended for use with a pocket that has been rough-cut, so that only a thin layer of material needs to be removed on the sides and bottom of the pocket. The finish_mill_ex operation may, however, be used to make the entire pocket in solid material in soft materials (such as wax) which have not been rough-cut.

9.5.5 Nc_change_ex Generator

The function that handles nc_change_ex takes one argument specifying:

1. a description of the specific cutting tool to be used.

The function and/or its subordinates use world model information regarding:

1. whether the spindle is turning or not.
2. the z-coordinate of the origin currently in use.
3. the high retract location of the current setup.

The function and/or its subordinates use options information regarding:

1. whether the machine has an automatic changer.
2. the type of place to move to for changing the tool.
3. the maximum tool length offset.
4. the location of a place to move to.

If the correct tool is already in the spindle, nothing is done. Otherwise, the procedure is:

1. Signal an error if coolant is not off.
2. Stop the spindle if it is not already stopped.
3. a. If the Task Planner world model change_location is STAY, do not move.
   b. If the Task Planner world model change_location is Z_UP, retract as high as possible.
   c. If the Task Planner world model change_location is HOME1 (i) retract vertically to the Z-coordinate of HOME1 in machine coordinates, (ii) move horizontally to the XY location of HOME1 in machine coordinates.
   d. If the Task Planner world model change_location is HOME2 (i) retract vertically to the Z-coordinate of HOME2 in machine coordinates, (ii) move horizontally to the XY location of HOME2 in machine coordinates.
4. Select the tool.
5. Change to the selected tool.
6. Use the tool length offset for the selected tool.
7. Move vertically to totally retracted minus max_tool_length_offset.

The last item is so the z-location of the tool tip is known. A move is needed for the generator to know that location since the tool length offset will usually be different after the change, and the generator does not know what that number is. The method of calculating the value was chosen as the safest method. The max_tool_length_offset must be set realistically.

9.5.6 Start_nc_ex Generator

The function that handles start_nc_ex takes no arguments.

The function uses world model information regarding:

1. the z-coordinate of the origin currently in use.
2. the high retract location of the current setup.

The function uses options information regarding:

1. the maximum tool length offset.
2. the location of a place to move to.
3. the z-value in machine coordinates when the Z-axis is fully retracted.

The procedure is:

1. Write the line or lines of NC code that begin a program. The RS274/NGC and Hexapod code writers start programs very differently.
2. Retract the Z-axis as far as feasible.
3. Traverse parallel to the XY plane to a point above the origin currently in use.

9.5.7 Twist_drilling_ex Generator

The function that handles twist_drilling_ex takes arguments specifying:

1. a description of the feature to be drilled; it must be a round_hole.
2. the incremental depth for peck drilling.
3. whether flood coolant should be on or off.
4. spindle speed for drilling.
5. feed rate for drilling.

The location and depth of the hole are extracted from the description of the hole being drilled.

The function uses world model information regarding:

1. current tool tip position.
2. whether flood coolant is currently on or off.
3. current spindle speed.
4. whether and which way the spindle is currently turning.
5. current feed rate.

The function uses options information for:

1. determining whether a hole is to be considered a deep hole.
2. deciding what technique to use for drilling deep holes.
3. deciding how far to retract while peck drilling and when finished.

The procedure is:

1. Traverse parallel to the Z-axis to retract_high if not already at that height.
2. If necessary, turn flood coolant on or off.
3. If necessary, change spindle speed,
4. If necessary, start spindle clockwise.
5. If necessary, change the feed rate.
6. Traverse parallel to the XY-plane to the XY location of the hole.
7. If ratio of the depth of the hole to its diameter is greater than the deep_hole_factor given in the task options and the deep_drill_cycle task option says to use peck drilling for deep holes, then write a line of code for peck drilling. Otherwise write a line of code for plunge drilling.

The RS274/NGC and Hexapod NC code writers write very different drilling code. The RS274/NGC writer writes one line of code using a G81 cycle for plunge drilling and one line using a G83 cycle for peck drilling. The Hexapod writer writes G0 and G1 lines for both kinds of drilling; for

peck drilling this may be half a page or more of code.

When this finishes, the tool tip is at the XY location of the hole, one retract_distance_low (a machining option) above the top of the hole.

9.6 The FBICS DMIS Code Generator

The DMIS code generator in the Fbics_Task process takes in an inspection executable operation description and generates a file of DMIS code for performing the operation on a 3-axis coordinate measuring machine or a 3-axis machining center. The executable operation may require very little code (end the program, for example) or a lot of code (inspect a large pocket, for example).

The individual files generated by code generator are not entire DMIS programs themselves. If the files generated in one setup are concatenated together, however, they make a complete DMIS program.

The algorithm used by the code generator for selecting points on a planar or cylindrical surface is discussed in Section 4.4.5.

Section 12.1.15 describes 25 executable operations, 6 of which are for inspection. The FBICS DMIS code generator can generate code for 5 of the 6. Inspect_surface_ex is not implemented. There is a separate function for generating code for each of the executable operations.

The code generator functions for the 5 executable inspection operations are described in the following subsections. The information used by the code generator functions comes from three sources: the arguments of the function, the Task Planner world model, and the task options (via the world model). World model information used by the generators includes such things as where the tool tip is when the function starts, what the current feed rate is, etc. The function descriptions that follow list the world model and options information used by each function, in addition to the arguments. World model information is updated by the functions.

9.6.1 Start_inspect_ex Generator

The function that handles start_inspect_ex takes one argument specifying:

1. the file pointer to print to.

The function uses world model information regarding:

1. whether machining is also taking place.
2. length units in use.

The function uses options information regarding:

1. the z-value in machine coordinates when the Z-axis is fully retracted.

If machining and inspection are being performed together, the procedure is:

1. Write two lines with the standard opening of a DMIS file.
2. Write a line specifying the length and angle units to use.

When only inspection is being done, this function expects that the part will be located by placing it with the XY-plane of the part coordinate system parallel to the XY-plane of the machine doing the inspection. With that restriction, the part coordinate system may be located in the machine coordinate system if the location of the origin of part is known and the amount of rotation of the

part around its Z-axis is known. This function assumes the DMIS_variables file used by the DMIS interpreter will provide that information by containing values for variables named ORG_X, ORG_Y, ORG_Z, and ANGLE1. In the implementation at NIST, values of those variables are written into the DMIS_variables file by a vision system.

If only inspection is being performed, the procedure is:

1. Write two lines with the standard opening of a DMIS file.
2. Write two lines of DMIS code declaring variables.
3. Write a line of DMIS code specifying the length and angle units to use.
4. Write a few lines of DMIS code establishing a working coordinate system translated from the machine coordinate system to the part origin.
5. Write a few lines of DMIS code establishing a working coordinate system rotated to be aligned with the part coordinate system; this makes the working coordinate system of the program be the part coordinate system.
6. Write a line of DMIS code retracting the probe fully.

9.6.2 End_inspect_ex Generator

The function that handles end_inspect_ex takes one argument specifying:

1. the file pointer to print to.

The function uses world model information regarding:

1. whether machining is also taking place.
2. the current location.

The function uses options information regarding:

1. what sort of retract move to perform at the end of a program (to a given z-value or by a given distance).
2. whether to move to a home location at the end of the program.
3. the z-value in machine coordinates when the Z-axis is fully retracted.
4. the distance by which to retract.
5. the locations of two home positions.

The procedure is:

1. Write one GOTO line to retract as specified by the options in use.
2. If the end_location option is not “stay put”, write two lines to go to home1 or home2, as specified by the options in use.
3. Write one ENDFIL line to end the program.

9.6.3 Tool-using Inspection Executable Operations

The functions for inspect_change_ex (see Section 9.6.4), inspect_geometry_ex, (see Section 9.6.5) and locate_part_block_block_ex (see Section 9.6.6) all use a specific instance of a probe tool (as specified in the operation). They are preceded by a call to a function that does preparatory work, as follows.

The function that does preparatory work takes arguments specifying:

1. the executable operation.
2. the name of the file describing the executable operation.
3. the file pointer to print to.

The function uses world model information regarding:

1. the tool inventory.

The function uses no options information.

The procedure is:

1. Find in the tool inventory the tool instance specified in the operation.
2. If the operation is inspect_geometry_ex: set the feed rate as given in the operation, display the feature, and set the inspection level as given in the operation.
3. If the operation is locate_part_block_block_ex: set the feed rate as given in the operation, and set the inspection level as given in the operation.

9.6.4 Inspect_change_ex Generator

The function that handles inspect_change_ex takes arguments specifying:

1. the tool instance to change to.
2. the probe model in the tool instance to change to.
3. the file pointer to print to.

The function uses world model information regarding:

1. whether machining is also taking place.
2. the coordinate system currently in use.
3. the tool currently in use.
4. the tools that have already been defined in the DMIS program for the current setup.
5. the current location.

The function uses options information regarding:

1. what sort of retract move to perform for changing a tool (to a given z-value or by a given distance).
2. whether to stay put, retract, or go to a home location for changing a tool.
3. the z-value in machine coordinates when the Z-axis is fully retracted.
4. the distance by which to retract.
5. the locations of two home positions.

If the tool to change to is the tool currently in use, nothing is done. Otherwise, the procedure is:

1. Find the tool in the list of tools that have already been defined in the DMIS program for the current setup.
2. If the tool was not found in step 1, write two lines of DMIS code to define it, and add it to the list of tools that have already been defined in the DMIS program for the current setup.
3. If the options do not say to stay put for a tool change, write lines of DMIS code to retract or go to a home position. If necessary for going to a home position, the DMIS code will include changing coordinate systems.
4. Write a line of DMIS code to select the tool.

5. If the coordinate system was changed by item 3, write a line of DMIS code to change back to the previous coordinate system.
6. If the options do not say to stay put for a tool change, write lines of DMIS code to go back to the position that was the current position before the tool change.

9.6.5 Inspect_geometry_ex Generator

The function that handles inspect_geometry_ex takes arguments specifying:

1. the inspection level (low, medium, or high).
2. the feature to inspect.
3. the tool currently in use.
4. the feed rate.
5. the file pointer to print to.

The function and its subordinates use world model information regarding:

1. the current location.
2. the number of DMIS features defined in the DMIS program for the setup.
3. the high retract location of the current setup.

The function uses options information regarding:

1. the clearance distance to use.
2. the number of points to inspect for a given feature type at a given inspection level.

The procedure is:

1. If the desired feed rate differs from the current feed rate, write a line of DMIS code to set the feed rate.
2. If not already retracted, write a line of DMIS code to retract along the Z-axis.
3. If the feature is a hole:
   (a) find the probe points for the cylinder that is the inside of the hole using the algorithm described in Section 4.4.5,
   (b) write two lines of DMIS code defining a cylinder representing the inside of the hole,
   (c) write a number of lines of DMIS code for measuring the cylinder, using the probe points already selected.
   (d) if there is a diameter tolerance on the hole, write a line a line of DMIS code defining the tolerance,
   (e) write a line of DMIS code to output the nominal cylinder,
   (f) write a line of DMIS code to output the actual cylinder and its actual diameter tolerance (if a tolerance was defined).
4. If the feature is a pocket:
   the pocket may be bounded by up to five planes (the four sides and the bottom) and up to four partial cylinders (the corners) — and there are various degenerate cases. The function determines which of the planes and cylinders that might be present actually are present and writes DMIS code to inspect them as follows.
   (a) for each plane present, find probe points using the algorithm described in Section 4.4.5 and write lines of DMIS code defining the plane and measuring the plane,

using the probe points already selected,
(b) for each plane present, write two lines of code DMIS code outputting the nominal and actual planes,
(c) for each full or partial cylinder present, follow the procedure described in step 3 for a hole.

5. If the Z-coordinate of the current position is less than fully retracted, write a line of DMIS code to retract along the Z-axis.

In the procedures above, the number of probe points to select for each plane and cylinder are determined by looking at the task options, which specify for each DMIS feature type and each inspection level, how many points should be used (see Section 9.9). The procedure descriptions omit mention of the many checks made by the functions.

9.6.6 Locate_part_block_block_ex Generator

The function that handles locate_part_block_block_ex[1] probes the sides of the block that is the base shape of a part to determine precise part location and reset the current coordinate system. The current coordinate system before this function runs is expected to be close to the coordinate system of the block. The executable operation data includes the coordinates of three corners of the block in the current coordinate system. The function takes arguments specifying:

1. the executable operation.
2. the inspection level (low, medium, or high).
3. the tool currently in use.
4. the feed rate.
5. the file pointer to print to.

The function uses world model information regarding:

1. the current location.
2. the high retract location of the current setup.
3. the coordinate system currently in use.

The function uses options information regarding:

1. the clearance distance to use.
2. the number of points to inspect for a DMIS plane at a given inspection level.

The procedure (with retractions and resettings of search and approach distances omitted) is:

1. Write lines of DMIS code to define and measure the face of the block on the XZ-plane.
2. Write lines of DMIS code to construct a line on the actual XZ-plane parallel to the XY-plane, and to rotate the coordinate system to a new system (named ROT2) which has the same origin as the previous coordinate system, but has its X-axis parallel to the just-measured XZ-plane.

---

1. The odd-looking “block_block” in the middle of this name means the bottom face of the block is used to determine the XY plane, and other faces of the block (rather than features of the block) are used to complete the process of locating the part.

3. Write lines of DMIS code to define and measure the face of the block on the XZ-plane in ROT2.
4. Write lines of DMIS code to translate ROT2 parallel to the Y-axis of ROT2 to a new coordinate system (named TRANS2) so that the origin of the actual XZ-plane lies on the X-axis of TRANS2.
5. Write lines of DMIS code to define and measure the face of the block on the YZ-plane.
6. Write lines of DMIS code to translate TRANS2 parallel to the X-axis of TRANS2 to a new coordinate system (named TRANS3) so that the origin of the actual YZ-plane coincides with the origin of TRANS3.
7. Set the name of the current coordinate system saved in the Task Planner world model to “TRANS3”, which is the name of the precisely located system.

During probe point selection, expected error in probe point location is considered in determining if candidate probe points are sure to be on the part. The expected error has two components: position error and shape error. The expected error decreases as this function proceeds and position error becomes very small. As the expected error gets smaller, the search and approach distances for probing are reset and the patches used by the Modeler for checking that candidate probe points are on the part shrink. The methods for handling error at each stage are interesting but are not discussed in this document.

9.7 Command Messages to the Fbics_Task2 Process

Command messages are sent to the Fbics_Task2 process, using NML communications, by the Fbics_Task process. The following command messages for the Fbics_Task2 process are defined.

All messages have an integer-valued `sequence_number` field. This field is a sequence number as described in Section 5.4.4.

9.7.1 TASK2_EXEC_DMIS_MSG

The `TASK2_EXEC_DMIS_MSG [int sequence_number, char dmis_file_name[80]]` has one field in addition to the `sequence_number`.

1. `dmis_file_name` — the name of the DMIS code file to execute.

The effect of carrying out this message is that the DMIS code file of the given `dmis_file_name` is executed.

9.7.2 TASK2_EXEC_NC_MSG

The `TASK2_EXEC_NC_MSG [int sequence_number, char nc_file_name[80]]` has one field in addition to the `sequence_number`.

1. `nc_file_name` — the name of the NC code file to execute.

The effect of carrying out this message is that the NC code file of the given `nc_file_name` is executed.

9.7.3 TASK2_EXIT_MSG

The `TASK2_EXIT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out this message is that the Fbics_Task2 process exits.

9.7.4 TASK2_INIT_MSG

The `TASK2_INIT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out this message is that the Fbics_Task2 process initializes itself. This includes initializing the DMIS and RS274/NGC interpreters.

9.8 Fbics_Task2 Process

The Fbics_Task2 process has no user interface. It is used by the Fbics_Task process, which sends it NML messages (see Section 9.7). The functions that carry out the messages call the DMIS interpreter and RS274/NGC interpreter API functions described below.

9.8.1 DMIS Interpreter API

This section describes the API for telling the DMIS interpreter in Fbics_Task2 what to do. This interface differs from the most recent previously documented version [Kramer17]. The DMIS interpreter has additional interfaces. They are as documented in [Kramer17].

The DMIS Interpreter API includes six interface functions. To turn the DMIS Interpreter on, call *`interp_init`*; other interface functions (except *`interp_exit`*) will return ERROR if they are called before *`interp_init`*. To use the DMIS Interpreter during a setup, first call *`interp_open_program`*. Call *`interp_read_section`* to read a file of DMIS code after opening a program and thereafter whenever the return value from *`interp_execute_next`* indicates that the file is finished executing but the end of the program has not been reached. Call *`interp_execute_next`* repeatedly after reading a file, until either the program ends or the file is completely executed. Call *`interp_close_program`* when the return value from *`interp_execute_next`* indicates the program is ended. To turn the DMIS Interpreter off, call *`interp_exit`*.

All the interface functions may return OK or ERROR. Only *`interp_execute_next`* returns anything else.

9.8.1.1 interp_execute_next

The *`interp_execute_next()`* function takes no arguments. It executes one DMIS statement. It returns EMPTY when necessary to read a file again, and returns EXIT at the end of a program.

9.8.1.2 interp_exit

The *`interp_exit()`* function takes no arguments. It shuts the DMIS Interpreter down, but the Interpreter can be re-initialized by a call to *`interp_init`*.

9.8.1.3 interp_init

The *`interp_init()`* function takes no arguments. The general meaning of *`interp_init`* is: Get ready to run. The DMIS Interpreter world model is initialized.

9.8.1.4 interp_open_program

The *interp_open_program()* function takes no arguments. It gets the interpreter ready for a new program.

9.8.1.5 interp_read_section

The *interp_read_section(char * file_name)* function takes one argument, the name of a DMIS file to read. The function reads a program file into active memory. The first file read after a call to *interp_open_program* must be a valid beginning to a DMIS program. Each other file must end at the end of a complete syntactical structure. In particular, a line cannot be split with a line continuation character ($) at the end of a file, and a MEAS or GOTARG block that starts in a file must end in the same file.

9.8.1.6 interp_close_program

The *interp_close_program()* function takes no arguments. It closes the currently open program. It also clears memory of all data associated with that program and resets a few things in the interpreter world model.

9.8.2 RS274/NGC Interpreter API

This section describes the API for telling the RS274/NGC interpreter in Fbics_Task2 what to do. This interface differs from that described in the most recent previously documented version [Kramer13] but is what is currently used in the EMC project. Several functions in the interface behave quite differently from their counterparts (judging by name) among the DMIS Interpreter interface functions.

The interface functions, since they were devised for other uses, do not help keep track of whether a program is open, and they do not return a special value when the end of a file is reached. The application must keep track of programs and file sizes on its own steam.

The RS274/NGC Interpreter API used by FBICS includes six interface functions (others are available but are not used). To turn the Interpreter on, call *rs274ngc_init*; other interface functions (except *rs274ngc_exit*) will return ERROR if they are called before *rs274ngc_init*. To open an RS274/NGC file (not program) call *rs274ngc_open*. Once a file is open, call *rs274ngc_read* followed by *rs274ngc_execute* for each line of the file. Call *rs274ngc_close* when all the lines of the file have been interpreted. The return value from *rs274ngc_execute* indicates when the program is ended. To turn the Interpreter off, call *rs274ngc_exit*.

All the interface functions may return RS274NGC_OK or RS274NGC_ERROR. Only *rs274ngc_execute* returns anything else.

9.8.2.1 rs274ngc_close

The *rs274ngc_close()* function takes no arguments. It closes the currently open NC code file.

9.8.2.2 rs274ngc_execute

The *rs274ngc_execute(const char * mdi=0)* function takes one argument, a string.

In other uses, the string is a line of NC code, but in FBICS it should always be null. When the argument is null, the last file line read (which has been stored by the interpreter) is executed.

9.8.2.3 rs274ngc_exit

The `rs274ngc_exit()` function takes no arguments. This function shuts the Interpreter down.

9.8.2.4 rs274ngc_init

The `rs274ngc_init()` function takes no arguments. This function gets the Interpreter ready to interpret NC code files.

9.8.2.5 rs274ngc_open

The `rs274ngc_open(const char * filename)` function takes one argument, a string giving the name of the NC code file to open. The file is opened for reading.

9.8.2.6 rs274ngc_read

The `rs274ngc_read()` function takes no arguments. When the function is called, the next line of the currently open file is read, parsed, and stored (but not executed).

9.9 Task Planner Options

The Task Planner uses both task options (options used only by the Task Planner, see Section 12.1.7) and shop options (options used by two or more planners, see Section 12.1.6). The Fbics_Task process, but not the Fbics_Task2 process, reads these when it initializes or re-initializes and saves them for reference in its world model.

9.9.1 Task Options

Task options include the user's preferences for machining and inspection regarding such things as retract height, tool change position, deep hole drilling, etc.

9.9.2 Shop Options

From the shop options, the Task Planner uses the tool catalog file name and the tool inventory file name.

10 Modeler

The Solid Modeling Server (Modeler) provides solid modeling services to its clients, which are the Cell, Work, and Task controllers. The Parasolid solid modeler serves as the underlying modeling engine.

The Modeler maintains a separate view of the shape of things for each client. This is done in the FBICS software, not by using Parasolid partitions.

The types of service the Modeler provides include, for example, maintaining a model of the current shape of the part, faceting a model and telling the Graphic Display to show it, and determining if a candidate touch point for probing is present on the current part.

There is no user interface to the Modeler, except for responding to time-outs, as described in Section 5.6.2.

10.1 Command and Status Messages of the Modeler

Command messages are sent to the Modeler, using NML communications, by the Cell, Work, and Task planners. Three command messages to the Modeler are defined. One type of status message sent back from the Modeler is defined.

All messages have an integer-valued `sequence_number` field. This field is a sequence number as described in Section 5.4.4.

10.1.1 MODEL_ATTACH_MSG

The `MODEL_ATTACH_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out this message is to attach the sender to the Modeler. If no clients are attached when this message is received, the Modeler re-initializes. If Parasolid is not running, it is started.

10.1.2 MODEL_DETACH_MSG

The `MODEL_DETACH_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out this message is to detach the sender from the Modeler. The Modeler shuts Parasolid down to free up memory when all the Planners are detached.

10.1.3 MODEL_FUNCTION_MSG

The
```
MODEL_FUNCTION_MSG [int sequence_number,
  int function_id, char string1[100], int i1, int i2, int i3,
  double d1, double d2, double d3, double d4, double d5, double d6,
   double d7, double d8, double d9, double d10, double d11, double d12]
```
has seventeen fields in addition to the `sequence_number`.

1. `function_id` — an integer code giving a function_id.
2. `string1`— a multipurpose string.

3. i1,i2,i3— three multipurpose integers.
4. d1, d2, ..., d12 — twelve multipurpose doubles.

There are 26 subtypes of MODEL_FUNCTION_MSG, as described in Section 10.2. Each subtype has a function_id, which is an integer from 1 to 26. The function_ids are defined in the model_functions.hh file. It would be straightforward to define a separate message for each subtype, but having a single message with subtypes was easier to implement.

The MODEL_FUNCTION_MSG is generic message that includes enough additional fields that arguments needed by all the subtypes of model function will fit into them. Each subtype uses some subset of the fields and ignores the rest of the fields.

The meaning of the arguments and the effect of carrying out a MODEL_FUNCTION_MSG varies according to the function_id, as described in Section 10.2.

10.1.4 MODEL_READY_MSG

The MODEL_READY_MSG [int sequence_number, int status,
  int i1, int i2, int i3, int i4, int i5, int i6,
  double d1, double d2, double d3, double d4, double d5, double d6]
is the status message from the Modeler. It has 13 fields in addition to the sequence_number.

1. status — an integer code giving the status.
2. i1, i2, ..., i6 — six multipurpose integers.
3. d1, d2, ..., d6 — six multipurpose doubles.

The six integer fields and six double fields of the MODEL_READY_MSG are for returned values. Their meaning varies by message subtype, as described in Section 10.2.

10.2 Model Function Subtypes

The subtypes of model function are described below, along with the parameters received from the fields of the MODEL_FUNCTION_MSG and the values returned in the MODEL_READY_MSG. In addition to the values listed below, every MODEL_READY_MSG has fields for the echoed command sequence number and the status of execution. If the value returned in the status of execution field is not OK, any other return values should be assumed to be invalid.

The function names used in model_functions.hh are the 26 names given below, but each with the prefix “pm_” and the suffix “_type”. In the descriptions below, the arguments are the arguments of a MODEL_FUNCTION_MSG.

Where a subtype description refers to drawing a body, this means the Modeler will facet the body, write a graphics file for one of the six types of object (part_in, part_out, part_now, feature, access volume, or fixture), and send a message to the Graphic Display telling it to read that type of file and make it available for display.

This set of model function subtypes may be thought of as an API for the Modeler. As such, it is a relatively high-level API, where one call may do a lot of work. This API is designed only for doing the modeling work that needs to be done for specific tasks performed by the FBICS controllers. It is far from being a general-purpose high-level API from controllers to modelers, but it might serve as the starting point for defining such an API. A real API would use meaningful names for the arguments, rather than the very anonymous ones given below.

10.2.1 block_intersects_part

Block_intersects_part determines if a block (of a specified size in a specified place) intersects the part_now. The tag of the part_now is not given. The Modeler is assumed to know it.

RECEIVE

d1	x-length of the block
d2	y-length of the block
d3	z-length of the block
d4	x-coordinate of middle of block bottom
d5	y-coordinate of middle of block bottom
d6	z-coordinate of middle of block bottom
d7	x-component of block Z-axis
d8	y-component of block Z-axis
d9	z-component of block Z-axis
d10	x-component of block X-axis
d11	y-component of block X-axis
d12	z-component of block X-axis

RETURN

i1	0 for no intersection, 1 for intersection

USED BY: Task Planner

10.2.2 bodies_intersect

Bodies_intersect determines if two existing bodies intersect, and if so, what the tags of the first five (or fewer) bodies in the intersection are.

RECEIVE

i1	tag of the first body to intersect
i2	tag of the second body to intersect

RETURN

i1	number of bodies in the intersection, which may be 0
i2	tag of the first body in the intersection if i1 > 0, else 0
i3	tag of the second body in the intersection if i1 > 1, else 0
i4	tag of the third body in the intersection if i1 > 2, else 0
i5	tag of the fourth body in the intersection if i1 > 3, else 0
i6	tag of the fifth body in the intersection if i1 > 4, else 0

USED BY: Cell Planner and Work Planner

10.2.3 circle_on_part

Circle_on_part determines if a given circle lies entirely on the part_now.

RECEIVE
- d1 x-coordinate of center of circle
- d2 y-coordinate of center of circle
- d3 z-coordinate of center of circle
- d4 x-component of normal to circle
- d5 y-component of normal to circle
- d6 z-component of normal to circle
- d7 radius of circle

RETURN
- i1 1 if the circle lies entirely on the part_now, 0 if not

USED BY: Task Planner

10.2.4 contains

Contains determines if one body is contained in another.

RECEIVE
- i1 tag of the possibly containing body
- i2 tag of the possibly contained body

RETURN
- i1 is 1 if the first body contains the second, 0 otherwise

USED BY: Cell Planner and Work Planner

10.2.5 copy_entity

Copy_entity copies an entity.

RECEIVE
- i1 tag of the entity to copy
- i2 code indicating whether to save the copy in the client model, as follows:
  0 = do not save as part_now model,
  1 = save as part_now model,
  other values give error.

RETURN
- i1 tag of the copy

USED BY: Cell Planner, Work Planner, and Task Planner

10.2.6 c_shell_intersects_part

C_shell_intersects_part determines if a partial or full cylindrical shell intersects the part_now. The tag of the part_now is not given. The Modeler is assumed to know it. A partial shell has the shape made by sweeping a circular arc, constrained to be not larger than a semicircle. The thickness of the shell is 0.00001. The shell is not made by a sweep, however. It is made by first subtracting an inner cylinder from an outer cylinder and then (if not a whole shell) subtracting a block.

RECEIVE

d1	radius of shell halfway from inside to outside
d2	height of shell
d3	angle of arc, (2 * M_PI) for whole shell, else <= M_PI
d4	x-coordinate of base of cylinders
d5	y-coordinate of base of cylinders
d6	z-coordinate of base of cylinders
d7	x-component of unit vector parallel to cylinder axis
d8	y-component of unit vector parallel to cylinder axis
d9	z-component of unit vector parallel to cylinder axis
d10	x-component of unit vector from center of base to arc start
d11	y-component of unit vector from center of base to arc start
d12	z-component of unit vector from center of base to arc start

RETURN

i1	0 for no intersection, 1 for intersection

USED BY: Task Planner

10.2.7 find_box

Find_box finds diagonally opposite corners of the smallest box around a body, where the sides of the box are parallel to the principal planes.

RECEIVE

i1	tag of body

RETURN

d1	box minimum x
d2	box minimum y
d3	box minimum z
d4	box maximum x
d5	box maximum y
d6	box maximum z

USED BY: Cell Planner

10.2.8 find_clear_length

Find_clear_length finds, for a given body, a length such that if a line that long has one end anywhere inside the body, the other end will be outside the body. The length is set to 1.01 times the length of the diagonal of a box around the body.

RECEIVE
- i1 tag of the body

RETURN
- d1 the length

USED BY: Cell Planner and Work Planner

10.2.9 find_tag

Find_tag, given the usage name of a feature, finds the tag of the feature.

For this function to work, each AP 224 feature must have a usage name that is unique among all the features currently of interest. Usage names originate in the design of the part_out, so having unique usage names is a constraint on the design. When a feature is assigned to a setup by being included in the features file for the setup, it keeps the usage name it had in the part_out design. Usage names of features of the part_in design are never used, so it does not matter what they are.

RECEIVE
- string1 usage_name of the feature

RETURN
- i1 tag of the body representing the feature

USED BY: Cell Planner and Work Planner

10.2.10 make_access

Make_access, given the tag of a feature, a point that should be on a face of the feature, and a sweep length and direction, makes a body which is the face swept in the given direction for the given distance. The new body is a volume through which access to the feature is made.

RECEIVE
- i1 tag of feature
- d1 length of sweep
- d2 x-coordinate of point on feature face to sweep
- d3 y-coordinate of point on feature face to sweep
- d4 z-coordinate of point on feature face to sweep
- d5 x-component of direction in which to sweep face
- d6 y-component of direction in which to sweep face
- d7 z-component of direction in which to sweep face

RETURN
- i1 tag of the access volume

USED BY: Cell Planner and Work Planner

10.2.11 make_block

Make_block makes a block of the given size with its sides parallel to the principal planes and the center of its base at the origin.

RECEIVE
- d1 x-size
- d2 y-size
- d3 z-size

RETURN
- i1 tag of body representing block

USED BY: Cell Planner

10.2.12 model_fixture

Model_fixture makes a body in a specified place representing the fixture. The fixture file name is the name of a STEP AP 224 file representing half of a vise. The function does the following.

1. The fixture file is read and a working form is created. The Part is found in the working form. The Part should be half of a vise.
2. A solid model of the vise-half is built.
3. The vise-half is moved to the location given by the arguments.
4. A copy of the vise-half is made, mirrored on the XZ plane, and translated in the Y-direction by the given amount.
5. Two blocks are made as crossbars.
6. The vise-halves and crossbars are united to make a single solid vise.
7. The completed vise is facetted, and a message is sent to fbics_draw to draw it.
8. The working form is deleted.
9. The tag of the completed vise is recorded.

RECEIVE
- string1 name of the AP 224 fixture file
- d1 x-coordinate of fixture position origin
- d2 y-coordinate of fixture position origin
- d3 z-coordinate of fixture position origin
- d4 x-component of fixture position Z-axis
- d5 y-component of fixture position Z-axis
- d6 z-component of fixture position Z-axis
- d7 x-component of fixture position X-axis
- d8 y-component of fixture position X-axis
- d9 z-component of fixture position X-axis
- d10 length of vise opening in the Y-direction

RETURN
- i1 tag of fixture model

USED BY: Work Planner

10.2.13 model_part_features

Model_part_features makes bodies representing all the features of a part defined in a STEP Part 21 AP 224 file. Optionally, this will also build an index of usage_name/tag pairs for the features. The file must define a single part which has manufacturing features. The part itself is not modeled, but provides a list of the features. When the name/tag index has been built, the tag of a model of a feature whose usage_name is known may be retrieved using the *find_tag* model function (see Section 10.2.9).

RECEIVE

string1	name of STEP Part 21 AP 224 file
i1	flag: non-zero means record usage_name/tag pairs
d1	x-coordinate of part position origin
d2	y-coordinate of part position origin
d3	z-coordinate of part position origin
d4	x-component of part position Z-axis
d5	y-component of part position Z-axis
d6	z-component of part position Z-axis
d7	x-component of part position X-axis
d8	y-component of part position X-axis
d9	z-component of part position X-axis

RETURN

i1	0
i2	number of features in features part model

USED BY: Work Planner

10.2.14 model_part_in

Model_part_in makes a body representing the part_in and a body for each feature of the part_in. Optionally, this will also build an index of usage_name/tag pairs for the features of the body.

RECEIVE

string1	name of STEP Part 21 AP 224 part model file
i1	flag: non-zero means record usage_name/tag pairs
d1	x-coordinate of part position origin
d2	y-coordinate of part position origin
d3	z-coordinate of part position origin
d4	x-component of part position Z-axis
d5	y-component of part position Z-axis
d6	z-component of part position Z-axis
d7	x-component of part position X-axis
d8	y-component of part position X-axis
d9	z-component of part position X-axis

RETURN

i1	tag of part_in model
i2	number of features in part_in model

USED BY: Cell Planner, Work Planner, and Task Planner

10.2.15 model_part_now

Model_part_now makes a body representing the part_now and a body for each feature of the part_now. Optionally, this will also build an index of usage_name/tag pairs for the features of the body.

RECEIVE

string1	name of STEP Part 21 AP 224 part model file
i1	flag: non-zero means record usage_name/tag pairs
d1	x-coordinate of part position origin
d2	y-coordinate of part position origin
d3	z-coordinate of part position origin
d4	x-component of part position Z-axis
d5	y-component of part position Z-axis
d6	z-component of part position Z-axis
d7	x-component of part position X-axis
d8	y-component of part position X-axis
d9	z-component of part position X-axis

RETURN

i1	tag of part_out model
i2	if i1 flag is non-zero, number of features in part_out model, otherwise 0

USED BY: Work Planner, and Task Planner

10.2.16 model_part_out

Model_part_out makes a body representing the part_out and a body for each feature of the part_out. Optionally, this will also build an index of usage_name/tag pairs for the features of the body.

RECEIVE

string1	name of STEP Part 21 AP 224 part model file
i1	flag: non-zero means record usage_name/tag pairs
d1	x-coordinate of part position origin
d2	y-coordinate of part position origin
d3	z-coordinate of part position origin
d4	x-component of part position Z-axis
d5	y-component of part position Z-axis
d6	z-component of part position Z-axis
d7	x-component of part position X-axis
d8	y-component of part position X-axis
d9	z-component of part position X-axis

RETURN

i1	tag of part_in model
i2	number of features in part_in model

USED BY: Cell Planner, Work Planner, and Task Planner

10.2.17 modify_part_now_file

Modify_part_now_file updates the part_now by subtracting a feature from it. The feature is described in a file which must define exactly one manufacturing feature. The tag of the part_now is not given. The Modeler is assumed to know it. Optionally, this will also draw the feature and/or the part_now.

RECEIVE
- string1 name of STEP Part 21 AP 224 feature file
- i1 not used
- i2 whether to draw feature: non-zero = draw it, 0 = do not draw
- i3 whether to draw part_now: non-zero = draw it, 0 = do not draw

RETURN
- no values returned

USED BY: Work Planner and Task Planner

10.2.18 modify_part_now_tag

Modify_part_now_tag updates the part_now by subtracting a feature from it, given the tag of a body representing the feature. The tag of the part_now is not given. The Modeler is assumed to know it. Optionally, this will also draw the feature and/or the part_now.

RECEIVE
- i1 tag of body representing feature
- i2 whether to draw feature: non-zero = draw it, 0 = do not draw
- i3 whether to draw part_now: non-zero = draw it, 0 = do not draw

RETURN
- no values returned

USED BY: Cell Planner and Work Planner

10.2.19 point_on_part

Point_on_part determines if a given point is on the surface of the part_now. The tag of the part_now is not given. The Modeler is assumed to know it.

RECEIVE
- d1 x-coordinate of point
- d2 y-coordinate of point
- d3 z-coordinate of point

RETURN
- i1 0 = point not on part_now, 1 = point on part_now

USED BY: Task Planner

10.2.20 rectangle_on_part

Rectangle_on_part determines if a given rectangle lies entirely on the part_now.

RECEIVE
- d1 x-coordinate of center of rectangle
- d2 y-coordinate of center of rectangle
- d3 z-coordinate of center of rectangle
- d4 x-component of rectangle normal pointing at part
- d5 y-component of rectangle normal pointing at part
- d6 z-component of rectangle normal pointing at part
- d7 length of rectangle; parallel to XY plane
- d8 width of rectangle; parallel to Z-axis

RETURN
- i1 1 if the rectangle lies entirely on the part_now, 0 if not

USED BY: Task Planner

10.2.21 relocate_body

Relocate_body relocates the given body to the given place. The tag of the body remains the same.

RECEIVE
- i1 tag of the body to be relocated
- d1 x-coordinate of placement origin
- d2 y-coordinate of placement origin
- d3 z-coordinate of placement origin
- d4 x-component of placement Z-axis
- d5 y-component of placement Z-axis
- d6 z-component of placement Z-axis
- d7 x-component of placement X-axis
- d8 y-component of placement X-axis
- d9 z-component of placement X-axis

RETURN
- no values returned

USED BY: Cell Planner

10.2.22 show_access

Show_access draws the given access volume.

RECEIVE
- i1 tag of access volume to show

RETURN
- no values returned

USED BY: Cell Planner and Work Planner

10.2.23 show_part_now

Show_part_now draws the part_now. The tag of the part_now is not given. The Modeler is assumed to know it.

RECEIVE
  no values received
RETURN
  no values returned

USED BY: Cell Planner, Work Planner, and Task Planner

10.2.24 show_volume_file

Show_volume_file draws the feature described in the file.

RECEIVE
  string1 name of file describing feature
  i1 volume-type: 0 = machining, 1 = inspection
RETURN
  no values returned

USED BY: Task Planner

10.2.25 show_volume_tag

Show_volume_tag draws the feature with the given tag.

RECEIVE
  i1 tag of feature to show
  i2 volume-type: 0 = machining, 1 = inspection
RETURN
  no values returned

USED BY: Cell Planner and Work Planner

10.2.26 unite_bodies

Unite_bodies unites two bodies by finding a single body which is their boolean sum. It is an error if the two bodies to be united do not intersect, since then the boolean sum is not a single body.

RECEIVE
  i1 tag of first body to unite
  i2 tag of second body to unite
RETURN
  i1 tag of united body

USED BY: Cell Planner and Work Planner

11 Graphic Display

11.1 Command Messages to the Graphic Display

Seven command messages to be sent to the Graphic Display (by the Modeler) are defined, as follows. All messages have an integer-valued `sequence_number` field. This field is a sequence number as described in Section 5.4.4.

If the Graphic Display is in AUTO update mode, it carries out each message as soon as the message is received. If the Graphic Display is in MANUAL update mode, it waits for the user to click on one of the graphics controls (the red bar that appears above MAN UPDATE — see Figure 17) before changing the display or returning a `DRAW_READY_MSG` to the Modeler, with the effect of stopping the Modeler and the controller that caused the Modeler to send a message to the Graphic Display.

Many controls on the Graphic Display are available to the user through its graphical user interface (see Section 11.2). The messages listed here provide commands that are not available through the user interface. The Graphic Display accepts one command message to clear the screen and six command messages to read and internalize files giving pictures of specific things. Once a picture file is internalized, the picture is available for display. Whether a particular picture is displayed at all and how it is displayed (wire frame or faces) depends on the current settings for that type of picture. When a file for a type of picture is read, data for the previous picture of that type is destroyed. The files read by the Graphic Display are generated by the Modeler.

11.1.1 DRAW_ACCESS_MSG

The `DRAW_ACCESS_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `DRAW_ACCESS_MSG` is to read and internalize the fbics_access_picture file.

11.1.2 DRAW_FIXTURE_MSG

The `DRAW_FIXTURE_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `DRAW_FIXTURE_MSG` is to read and internalize the fbics_fixture_picture file.

11.1.3 DRAW_FLUSH_MSG

The `DRAW_FLUSH_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `DRAW_FLUSH_MSG` is to clear the screen and delete all pictures from memory.

11.1.4 DRAW_PART_IN_MSG

The `DRAW_PART_IN_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `DRAW_PART_IN_MSG` is to read and internalize the fbics_part_in_picture file.

11.1.5 DRAW_PART_NOW_MSG

The `DRAW_PART_NOW_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `DRAW_PART_NOW_MSG` is to read and internalize the fbics_part_now_picture file.

11.1.6 DRAW_PART_OUT_MSG

The `DRAW_PART_OUT_MSG [int sequence_number]` has no fields other than the `sequence_number`.

The effect of carrying out a `DRAW_PART_OUT_MSG` is to read and internalize the fbics_part_out_picture file.

11.1.7 DRAW_VOLUME_MSG

The `DRAW_VOLUME_MSG [int sequence_number, int volume_type]` has one field in addition to the `sequence_number`.

1. `volume_type` — an integer code for the type of volume.

The effect of carrying out a `DRAW_VOLUME_MSG` is to read and internalize the fbics_volume_picture file. If the volume type is 0, that means the volume is a machining feature, and it is displayed in red when shown as a solid or mesh. If the volume type is 1, that means the volume is an inspection feature, and it is displayed in pink when shown as a solid or mesh.

11.2 Graphic Display User Interface

The Graphic Display provides a realistic picture of what FBICS is doing. The Graphic Display was developed for two purposes: (1) so developers can tell what FBICS is doing when it runs, in order to debug and improve automatic operation of FBICS, and (2) so people interested in FBICS can understand it readily. To serve these purposes, the user is provided with many controls over the view of what is happening.

The Graphic Display process displays one window to the user. This is a color graphics window divided into two parts: a scene and a control panel for manipulating the scene. The window may be manipulated in the typical ways (raise/lower, iconize/deiconize, move, resize). If the window is resized, it automatically keeps constant aspect ratio.

In order to tell what is happening during FBICS processing, it is useful to be able to see most or all of the objects in the scene shown in the window. The objects in the scene, however, tend to block the view of each other. Also, it may be difficult to tell which object is which, and the system may run too fast for the eye to keep up with it. To deal with these problems, each type of object is a different color, the visibility of each object may be controlled independently, the nature of the view of an object may be changed, and the system can be paused. The visibility and update controls discussed in the following sections enable the user to see what is happening in the system.

The user is provided with no Graphic Display controls that affect what FBICS is doing (other than being able to pause or stop it). In a commercial version of FBICS, it would be essential to provide graphics-based systems with friendly user interfaces to allow the user to participate effectively in the work of the system. The existing Graphic Display provides none of that.

The Graphic Display window shows a scene including seven types of object. Three of these change during processing, the other four do not.

1. part_in - brown, constant — the part before processing starts (shown if machining).
2. part_out - gold, constant — the part after processing ends (shown if machining).
3. part_now - green, sometimes changing — the part at its current stage of processing.
4. fixture - dark blue, constant — the fixture holding the part.
5. machining or inspection volume - red for machining or pink for inspection, changing — the feature currently being processed for machining or inspection.
6. access volume (of a feature) - cyan, changing — the volume which must be empty in order for a tool to reach the feature from the direction of the +Z-axis without interference.
7. coordinate axes - magenta, constant — the coordinate axes of the part (for cell-level planning) or setup (for work-level or task-level planning).

When processing includes machining, the three parts are all different, and the part_now changes frequently. When processing includes only inspection, only the part_now is shown, and it is unchanging.

The objects in the scene have fixed positions with respect to each other, as determined by the planning problem (for cell-level planning) or the setup file (for work-level or task-level planning).

The coordinate system shown is that of the part_now, part_in, and part_out (it being required that all three be the same).

11.2.1 What the Scene Shows

The scene shows what each planner is thinking about as planning proceeds in terms of parts, features, access volumes and fixtures. If machining planning is being performed, the part_now is updated periodically by each of the planners, reflecting the removal of features the planner has dealt with up to that point. Each planner empties the scene before starting to show what it shows.

11.2.1.1 Shown by Cell Planner

The Cell Planner is concerned with planning an entire part.

The Graphic Display is driven by the Cell Planner when the Planner is doing (i) stage-one planning for machining (possibly) with inspection, (ii) stage-one planning for pure inspection, (iii) stage-two planning for machining (possibly) with inspection, or (iv) execution of a stage-one plan for machining (possibly) with inspection.

The fixture is not shown during cell-level planning.

During stage-one planning for machining (possibly) with inspection, the following are shown in order — but nothing is shown relating to inspection (since no decisions about inspection are made during the planning process).

1. the part_in and part_out. These are shown unchanged as long as planning is in

progress.
2. the part_now. This is redrawn periodically.
3. any features of the part_out that do not intersect the part_in. These features do not need to be machined. They are shown briefly, one at a time.
4. for each feature that intersects the access volume of one or more other features, the feature and the access volumes. These are shown briefly, one pair at a time.
5. the set of features that can be made in any order (shown briefly, one at a time). These sets are grouped by setup (but there is no indication when one setup ends and another begins). After each set is shown, the part_now picture is updated.

During stage-one planning for pure inspection, the following are shown in order.

1. the part_now.
2. any features to be inspected. They are shown briefly, one at a time, grouped by setup (but there is no indication when one setup ends and another begins).

During stage-two planning for machining (possibly) with inspection and during execution of a stage-one plan for machining (possibly) with inspection, the following are shown.

1. the part_now as it appears at the end of each setup.

11.2.1.2 Shown by Work Planner

The Work Planner plans for a single setup.

The Graphic Display is driven by the Work Planner when the Planner is doing (i) stage-one planning for machining (possibly) with inspection, (ii) stage-one planning for pure inspection, (iii) stage-two planning for machining (possibly) with inspection, or (iv) execution of a stage-one plan for machining (possibly) with inspection.

During stage-one planning for machining (possibly) with inspection, the following are shown in order. Nothing is shown relating to inspection; this should be changed to show features planned for inspection.

1. the part_in and part_out. These are shown unchanged as long as planning is in progress.
2. the part_now. This is redrawn periodically.
3. for each feature that intersects the access volume of one or more other features, the feature and the access volumes. These are shown briefly, one pair at a time.
4. the set of features that can be made in any order (shown briefly, one at a time). After each set is shown, the part_now picture is updated.

During stage-one planning for pure inspection, the following are shown in order.

1. the part_now.
2. any features to be inspected. They are shown briefly, one at a time.

During execution of a stage-one plan for machining (possibly) with inspection, the part_in, part_out, part_now, and fixture are shown. As the system currently works, however, the three parts are promptly overwritten by the Task Planner.

During execution of a stage-one plan for pure inspection, the part_now and fixture are shown. As the system currently works, however, the part is promptly overwritten by the Task Planner.

11.2.1.3 Shown by Task Planner

The Task Planner plans for single features but does so in the context of a setup.

When a setup is opened for NC code generation and (possibly) DMIS code generation, the part_in, part_out, and part_now are shown. Whenever NC code is generated to make a feature, the feature is shown, and the part_now is shown after being modified by removing the feature. Whenever DMIS code is generated to inspect a feature, the feature is shown.

When a setup is opened for DMIS code generation only, the part_now is shown. Whenever DMIS code is generated to inspect a feature, the feature is shown.

11.2.1.4 A Current Shortcoming

Objects in the scene may be updated by the Cell Planner, the Work Planner, and the Task Planner. The user may not know which planner's view is being shown of which object. This difficulty is exacerbated by the fact that updates by the Work and Task planners are sometimes interleaved. It would be useful (1) to add an indicator telling which planner's view is being shown, and/or (2) to add a control so the user can select the planner whose view is shown. The Modeler already maintains separate models for each of the three planners.

11.2.2 Overview of User Interface Controls

This section is provided for current and potential FBICS users so the interface can be rapidly understood and used. The interface is intended to be easy to learn and use for experienced CAD users. There are no significant innovations in the Graphic Display user interface.

The control panel of the user interface for the Graphic Display is shown in Figure 17. This control panel appears on the right side of the graphics window as shown in Figure 3. The scene fills the rest of the window. Each of the boxes on the panel is a mouse-sensitive button. The lightly shaded boxes in the top half of the figure are usually colored green. Almost all of these become yellow briefly when pushed. The darker shaded boxes in the bottom half of the figure are brown when off and yellow when on.

The seven control panel buttons at the top are each divided into four parts, which are areas of sensitivity. The leftmost part is least sensitive, rightmost is most sensitive. These control panel buttons react differently to clicks from a 3-button mouse. The left mouse button generally means smaller, down, or left. The right (or middle) mouse button generally means larger, right, or up. As an example, if the left mouse button is pressed while the cursor is in the far left quarter of the ZOOM control panel button, the picture gets a little smaller; if the right mouse button is pressed while the cursor is in the far right quarter, the picture gets much bigger.

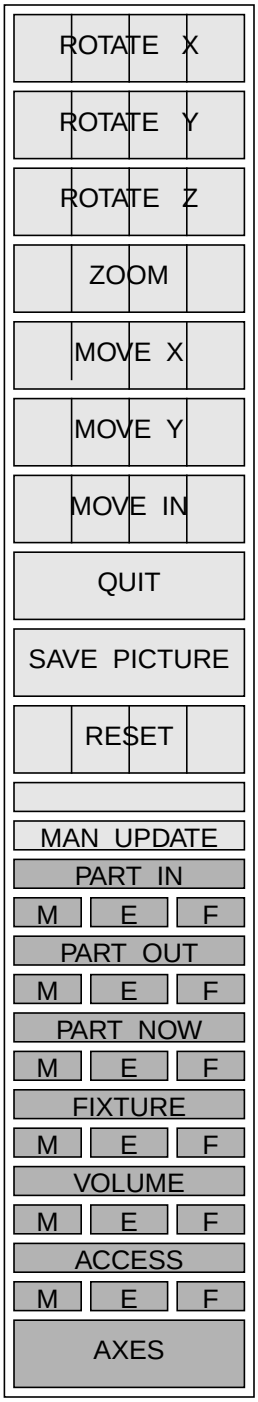


Figure 17. Graphic Display Controls

11.2.3 Scene View Control

The scene is rotated using the three buttons labelled ROTATE. Each of these buttons rotates the scene about an axis through the midpoint of the object parallel to the axis named in the button label. The axes are shown on the picture when the AXES button is yellow.

The MOVE X and MOVE Y control panel buttons move the scene horizontally (X) and vertically (Y) on the screen.

The ZOOM and MOVE IN control panel buttons both make the scene appear larger or smaller, but they differ in how they do it. ZOOM affects the size of the picture but not the shape. MOVE IN changes the size and shape of the picture because it behaves as though it is moving the user's eye closer or farther from the object. Very close to the scene (relative to its size), the effect of perspective is significant. Farther from the scene there is little perspective effect. By alternately moving away and zooming in, the picture approaches being a parallel projection (with no perspective effects).

The RESET control panel button resets the view of the scene to one of four preset views. The leftmost quarter of the button shows a top view. The middle left quarter shows a front view. The middle right quarter shows a right view, and the rightmost quarter shows an isometric view. All three mouse buttons have the same effect on the RESET button.

Some aspects of scene view are handled automatically. The most important of these is that the picture is automatically sized so that in any of the preset views the entire part_in is visible on the screen. Also, the viewpoint for all of the preset views is located so the eye seems to be moderately

far from the scene.

11.2.4 Object Visibility Control

The part_in, part_out, part_now, fixture, feature volume, and access volume each have three kinds of visibility: faces, edges, and mesh, denoted by F, E, and M, respectively on the control panel. Each of these objects is facetted for viewing. The faceting defines planar polygons. The F button toggles the visibility of the faces of these polygons. The M button toggles the visibility of the edges (mesh) of the polygons. The E button toggles the visibility of the geometric edges of the object itself, independent of any faceting. The three buttons work independently.

The object name buttons (just above each triple of M E F buttons) are also active. If any of the three M E F buttons is yellow, the corresponding name button is also yellow. If the name button is pushed when it is yellow (using any of the three mouse buttons), it turns off all visibility for that object, and all four related buttons go dark. An object name button has no effect if it is pushed when it is dark.

Coordinate axes may be turned on or off with the AXES control panel button. They do not exist until the part_out is read in, so they are not visible until then, regardless of the setting of the AXES button.

11.2.5 Other Graphic Display Controls

The SAVE PICTURE control panel button, when pushed, saves a Postscript file containing the contents of the graphics window (scene and control panel). The SAVE PICTURE button turns yellow while the picture is being saved, then turns green again. The name of the Postscript file is determined by an environment variable, HOOPS_HARDCOPY.

The MAN UPDATE button indicates that updates to the appearance of the picture are under user control. If the MAN UPDATE is visible, whenever the Graphic Display receives a message, the unlabeled button above the MAN UPDATE button turns red. The red button must be pushed to make the Graphic Display process the message. When the red button is pushed, it turns yellow briefly while the update is taking place; then it turns green again. If the MAN UPDATE button is pushed, it becomes an AUTO UPDATE button, and messages are processed as soon as they are received. If the AUTO UPDATE button is pushed, it becomes a MAN UPDATE button again.

The QUIT button shuts down the Graphic Display, and its window disappears. The QUIT button works as soon as it is pushed and does not ask for confirmation.

12 Data Types

Data files are used in FBICS for input to the system (part designs, for example), for persistent system data (tool catalog, for example), and for passing information across interfaces between modules (a machining operation, for example). All data files are kept in the Data Repository.

This section is arranged by data format. Each format is used for one or more types of data. FBICS uses many STEP Part 21 formats, two dialects of RS274 NC code for machining, DMIS code for inspection, and a simple unnamed graphics file format.

Files containing data of almost all types used in FBICS are shown in Appendix A and Appendix B.

12.1 STEP Part 21 Formats

12.1.1 Introduction

The format of most FBICS files is the STEP Part 21 exchange file format. The Part 21 specification is generic, in that it requires an EXPRESS information model to be plugged into it in order to make a complete file format specification. To know what the format of a file is, one must know both that it is a STEP Part 21 file and which EXPRESS model it uses. This differs from many other non-STEP file formats, for which the specification combines information model and file format in a single document.

Table 1 shows the STEP Part 21 file types used in FBICS. With each file type (identified by EXPRESS model name) are given the items modeled using the file type, the writers of the items, the readers of the items, and the names of the EXPRESS schemas that define the model. In the table, “all 3 planners” means the Cell, Work, and Task planners. Where the “EXPRESS schemas” column lists more than one schema, the uppermost schema relies on the lower schemas as described in Section 12.1.2.

Table 1. FBICS Data in STEP Part 21 Files

Model Name	Item Modeled	File Writers	File Readers	EXPRESS schemas
AP 224	initial workpiece	User Cell Planner	all 3 planners	arm224
	intermediate workpiece	Cell Planner	Work Planner Task Planner	
	final part	User or CAX system	all 3 planners	
	features	Cell Planner	Work Planner	
	fixture	User or CAX system	Work Planner Task Planner	
Cell-level Stage-one Plan	Cell-level stage-one plan	Cell Planner	Cell Planner	fbics_combo fbics_alps
Cell-level Stage-two Plan	Cell-level stage-two plan	Cell Planner	Cell Planner	fbics_combo
Setup	setup	Cell Planner	all 3 planners	setup
Shop Options	shop options	User	all 3 planners	shop_options
Task Options	task options	User	Task Planner	task_options
Work Options	work options	User	Work Planner	work_options
Tool Catalog	tool catalog	User	Work Planner Task Planner	tool_catalog
Tool Inventory	tool inventory	User	Work Planner Task Planner	tool_inventory tool_catalog
Tool_Usage_ Rules	tool usage rules	User	Work Planner	fbics_combo expressions
Work-level Stage-one Plan	Work-level stage-one plan	Work Planner	Work Planner	fbics_combo fbics_alps
Work-level Stage-two Plan	Work-level stage-two plan	Work Planner	Work Planner	fbics_combo
Work-level Executable Operations	Work-level executable operation	Work Planner	Task Planner	fbics_combo fbics_alps arm224

One of the formats used in FBICS (STEP AP 224) is an ISO international standard. One of them (the tool catalog) is in the process of being standardized.

Subsections 4.1.2 and 4.1.3, immediately following, cover STEP modeling topics. The remaining subsections cover specific EXPRESS data models.

12.1.2 Information Modeling In EXPRESS

The EXPRESS information modeling language [ISO1] provides the construct “schema” as the basic unit for a model. A single model is often contained in a single schema. EXPRESS schemas (including those used in FBICS) are usually composed largely of definitions of “types” and “entities”. EXPRESS types are similar to C or C++ types. EXPRESS entities are similar to C++ classes with data members only.

It is common, however, that two or more domains which are to be modeled share a coherent subdomain. Automobiles and ships, for example, both have geometry. In such cases, a model may be made for the subdomain, and the models for the domains will use the subdomain model via the EXPRESS “USE” statement. FBICS uses this modeling technique frequently, so that a single model is spread over more than one schema.

Unrelated models may be combined together in a single schema. This is bad modeling practice, in principle, but in practice, there may be no technical problems, and doing it saves schema processing overhead. In FBICS, five small models have been combined in a single “fbics_combo” schema. In the long run, it may be useful to split the fbics_combo into several schemas with some USEing others.

It may be that one domain is a proper subset of another. Two reasonable methods for handling such a case are (1) put the proper subset in one schema and the rest of larger domain in a separate schema which USEs the first or (2) put everything in one schema. In FBICS, the tool catalog model is a proper subset of the tool inventory model. This has been handled using the first method, with a separate schema for each.

Frequently, the processing of STEP data in a program will require the use of new related data types. For example, “tool_instance” is defined in the tool inventory, and it is useful to have a data type which is a list of tool_instances. The STEP Tools utility (see Section 12.1.3) provides a convenient method (working set files) for providing related data types. That method is used in FBICS.

12.1.3 Data Handling Tools

STEP Tools, Inc. is a commercial venture which provides tools for dealing with STEP methods, models, and data. STEP Tools provides (among other things) a set of utilities for doing useful things with EXPRESS and STEP Part 21 files [STEPTools1], and a library of functions that may be used in building a STEP-based application [STEPTools2]. Similar tools are available from other companies.

The STEP Tools express2c++ utility has been used extensively in building FBICS. This utility reads an EXPRESS schema (and any another schemas the original schema USEs) and produces C++ header files and code files. As part of code generation, any schema processed by express2c++ has its syntax checked. The syntax checker can be run separately, if that is desired.

STEP Tools also provides utilities for compiling the source code and building libraries from it that may be linked into an application. FBICS uses 4 such libraries:

1. libfour_schemas224.a (arm224, fbics_alps, fbics_combo, and expressions)
2. libsetup.a (setup)
3. liboptions.a (shop_options, work_options, and task_options)
4. libtool.a (tool_catalog and tool_inventory)

The schemas named in parentheses above correspond one-to-one with the following data formats (similar names match) except that fbics_combo contains five models: tool_usage_rules, cell-level tasks, work-level inspection operations, work-level machining operations, and work-level executable_operations (both inspection and machining).

As mentioned earlier, several of the schemas on which the libraries are based USE other schemas: fbics_alps USEs expression, fbics_combo USEs both arm224 and fbics_alps (and, thereby, expressions), and tool_inventory USEs tool_catalog.

12.1.4 STEP AP 224

STEP AP 224 and its uses in FBICS were discussed in Section 4.3.2 and elsewhere.

The EXPRESS schema used in FBICS for STEP AP 224 is a schema provided by Len Slovensky, owner of AP 224. The schema has been modified slightly. In STEP terms it is an Application Resource Model (ARM) type of model.

Sample AP 224 STEP Part 21 files for parts are shown in Appendix A.1, Appendix A.2, and Appendix A.3. Sample AP 224 STEP Part 21 files for features are shown in Appendix A.8 and Appendix A.9.

12.1.5 Options Introduction

There are three options models: shop options, work options, and task options. Each model is in a separate EXPRESS schema, but all three schemas are in a single file, options.exp. The work_options and task_options schemas USE the shop_options schema.

The name of every option ends in “_use”. For each option, an EXPRESS entity data type to be used with the option is defined whose name is the same as the option name, except without the “_use” ending. Most of these EXPRESS entities simply encapsulate a number, a boolean, or an enumeration. The modeling was done this way so that options files are human-readable, and it is feasible for a human to hand-edit options files. If the EXPRESS entities were not defined, it would be necessary to refer to the schemas to edit the files. Sample options files are shown in Appendix B.1, Appendix B.2, and Appendix B.3.

12.1.6 Shop Options

The shop options model gives option settings used in more than one of the FBICS planners. During FBICS operation, these options are expected to be set the same in each FBICS planner that uses them.

The shop options rest, in part, on the notions that some level of implicit tolerance should be held in the work done by a shop, and some level of tolerance is beyond the shop’s capability. The milling_tolerance_default and milling_tolerance_tightest are defined in the shop options to

represent those notions. No point on the surface of an outgoing workpiece made in the shop should be more than the milling_tolerance_default distance from the nominal surface. A “tight” tolerance for milling is a tolerance less than the milling_tolerance_default and is expected to require more than normal care to achieve. A part whose design has tolerances tighter than the shop’s milling_tolerance_tightest is not expected to be machinable in the shop.

Shop options STEP Part 21 files are used in FBICS for shop options. The shop options file “data/shop_options1.stp” is read by each of the three FBICS planners when it initializes or re-initializes. Selected values in the file are saved for reference in the planners’ world models as described in Section 7.4, Section 8.5, and Section 9.9. An example of a shop options file is given in Appendix B.1.

A brief description of each shop option follows.

12.1.6.1 Inspect_action_use

The inspect_action_use option indicates what to do if a feature is inspected and found out-of-spec. The choices are (a) abort, (b) try to repair the problem, or (c) ignore the problem.

12.1.6.2 Inspect_decision_use

The inspect_decision_use indicates how to make decisions about what features to inspect. The choices are (a) inspect all features, (b) inspect no features, (c) inspect a feature if any attribute has any tolerance, (d) inspect a feature if any attribute has a tight tolerance, or (e) let the user decide which features to inspect.

12.1.6.3 inspect_interval_use

The inspect_interval_use indicates how many features to queue up for inspection before switching from machining to inspection. This exists to provide control over switching between machining and inspection. The value of the option should be a positive integer.

No use is made of this option in FBICS, currently.

12.1.6.4 inspect_level_use

The inspect_level_use indicates how intensely to inspect. The choices are (a) high, (b) medium, (c) low.

12.1.6.5 length_unit_rule_use

The length_unit_rule_use indicates which length unit to use. The choices are (a) use_in, i.e., inches or (b) use_mm, i.e., millimeters. Handling of units is described in Section 3.8.

12.1.6.6 milling_tolerance_default_use

The milling_tolerance_default_use is a number representing the largest error in the position of a nominal point on the workpiece that may occur as a result of a milling operation. Units for milling_tolerance_default are those specified by the length_unit_rule_use.

12.1.6.7 milling_tolerance_tightest_use

The milling_tolerance_tightest_use is a number representing the smallest tolerance that can

reliably be achieved during milling. Units for milling_tolerance_tightest are those specified by the length_unit_rule_use.

12.1.6.8 tool_catalog_name_use

The tool_catalog_name_use gives the name of the tool catalog file to use.

12.1.6.9 tool_inventory_name_use

The tool_inventory_name_use gives the name of the tool inventory file to use.

12.1.6.10 tool_usage_name_use

The tool_usage_name_use gives the name of the file containing rules for setting tool use parameters.

12.1.7 Task Options

The 28 task options listed below cover the user's preferences regarding task-level machining and inspection activities. The task options file "data/task_options1.stp" is read by the Task Planner each time it executes the taskpl_init function. An example of a task options file is shown in Appendix B.2.

The first nine options apply to both machining and inspection. The next nine apply to inspection only. The last ten apply to machining only.

12.1.7.1 automatic_changer_use

The automatic_changer_use indicates whether or not the machine has an automatic tool changer. Its value is a boolean true or false.

12.1.7.2 change_location_use

The change_location_use indicates where to go to change the tool. The choices are: (a) stay at the current position, (b) stay at the current position in XY but retract in Z, (c) go to home1, or (d) go to home2.

12.1.7.3 end_location_use

The end_location_use indicates where to go at the end of a program. The choices are: (a) stay at the current position, (b) stay at the current position in XY but retracted in Z, (c) go to home1, or (d) go to home2.

12.1.7.4 home_one_use

The home_one_use gives the location of home1. Its value is a triple of real numbers.

12.1.7.5 home_two_use

The home_two_use gives the location of home2. Its value is a triple of real numbers.

12.1.7.6 length_units_use

The length_units_use option indicates (only! - see Section 3.8) what length units are used in the

rest of the options file (for locations, thicknesses, etc.). The choices are: (a) use_in or (b) use_mm.

12.1.7.7 max_tool_length_offset_use

The max_tool_length_offset_use gives the largest allowable tool length offset. Its value is a real number.

12.1.7.8 origin_use

The origin_use gives the location in machine coordinates of the point to use as the origin for writing programs. Its value a triple of real numbers.

12.1.7.9 z_up_value_use

The z_up_value_use gives the value of Z to go to when told to retract in Z. Its value is a real number.

12.1.7.10 inspect_clear_use

The inspect_clear_use gives the clearance distance of the probe tip during inspection operations. Its value is a real number. For example, this would be used as the distance between the walls of a pocket and the probe tip while inspecting the bottom of the pocket.

12.1.7.11 inspect_points_circle_use

The inspect_points_circle_use data gives the number of points to inspect on a DMIS circle at the high, medium, and low inspecting levels. Its value is three positive integers.

12.1.7.12 inspect_points_cone_use

The inspect_points_cone_use data gives the number of points to inspect on a DMIS cone at the high, medium, and low inspecting levels. Its value is three positive integers.

12.1.7.13 inspect_points_cylinder_use

The inspect_points_cylinder_use data gives the number of points to inspect on a DMIS cylinder at the high, medium, and low inspecting levels. Its value is three positive integers.

12.1.7.14 inspect_points_line_use

The inspect_points_line_use data gives the number of points to inspect on a DMIS line at the high, medium, and low inspecting levels. Its value is three positive integers.

12.1.7.15 inspect_points_plane_use

The inspect_points_plane_use data gives the number of points to inspect on a DMIS plane at the high, medium, and low inspecting levels. Its value is three positive integers.

12.1.7.16 inspect_points_sphere_use

The inspect_points_sphere_use data gives the number of points to inspect on a DMIS sphere at the high, medium, and low inspecting levels. Its value is three positive integers.

12.1.7.17 inspect_retract_distance_high_use

The inspect_retract_distance_high_use gives an incremental high distance to move above the part between inspecting features. Its value is a positive real number.

12.1.7.18 inspect_retract_distance_low_use

The inspect_retract_distance_low_use gives an incremental low distance to move above the part between inspecting features. Its value is a positive real number.

12.1.7.19 deep_drill_cycle_use

The deep_drill_cycle_use indicates what type of cycle to use for drilling a deep hole. The choices are (a) peck_drilling, (b) chip_breaking_drilling, or (c) plunge_drilling.

12.1.7.20 deep_hole_factor_use

The deep_hole_factor_use gives the depth to diameter ratio above which a hole is considered to be deep. Its value is a positive real number.

12.1.7.21 entry_strategy_use

The entry_strategy_use gives the method to use for making an entry cut. The choices are (a) plunge, (b) ramp, (c) side, (d) spiral, or (e) void. The “side” choice means to approach the feature from the side; this is a valid choice only if it is possible to approach from the side. The “void” choice means to approach from the top by entering a void in the feature; this is a valid choice only if there is a void in the feature.

12.1.7.22 finish_cut_thickness_use

The finish_cut_thickness_use gives the thickness of material to leave on surfaces during bulk removal cutting (rough cutting) which is expected to be removed by finish cutting. Its value is a positive real number.

12.1.7.23 nc_language_use

The nc_language gives_use which RS274 NC language dialect to write. The choices are (a) hexapod or (b) RS274/NGC.

12.1.7.24 plunge_feed_factor_use

The plunge_feed_factor_use is a factor by which to multiply the feed rate that would otherwise apply to obtain the rate for doing plunge cutting. Its value is a positive real number.

12.1.7.25 retract_distance_high_use

The retract_distance_high_use gives an incremental high distance to move above the part between machining features. Its value is a positive real number.

12.1.7.26 retract_distance_low_use

The retract_distance_low_use gives an incremental low distance to move above the part between machining features. Its value is a positive real number.

12.1.7.27 slot_feed_factor_use

The slot_feed_factor_use is a factor by which to multiply the feed rate that would otherwise apply to obtain the rate for doing slot cutting. Its value is a positive real number.

12.1.7.28 spiral_feed_factor_use

The spiral_feed_factor_use is a factor by which to multiply the feed rate that would otherwise apply to obtain the rate for doing spiral cutting. Its value is a positive real number.

12.1.8 Work Options

The work options file “data/work_options1.stp” is read by the Work Planner each time it executes the workpl_init function. An example of a work options file is shown in Appendix B.3.

12.1.8.1 angle_error_max_use

The angle_error_max_use gives the largest angle in degrees that the actual X-axis may be rotated from the nominal part X-axis in machine coordinates as determined before inspection begins. Its value is a positive real number.

12.1.8.2 length_units_use

The length_units_use option indicates (only! - see Section 3.8) what length units are used in the rest of the options file (for locations, thicknesses, etc.). The choices are: (a) use_in or (b) use_mm.

12.1.8.3 locating_method_use

The locating_method_use gives the method of locating the part during pure inspection. The choices are:

1. block_block_auto — put one face of the part on the table and select two other faces automatically to serve as datums.
2. block_block_user — put one face of the part on the table and have the user select two other faces to serve as datums.
3. block_datums_auto — put one face of the part on the table and select two other datums automatically.
4. block_datums_user — put one face of the part on the table and have the user select two other datums.
5. block_model_auto — put one face of the part on the table and select two features automatically to serve as datums.
6. block_model_user — put one face of the part on the table and have the user select two features to serve as datums.
7. datums_auto — select three datums automatically.
8. datums_user — have the user select three datums.
9. all_auto — select three features automatically to serve as datums.
10. all_user — have the user select three features to serve as datums.

Only block_block_auto is implemented in FBICS.

12.1.8.4 origin_error_max_use

The origin_error_max_use gives the largest distance that the part origin may be from its nominal location in machine coordinates as determined before inspection begins. Its value is a positive real number. As used, the error in the Z-coordinate is expected to be negligible.

12.1.8.5 shape_error_max_use

The shape_error_max_use gives the largest deviation of the actual location of a point from its nominal location on the part in part coordinates. Its value is a positive real number.

12.1.9 FBICS_ALPS

Pure FBICS_ALPS STEP Part 21 files are not used in FBICS, but Part 21 files representing cell-level process plans (see Section 12.1.12) and work-level process plans (see Section 12.1.13 and Section 12.1.14) consisting largely of things from FBICS_ALPS, but with additional types of data added, are used.

12.1.10 Expressions

This is an EXPRESS model of C language expressions, plus the C if - else if- else construct. Any expression (other than an if - else if - else expression) written in a STEP Part21 file following this model has a trivially easy translation to C by substituting for names of operators, deleting commas, and inserting semicolons. If - else if - else expressions are easily translated to C by also inserting “if” or “else” in the appropriate places and removing any empty parentheses. The meaning of the expression is the same as the meaning of the C counterpart. In this model, as in C expressions, all factors in an expression are numbers. Some of the operators used in the model are not found in C itself, but are found in the standard C math library. Figure 18 shows the operators used in the Part 21 files with the C translations and an example of an if - else if - else expression from a STEP Part 21 file with its C equivalent.

Operator Translation Table

FBICS	C
.ABSVAL.	abs
.BIGR.	>
.BIGR_EQ.	>=
.B_AND.	&&
.B_NOT.	!
.B_OR.	||
.COSINE.	cos
.DIVI.	/
.E_TO.	exp
.EQUAL.	==
.LESS.	<
.LESS_EQ.	<=
.LOG_E.	log
.MINUS.	-
.PLUS.	+
.ROOT.	sqrt
.SINE.	sin
.TANG.	tan
.TIMES.	*

Example Expression from STEP Part 21 file

```
#50  = IF_THEN_EXPRESSION(
/* if       */   (('material', .EQUAL.,'wax'),
                   ('diameter', .TIMES., 2.0)),
/* else if*/   ((('tool_type', .EQUAL., 'twist_drill'),
                    ('diameter')),
/* else if*/   ((('tool_type', .EQUAL., 'endmill'),
                   .B_OR., ('tool_type', .EQUAL.,
                                'ballnose_endmill')),
                   ('diameter', .TIMES., 0.5))),
/* else     */
                   (4.444));
```

Translation of Example Expression into C

```
if (material == wax)
        (diameter * 2.0);
else if (tool_type == twist_drill)
        (diameter);
else if ((tool_type == endmill)
        || (tool_type ==
           ballnose_endmill))
       (diameter * 0.5);
else
       4.444;
```

Figure 18. FBICS Expressions

The expressions model is used through FBICS_ALPS and through the tool_usage_rules model. FBICS is not currently using the expression capability in FBICS_ALPS but is using the capability in tool_usage_rules, as described in Section 12.1.11.1. No pure expressions STEP Part 21 files are used.

12.1.11 Tool Usage Rules

FBICS uses rules to determine how a tool is used in an executable operation, namely its spindle speed, feed rate, (horizontal) stepover, (vertical) pass depth, flood coolant use, and mist coolant use. The rules are not hard coded. Rather, they are soft coded as rules in a data file which is loaded when the Work Planner (the module that uses the rules) is initialized. The name of the rules data file to use is one of the shop options used in FBICS.

Most of the machinery for dealing with rules is in the expressions EXPRESS schema and the software for expression evaluation. The FBICS implementation of tool usage rules uses these items plus: (1) an EXPRESS model for rules, (2) a set of variables that may be used in tool_use_rules, and (3) a data file with rules.

A tool use rules STEP Part 21 file is used by the Work Planner. The file is read when the planner initializes or re-initializes, and an in-memory representation of its contents is attached to the Work Planner world model. The file is based on the both the expression schema and the tool_usage_rules schema. An example of a tool use rules file is shown in Appendix B.6.

12.1.11.1 Tool_Usage_Rules Schema and Data

The tool_usage_rules model is a special-purpose model written for FBICS. It is included in the fbics_combo schema and defines only four items:

1. the type “rule_type”, which is an enumeration of six values: speed, feed, stepover, pass_depth, flood, and mist.
2. the entity tool_use_rule, which has an attribute giving its type as one of the six just listed and an attribute which is an expression to evaluate to get a value.
3. the entity “class_instances”, whose attributes are a class name and a list of strings which are values that the class may have. For example, a class_instances might be defined with the name “material” and the allowable values of (“aluminum”, “brass”, “steel”, “wax”). A class_instances is an enumeration.
4. The entity “tool_use_rules”, whose attributes are a list of tool_use_rule, and a list of class_instances.

The expression used in the current rules file for pass depth, for example, is shown in Figure 18.

12.1.11.2 Tool Use Variables

As may be seen in Figure 18, the variable references used in expressions in FBICS are modeled as strings. The expression schema, however, does not model “variable”; it is left to the application using the expression schema to model variables in such a way that variable references can be evaluated to numbers, since all variable values are numbers.

In FBICS, variables are used only in the tool_usage_rules of the Work Planner. All information needed about variables is kept in the Work Planner world model. Some of the information gets into the model when the tool_usage_rules file is read, some gets in when a tool is selected, and

some gets in when the rules are evaluated.

There are two types of variables: those whose values are kept as the values of named attributes in the Work Planner world model, and those whose values are determined by their position in a class_instances. FBICS requires that the tool_usage_rules file include exactly one instance of a tool_use_rules entity, and that the instance include at least two class_instances: materials and tool_types; a check is made when the tool_usage_rules file is read.

For variables kept in the Work Planner world model, the data member name in the model is the variable name (or a synonym of the variable name). These variables fall in three sets (the names given below are the data member names):

1. variables that represent attributes of the currently selected tool: diameter, number_of_flutes, tip_angle, and tool_type.
2. variables that represent machining attributes for the currently selected tool: feed_rate, flood (coolant), mist (coolant), pass_depth, spindle_rpm, and stepover.
3. variables that represent other things: material (material from which part_in is made).

The value of a variable whose name appears in a class_instances is the (zero-based) index of the variable name in the list of allowable names. The variable “wax” would evaluate to 3 in the materials list (“aluminum”, “brass”, “steel”, “wax”), since it is the fourth item in the list of materials.

When a variable is to be evaluated, FBICS first checks if its name is one of those which corresponds to a data member of the Work Planner world model. It it is, FBICS checks whether the variable has been set; if so, the value is returned; if not, an error occurs. If the variable name does not correspond to a world model data member, FBICS checks whether it is included in a class_instances list. It is an error to put a variable in the tool_usage_rules file whose value cannot be found as just described.

12.1.11.3 Rule Evaluation

When a tool_use_rule is evaluated, the world model data member whose name corresponds to the rule_type is given a value. This is logically equivalent to setting the value of the variable. FBICS keeps track of whether the corresponding world model data member has been set. A variable may occur in the expression of a tool_use_rule that is evaluated after the value of the corresponding world model data member has been set. In the rule evaluation software, each tool_use_rule is evaluated in the order in which it appears in the list that is in the tool_use_rules instance.

In the tool_usage_rules file, any number of instances of tool_use_rule may be defined in any order. The same type of rule may be given more than once in a list of rules (using two rules for spindle speed, for example, makes it easy to put an upper limit on spindle speed — the second rule says if the speed is more than the maximum, set it to the maximum).

12.1.12 Cell-level Tasks

The cell-level tasks model is a tiny special-purpose model written for FBICS. It is included in the fbics_combo schema. The only item defined in the model is the entity run_setup, which is a subtype of primitive_task_node, defined in FBICS_ALPS. Run_setup identifies the name of the file describing the setup to be run.

Cell-level task STEP Part 21 files are used for stage-one cell-level process plans. They are generated by the Cell Planner at planning time and read by the Cell Planner when executing a stage-one plan or generating a stage-two plan. Since the cell-level tasks model is built on top of FBICS_ALPS, a stage-one cell-level process plan may contain instances of other entities defined in FBICS_ALPS. An example of a stage-one cell-level process plan is shown in Appendix A.4.

No cell-level executable operations are modeled in EXPRESS in FBICS because the information content of a what could be a “run_setup_ex” operation is small enough to fit in a message. When the Cell Controller commands the Work Controller to run a plan to do the work of a setup, all it does is send a `WORK_RUN1_MSG` or a `WORK_RUN2_MSG`.

12.1.13 Work-level Inspection Tasks

The work-level inspection tasks model is a small special-purpose model written for FBICS. It is included in the fbics_combo schema. It uses the FBICS_ALPS schema but does not use the AP 224 schema. Work-level inspection tasks are subtypes of primitive_task_node, defined in FBICS_ALPS.

The entity inspection_task is defined in the model as a subtype of primitive_task_node. Three subtypes of inspection_task are defined: locate_part, inspect_feature_geometry, and inspect_feature_surface. Only the first two have been implemented. Inspect_feature_surface will need additional attributes to be usable.

Work-level inspection task STEP Part 21 files are used for stage-one work-level process plans. They are generated by the Work Planner at planning time and read by the Work Planner when executing a stage-one plan or generating a stage-two plan. If a plan is for pure inspection, only these inspection tasks may be used. If a plan is for machining with inspection, it will contain both inspection tasks and machining tasks. Since the work-level inspection tasks model is built on top of FBICS_ALPS, a work-level process plan may contain instances of other entities defined in FBICS_ALPS.

The names of work-level task STEP Part 21 files are created by the Cell Planner and included in setup files. The Work Planner discovers what name to use for a process plan file it is to write by reading the setup file.

12.1.14 Work-level Machining Tasks

The work-level machining tasks model is a special-purpose model written for FBICS. It is included in the fbics_combo schema. It uses the FBICS_ALPS schema but does not use the AP 224 schema. Work-level machining tasks are subtypes of primitive_task_node, defined in FBICS_ALPS.

Work-level machining task STEP Part 21 files are used for stage-one work-level process plans. They are generated by the Work Planner at planning time, and read by the Work Planner when executing a stage-one plan or generating a stage-two plan. If a plan is for pure machining, only these machining tasks may be used. If a plan is for machining with inspection, it will contain both inspection tasks and machining tasks. Since the work-level machining tasks model is built on top of FBICS_ALPS, a work-level process plan may contain instances of other entities defined in FBICS_ALPS. Examples of stage-one work-level process plans are shown in Appendix A.10 and Appendix A.11.

The following machining tasks are defined for use in work-level process plans. Of these, FBICS currently handles counterboring, finish_mill (for rectangular pockets), finish_mill_adaptive, and twist_drill. Each of these tasks includes a pointer to a machining feature on which the task is to be performed. The form of the pointer is the index number of the feature from the list of features included in the features file used in the same setup as the process plan.

1. bore
2. center_drill
3. counterboring
4. countersinking
5. face_mill
6. finish_mill
7. finish_mill_adaptive
8. fly_cut
9. machine_chamfer
10. machine_round
11. peripheral_mill
12. ream
13. rough_mill
14. slot_mill
15. tapping
16. twist_drill

Figure 19 shows the hierarchy of entities defined in the two work tasks models (inspection and machining). The figure shows both items from the above list and supertypes of the items. Items from the list are the leaves of the supertype-subtype tree shown in the figure. The attributes of primitive_task_node shown in the figure are all inherited from node, except for duration.

In addition to the entities shown in the figure, the EXPRESS schema contains a few entities which are not being used (and probably should be deleted from the schema).

12.1.15 Work-level Executable Operations

The work-level executable operations model is a special-purpose model written for FBICS. It is included in the fbics_combo schema. It uses the AP 224 schema but does not use the FBICS_ALPS schema. Work-level executable operations are generated (not executed) at the work level.

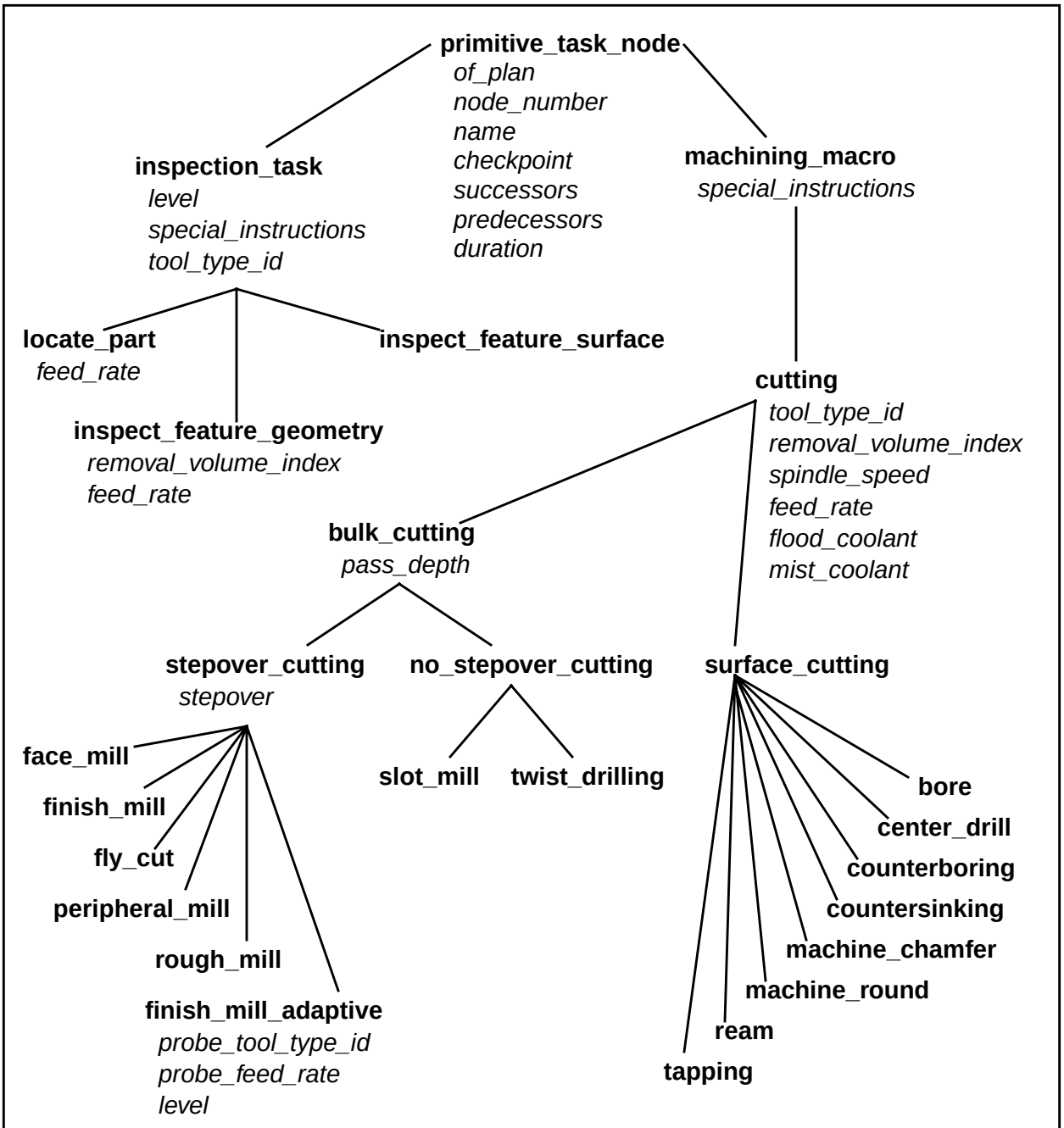


Figure 19. Work-level Tasks

Tasks are shown in **boldface** type.
Supertypes are connected to subtypes by lines, with the supertype higher on the page.
Attributes names are shown in *italic* type. Data types of attributes are not shown.
Only leaf nodes may be instantiated.
To find all the attributes of a task, trace down the tree from “primitive_task_node” to the task, and include the attributes of every node along the path, in order.

The following executable operations are defined. Of these, the FBICS NC code generator handles: coolant_ex, counterboring_ex, end_nc_ex, finish_mill_ex (for rectangular pockets), nc_change_ex, start_nc_ex, and twist_drilling_ex (see Section 9.5). The FBICS DMIS code generator handles: end_inspect_ex, inspect_change_ex, inspect_geometry_ex, locate_part_block_block_ex, and start_inspect_ex (see Section 9.6). Most operations may act on only one type of AP 224 feature, but some, such as finish_mill_ex, may be used with several different types of AP 224 feature.

1. bore_ex
2. center_drill_ex
3. coolant_ex
4. counterboring_ex
5. countersinking_ex
6. end_inspect_ex
7. end_nc_ex
8. face_mill_ex
9. finish_mill_ex
10. fly_cut_ex
11. inspect_change_ex
12. inspect_geometry_ex
13. inspect_surface_ex
14. locate_part_block_block_ex
15. machine_chamfer_ex
16. machine_round_ex
17. nc_change_ex
18. peripheral_mill_ex
19. ream_ex
20. rough_mill_ex
21. slot_mill_ex
22. start_inspect_ex
23. start_nc_ex
24. tapping_ex
25. twist_drilling_ex

Work-level executable operation STEP Part 21 files are used in large numbers during execution of stage-one work-level process plans and generation of stage-two work-level process plans. For each work-level executable operation to be carried out, the Work Planner writes a STEP Part 21 file describing the operation. Shortly thereafter, the Task Planner reads the file and uses the information in it to help generate machining code or inspection code. Many examples of work-level executable operation files are shown in Appendix A.12 and Appendix A.13.

Unlike a task_node in a process plan, which may have only a reference to a feature, an executable operation that uses a feature includes a full model of the feature. Typically, most of the contents of a work-level executable operation file will be devoted to a feature description.

Figure 20 shows the hierarchy of entities defined in the executable operations model. The figure shows both items from the above list and supertypes of the items. Items from the list are the leaves of the supertype-subtype tree shown in the figure.

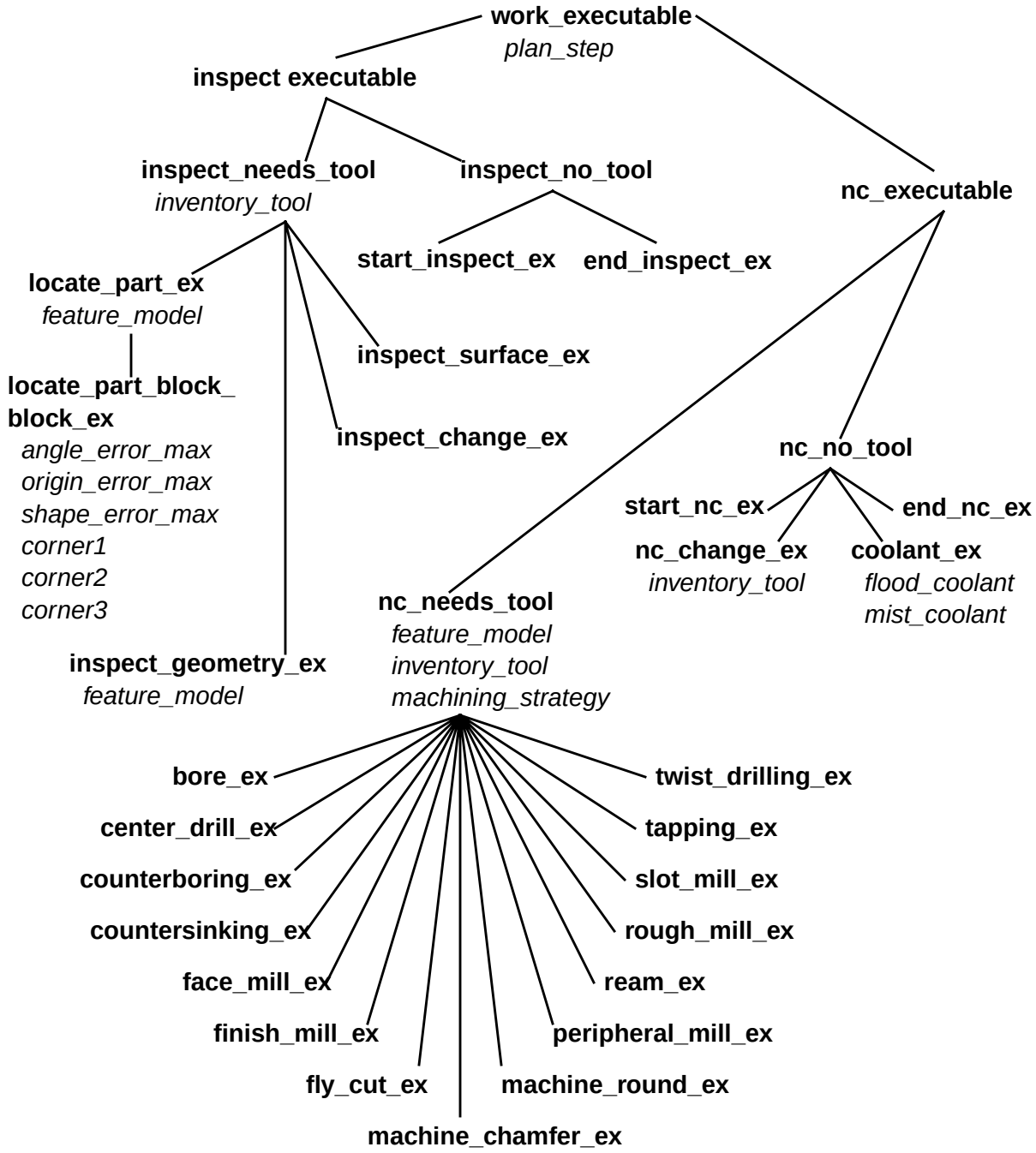


Figure 20. Work-level Executable Operations

Executable operations are shown in **boldface** type.
Supertypes are connected to subtypes by lines, with the supertype higher on the page.
Attributes names are shown in *italic* type. Attribute data types are not shown.
Only leaf nodes may be instantiated.
To find all the attributes of an operation, trace down the tree from “executable” to the operation, and include the attributes of every node along the path, in order.

12.1.16 Setup

The setup schema is a special-purpose schema written for FBICS. It does not use other schemas.

A setup specifies:

1. a set of data about data, primarily the names of the files associated with the work done in one setup for a specific part. These include the names of: the setup file itself, the part_out file, the fixture file, the work-level process plan file, the removal_volumes file, and the part_in file.
2. a description of the location of the fixture, workpiece, design, and features with respect to the coordinate system of the machining center or coordinate measuring machine.
3. a description of a box-shaped volume containing the workpiece.

Setup STEP Part 21 files are written by the Cell Planner during planning. They are read by the Work Planner during planning and during plan execution. They are read by the Task Planner when `TASK_OPEN_MSG`s are received. Sample setup files are shown in Appendix A.6 and Appendix A.7.

12.1.17 Stage-two Plans

The model for stage-two cell-level and work-level plans consists of only two entities: one_operation and operation_plan. Both are defined in the fbics_combo schema. One_operation includes an identifier for the operation type and the name of a file describing the operation in more detail. Operation type identifiers used in the model are simply integers. Operation_plan includes a list of one_operations.

It would be useful to make this model a little richer. The operation type identifiers, which are currently modeled in a C++ header file, not in EXPRESS, might be brought into EXPRESS as an enumeration, for example.

An example of a cell-level stage-two plan file is shown in Appendix A.5.

12.1.18 Tool Catalog

The tool catalog model is almost entirely a large subset of the model built in EXPRESS by the NIST Manufacturing Systems Integration Division and contractors of that division (see Section 2.5.6 and [Jurrens]). The catalog includes milling and turning machine tools; cutting tools appropriate to the processes of milling, drilling, boring, reaming, tapping, and turning; cutting tool inserts; and the tool holding and assembly components required to mount the tools to the machines. The data, while extensive, do not include a solid model of the tool. For (at least) twist drills and endmills, the data are adequate to support building a solid model of the volume occupied by the tool while it is spinning, which is the relevant volume with which to calculate interferences and swept cut volumes.

The NIST model includes only cutting tools. FBICS also requires inspection tools. A few entities were added to the NIST model to provide for describing probes for inspection.

The tool catalog lists all types of tools available in principle for use. Tool catalog data includes, for example:

1. tool type id
2. nominal dimensions (such as length and diameter)
3. material from which the tool is made
4. materials the tool can cut
5. number of flutes
6. maximum RPM of use
7. maximum number of reworks

Tool catalog STEP Part 21 files are written by hand. An example of a tool catalog file is shown in Appendix B.4.

A tool catalog file is read by the Work Planner when it initializes or re-initializes. A data structure representing the tool catalog is attached to the Work Planner world model. The tool catalog is used directly by the Work Planner for planning and indirectly (through the tool inventory) for plan execution.

A tool catalog file is read by the Task Planner when it initializes or re-initializes. A data structure representing the tool catalog is attached to the Task Planner world model. The tool catalog is used by the Task Planner indirectly (through the tool inventory) during code generation.

12.1.19 Tool Inventory

The tool inventory schema is a special-purpose schema written for FBICS. It uses the tool catalog schema.

The tool inventory lists the tools available. “Available” might mean “in the carousel” or “easily obtainable for putting in the carousel.” In FBICS, each tool_instance has a slot number in the carousel.

Each tool in the inventory is an instance of some tool type described in the catalog and inherits all the information about that type of tool contained in the catalog. A catalog tool and inventory tools of that type are linked by having the catalog tool name be an attribute of the inventory tool. Each inventory tool has some information associated with it which does not exist in the catalog, specifically:

1. tool_id — an identifier for the tool.
2. carousel_slot — the slot in the tool changer carousel where the tool is (or will be).
3. number_of_reworks — how many times this tool has been reworked.

It would be useful to add service life data, total time of cutting, for example, but FBICS is not currently using service life data.

Tool inventory STEP Part 21 files are written by hand. An example of a tool inventory file is shown in Appendix B.5.

A tool inventory file is read by the Work Planner when it initializes or re-initializes. A data structure representing the tool inventory is attached to the Work Planner world model. In this data structure, each tool in the inventory has a pointer to the corresponding catalog tool. When the tool inventory file is first read, this is a null pointer, but after both catalog and inventory have been read, there is a linking step in which the pointer is set. The Work Planner uses the tool inventory during execution.

The Task Planner sets up the tool inventory in the same manner as the Work Planner. The Task Planner uses the tool inventory during code generation.

12.2 DMIS Files

This section gives an overview of the DMIS language.

12.2.1 Introduction

The DMIS interpreter used in FBICS conforms to Revision 3.0 of the DMIS language.

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

1. defining and measuring features (planes, circles, cylinders, lines, etc.)
2. defining tolerances and determining if features are in or out of tolerance
3. defining coordinate systems (and activating and deactivating them)
4. defining sensor characteristics and changing sensors
5. setting machine parameters (feed rates, probe tip radius, etc.)
6. 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 which 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.

12.2.2 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 which 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.

12.2.3 Programs and Files

Statements may be collected in a file to make a program. A program consists of a DMISMN statement at the beginning, an ENDFIL statement at the end, and any number of other types of statements in between.

FBICS uses DMIS input files which contain sections of programs rather than entire programs. Each file carries out a task, such as opening a program, inspecting an AP 224 feature, or changing probes. The program sections must be such that if they were all concatenated in order, they would make a legitimate DMIS program. The DMIS interpreter keeps track of when it is working on a program and when it is not. It returns different values for the end of a program and the end of the

current section of a program which is not the end of a program.

Several examples of DMIS files are shown in Appendix A.13.

12.2.4 Program Subunits

DMIS includes program subunits. A program subunit is a sequence of statements which forms a functional group. Each type of program subunit requires a particular type of first statement and a particular type of last statement. Two types of subunit are used in FBICS: measurement sequence and motion sequence.

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.

12.2.5 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, circle, cone, cparln, cylinder, ellipse, gcurve, gsurf, line, object, parpln pattern, plane, point, rectangle, and sphere. The underlined five are implemented in the interpreter. DMIS features (such as the cylindrical side of a hole) may be visible on a part being inspected or they may be purely conceptual (such as the line which 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 which 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.

Each feature has a label which serves to identify it within a DMIS program. No other feature may share that label in the same program.

12.2.6 Tolerances

DMIS tolerances also have labels which 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 millimeter 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.

12.2.7 Comments

A DMIS program may include comments. A comment is a line which 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.

12.3 RS274 Files

12.3.1 Numerical Control Programming Language RS274

RS274 is a programming language for numerically controlled (NC) machine tools, which has been used for many years. The most recent standard version of RS274 is RS274-D, which was completed in 1979. It is described in the document "EIA Standard EIA-274-D" by the Electronic Industries Association [EIA]. Most NC machine tools can be run using programs written in RS274. Implementations of the language differ from machine to machine, however, and a program that runs on one machine probably will not run on one from a different maker.

The RS274 language is based on lines of code. Each line may include commands to a machine tool to do several different things. A line is terminated by a carriage return or line feed. Lines of code may be collected in a file to make a program.

A typical line of code consists of a line number at the beginning followed by one or more "words." A word consists of a letter followed by a number or an expression that can be evaluated to a number. A word may either give a command or provide an argument to a command. For example, "G1 X3" is a valid line of code with two words. "G1" is a command meaning "move in a straight line at the programmed feed rate," and "X3" provides an argument value (the value of X should be 3 at the end of the move) to the command. Most RS274 commands start with either G or M (for miscellaneous). The words for these commands are called "G codes" and "M codes." The order of words on a line is usually irrelevant.

12.3.2 The RS274/NGC Language

The NGC project (see Section 2.5.4) developed a specification for the RS274/NGC language, a numerical control code language for machining and turning centers. The RS274/NGC language has many capabilities beyond those of RS274-D. The specification was originally given in a 1992 report prepared by the Allen Bradley company [Allen Bradley]. A second draft of that document was released in 1994 by the National Center for Manufacturing Sciences [NCMS]. The NIST RS274/NGC interpreter uses the second draft as the specification.

12.3.3 FBICS use of RS274

The FBICS Task Planner generates files of NC code either in RS274/NGC or in the Hexapod dialect [Ingersoll] of RS274. The dialect is determined by the setting of task options. The RS274/NGC interpreter in the Task Planner reads files in that dialect and interprets them. Hexapod code generated off-line by FBICS has been downloaded to the Hexapod controller and used to machine parts. Many examples of RS274/NGC files are shown in Appendix A.12 and Appendix A.13.

12.4 Graphics Files

A simple file format, devised some years ago, is used for graphics files. The machine-readable content of graphics file starts with the word “data” on its own line; any preceding lines are ignored as comments. The machine-readable content ends with the word “end” on its own line; any following lines are ignored as comments. In between are any number of polygons and lines, in any order. A (graphics) line consists of three file lines: (a) the word “line” on the first line, (b) three real numbers representing the coordinates of one end of the line on the second line, and (c) three real numbers representing the coordinates of the other end of the line on the third line. A polygon consists of four or more lines in the file: (a) the word “polygon” on the first line and (b) three real numbers representing the coordinates of the nth point of the polygon on the (n+1)th line. The points of the polygon should be co-planar. The last point of a polygon should not be the same as the first point; it is understood that the polygon should be closed. Files may be all lines, all polygons, or a mixture of the two.

FBICS graphics files are written by the Modeler, using the Parasolid faceting routines. Each graphics file describes a solid object two ways: as a wire frame made of lines, and as a surface covered by polygons. Graphics files are read and displayed by the Graphic Display. Six files with fixed names are used. Each file is overwritten repeatedly as FBICS runs. When one of these files is read, the old picture is discarded and a new one displayed. An example of a graphics file is shown in Appendix A.15.

12.5 File Names

FBICS generates many files when it runs, each of which must be named. Often, hundreds of files are associated with a single part. Several naming conventions are used. For clarity, file names and parts of file names are given in `courier` type in this section.

Base file names may be given by the user in cell-level commands. This section describes how full file names are created from base names given this way. Base file names or full file names may also be given by the user in work-level or task-level commands.

Most files generated or used by FBICS have a suffix of some sort. All STEP Part 21 files generated by FBICS have the suffix `.stp`. STEP Part 21 files used by FBICS but not generated by FBICS should also have the `.stp` suffix. Files of NC code generated by FBICS have the suffix `.nc`. Files of DMIS code generated by FBICS have the suffix `.dmis`.

12.5.1 Setup Files

Setup files are generated and revised by the Cell Planner as FBICS runs. Names for setup files start with a base setup name provided by the user when the user gives a planning command to the Cell Controller. An extension is added to the base name of the form `_N`, where `N` is the number of the setup.

Setup files are generated initially with some information marked “not_set”. The setup files are rewritten later with the unset information replaced. When setup files are made off-line, the revised setup files should be kept for later use during plan execution. These are named by adding `_keep` to the name of the original setup file. When plans are made on-line (during execution) the same types of file are only for one-time use, so `_temp` is added instead.

As an example, if the base name for setup files is `ase`, the second setup file will be named

ase_2.stp. When this is revised, the new file will be named either ase_2_keep.stp or ase_2_temp.stp.

12.5.2 Intermediate Workpiece Files

If two or more setups are needed to make a part, the shape of the workpiece coming out of one setup and going into the next must be saved. The Cell Planner does this by writing AP 224 files representing these intermediate workpiece shapes. The base name for the intermediate workpiece shape files is the name of the starting workpiece shape file (without the .stp suffix). As with setups, either _keep or _temp is concatenated with the base name. The setup number is also concatenated (but at the end). For example, if the name of the starting workpiece file is start.stp, and the file is temporary, then the name of the file for the workpiece coming out of the second of three or more setups will be start_temp_2.stp.

12.5.3 Process Plan Files

Process plans for the Cell Controller are generated by the Cell Planner as FBICS runs, and process plans for the Work Controller are generated by the Work Planner. Names for process files start with a base name provided by the user when the user gives a planning command to the Cell Controller. The extension _1cell is added to the base for stage-one cell-level plan names. For stage-two cell-level plan names, _2cell is added instead. For example, if the base name is apl, the stage-one cell-level plan name will be named apl_1cell.stp.

A work-level process plan is associated with each setup. Work-level process plan names are formed from the base name by adding two extensions. The first extension is _N, where N is the number of the setup. The second extension is _1work for stage-one plans and _2work for stage-two plan names. For example, if the base name is apl, the name for the stage-one work-level plan for the third setup will be apl_3_1work.stp.

12.5.4 Feature Files

The Cell Planner writes files of features to be made or inspected in each setup. These are subsets of the features found on the design of the part. Feature file names are formed by extending a base name given by the user with _N, where N is the number of the setup. For example, if the base name is feats, the name for the features file to be used in the first setup will be feats_1.stp.

12.5.5 Task-level Executable Instruction and Code Files

Task-level executable instruction files (which are STEP Part 21 files) are written by the Work Planner and read by the Task Planner. The names for these files are formed by concatenating the base plan name with two extensions. The first extension is _N, where N is the number of the setup. The second extension is _M, where M is the number of the executable instruction in the list of executable instructions the Work Planner is executing. The second extension always uses at least three spaces for M. If M is 18, for example, the second extension is _018.

Planning for an executable instruction in the Task Controller always causes either a single DMIS code file or a single NC code file to be written. The name of the code file is the same as the name of the executable instruction file with a different suffix, either .dmis or .nc, as the case may be. For example, if the base plan name is apl and a work-level plan for the third setup is being executed, the name for the 18th executable instruction file will be apl_3_018.stp. If NC code

is generated from that file, the name of the NC code file will be `apl_3_018.nc`.

12.5.6 Graphics Files

As it runs, FBICS writes and rewrites six graphics files. These have fixed names, which are: `fbics_access_picture`, `fbics_fixture_picture`, `fbics_part_in_picture`, `fbics_part_now_picture`, `fbics_part_out_picture`, and `fbics_volume_picture`. See Section 11 and Section 12.4.

13 Strengths and Limitations

13.1 Strengths

The hierarchical division of labor in the Cell, Work, and Task controllers made by FBICS is a great strength. Focusing on an entire part, a setup, and a feature at successively lower hierarchical levels is the right way to do it.

The data interfaces between controllers are a strength, since they use standard data formats (STEP APs, DMIS, and RS274) where available and a standard modeling language (EXPRESS) accompanied by a standard data representation format (STEP Part 21) elsewhere. The data interfaces provide the hooks for enabling user participation (essential for a successful commercial implementation) at every stage of FBICS operation.

The modularization of FBICS is a strength. Every module has clear, explicitly defined interfaces, implemented in APIs, interprocess messages, user commands, and data.

The ability to go from art (features, at least) to part fully automatically is a strength.

The use of a solid modeler to support process planning at all levels is a strength. Many key planning decisions require the support of a solid modeler if they are to be made automatically.

The use of ALPS to represent process plans is a strength. ALPS is both powerful and flexible. It works well for plans related to discrete part manufacturing.

13.2 Limitations

FBICS assumes that machining features have already been identified when the planning activity starts. This makes the problem easier, but also makes it unrealistic. To make an effective and efficient process plan, it is necessary to define the machining features and to plan how to make them concurrently.

The point on workpiece test for inspection is not always checking that entire patches around the candidate probe point lie on the workpiece. In some cases it is checking only for the point itself. This is not, in general, safe.

Operations to machine only a few features from the AP224 feature suite are implemented.

Operations to inspect only a few features from the AP224 feature suite are implemented.

All the inspection operation verification functions are stubs that do not really verify anything. This means that inspection may be performed only with extreme caution. The verification functions for the machining operations that are implemented are fully implemented, however — but they could be improved.

The work control level contains only a single workstation. For the existing FBICS applications, there should be two workstations, one for a CMM and one for a machining center. The CMM workstation would do only pure inspection. The machining center workstation would do pure machining, pure inspection, or machining with in-process inspection.

The Graphics Display interleaves showing the views of the Cell, Work, and Task controllers without informing the user which is being shown. This is very confusing. Buttons should be added to the Graphics Display control panel to allow the user to select which view is shown and to inform the viewer which view is being shown.

Batch size is not considered in planning.

Fixture selection in FBICS is barely above the stub level. There are only two fixtures to choose from, both of which are vises with parallel jaws, and it is assumed that if the fixture is open wide enough to hold the part, fixturing is adequate. A much more realistic and sophisticated method of fixture selection is needed.

FBICS does not deal with tool holders. The current assumption is that standard tool holders are used. A fully functional FBICS would include tool holder data sufficient to check for collisions of the end of the tool holder with the workpiece and fixture.

FBICS is not dealing with moving tools in and out of workstations. ALPS provides methods of specifying the allocation of resources that could be used in FBICS. If explicit allocation of tools to a workstation is implemented, it is expected that all tool allocation will be performed before the machining done in a setup is started. It is not expected that tool allocation steps will ever be included in FBICS process plans for machining and inspection, since shops where it is a good idea to postpone getting a tool until cutting is in progress are believed to be very rare.

It would be a simple matter to extract a list of catalog tools to be used from work-level process plans, since each step of a plan requiring a tool includes the catalog ID for the tool.

FBICS does not deal with machine capabilities — work volume, horsepower, etc.

Error recovery is very limited.

The FBICS Cell Planner does not consider that some features do not have to have their native Z-axis parallel to the machine Z-axis in order to be machined. For example, when a pocket extends outside of a workpiece, subtracting the pocket from the workpiece may produce a step on the workpiece. If this is the case, the pocket could be machined either from the top or the side. The Cell Planner does not realize that machining from the side is possible. Thus, the Cell Planner may decide to use more setups than necessary.

FBICS is not dealing with scheduling and does not have the hooks required to use it in a scheduled environment. Adding scheduling may be expected to be a major effort.

14 Software

The software for FBICS is done in the C++ language. It is compiled and linked with the widely available Gnu C++ compiler. Most of it may be compiled with other C++ compilers. There is nothing conceptually unusual about the way executables are built. Source code is handwritten or automatically generated and compiled into object or archive files. The resulting object or archive files are linked with each other and additional such files obtained from commercial sources or from within ISD to produce executable files. The executable files are machine specific and have been produced only for the Sun Solaris operating system running on Sun computers.

Since one of the objectives of FBICS is to test STEP techniques and standards, heavy use is made of data in STEP format and software for manipulating STEP models and data.

14.1 Modularization

The handwritten source code has been modularized to match the architecture of processes shown in Figure 1. The Fbics_Cell, Fbics_Work, and Fbics_Task processes each have two principal source code files: one for the controller body and one for the planner. Fbics_Task2, Fbics_Draw, and Fbics_Model each have only one principal source code file but are built by linking the object file corresponding to that source code with large additional object or archive files not specific to FBICS.

14.2 In-Line Documentation

Each function in the handwritten source code is documented in-line as shown in Figure 21 using the find_expression_value function as an example. The in-line documentation includes the function name, returned value, side effects, and called by (the other functions that call the function). Each argument to the function is given on a separate line with a comment giving the meaning of the argument. Local variables of a function are each declared on a separate line, and many functions include comments describing local variables similar to the comments describing arguments.

```
/* find_expression_value

Returned Value: how
If any of the following errors occur, this returns the error
code shown. Otherwise, it returns RET_OK.
  1. The expression is a NULL pointer:
     FWE_EXPRESSION_ERROR_NULL_POINTER_TO_EXPRESSION
  2. find_expression2_value returns an error code.
  3. find_if_then_expression_value returns an error code.

Side Effects:  The value of result is set.

Called By: set_attribute

*/

how find_expression_value( /* ARGUMENTS                        */
 expression * expr, /* an expression whose value is sought   */
 double * result)   /* the value of the expression, set here */
{code here}
```

Figure 21. In-Line Documentation Example

In addition to the type of documentation shown in Figure 21, if the workings of any function are complex, an explanation is given. The explanations average two or three paragraphs long. A few are two or more pages long.

14.3 Software Files

The software for FBICS includes handwritten and automatically generated C++ files (.h or .hh files and .c or .cc files), commercial, ISD, and automatically generated libraries (.a or .so files), essential data, executables, and miscellaneous other files. This section gives an overview of these files. Further details regarding how these files are used in FBICS are in the Makefile for FBICS and the “fbics” shell script.

This section does not describe part-specific[1] data files read and/or written by FBICS while it is running.

The commercial and ISD libraries described below require additional commonly available libraries (xgl and sockets, for example) not described here.

14.3.1 Handwritten C++ Code

FBICS uses a total of about 40,000 lines of handwritten .cc files and about 2,000 lines of .hh files. These include the class and function definitions specific to FBICS for the seven FBICS processes. Since the source code is heavily commented, perhaps half that many lines are actual code. Fbics_Cell, Fbics_Work, and Fbics_Task each have around 10,000 lines. Fbics_Model has 3,900 lines, Fbics_Draw 1,800 lines, the driver for Fbics_Task2 600 lines, and Fbics_Serve only 25

1. Workpiece designs, part designs, setups, feature sets, process plans, DMIS code, NC code, and fixture designs are part-specific.

lines.

For use in FBICS, this source code is compiled into object (.o) files and linked into one or more executables.

14.3.2 Automatically Generated C++ Code

FBICS uses automatically generated C++ code produced in two ways: (1) by the STEP Tools express2c++ utility and (2) by a combination of shell scripts and lex-based executables written for FBICS for automatically generating .h and .c files defining error codes and arrays of error messages for Fbics_Cell, Fbics_Work, Fbics_Task, and Fbics_Model.

The source code generated by the STEP Tools express2c++ utility is in four directories, each corresponding to an EXPRESS file, and totals 65,000 lines. The utility automatically generates a Makefile in each directory. For use in FBICS, each directory is compiled into an archive (.a) file and linked into one or more executables. A list of the four archive files is given in Section 12.1.3. In addition, the .h files are used (via #includes) in compiling the handwritten source code. For each archive file, the STEP Tools express2c++ utility also generates a data file with the same base name but with a .rose suffix. The .rose files are read by the STEP Tools library functions when FBICS runs, so FBICS cannot run without them.

The error code totals about 1,000 lines. For use in FBICS, each of four pairs of files (.h and .c) is compiled into an object file and linked into the appropriate executable. In addition, the .h files are used in compiling the handwritten source code.

14.3.3 Object and Archive Files from Other ISD Projects

FBICS uses three sets of header and object or archive files from other ISD projects. Two of these are for the RS274/NGC [Kramer17] and DMIS [Kramer18] interpreters that are linked into the Fbics_Task2 executable. The third set is for NML interprocess communication [Shackleford], which is linked into all seven FBICS executables.

14.3.4 Commercial Software Libraries

FBICS makes heavy use of three commercial software libraries. The versions of these libraries linked into FBICS in August 2003 are: STEP Tools ST-Developer v8 for manipulating STEP data, Parasolid 13.0 for solid modeling, and HOOPS 620 for 3D graphics. More recent versions of all three libraries are available and are expected to be readily usable.

14.3.5 Essential Data

FBICS uses eight files of site-specific (but not part-specific) data: tool catalog, tool inventory, tool use rules, shop options, task options, work options, DMIS_variables, and rs274ngc.var. The first six of these are read in by one or more of Fbics_Cell, Fbics_Work, and Fbics_Task each time the planner in the process is initialized. DMIS_variables is read by the DMIS interpreter in Fbics_Task2 if that interpreter is used on a DMIS program containing variables. rs274ngc.var is read by the RS274/NGC interpreter in Fbics_Task2 whenever that interpreter is initialized. These last two files are files of variables that should persist between uses of the interpreters. In both cases, the respective interpreter will rewrite the file when it exits to preserve the current values of persistent variables.

14.3.6 Executables

The seven FBICS executable files are: fbics_cell (4.1 Mb), fbics_work (5.3 Mb), fbics_task (5.4 Mb), fbics_task2 (2.1 Mb), fbics_model (24.3 Mb), fbics_draw (0.5 Mb), and fbics_serve (0.4 Mb). The fbics_model file is large because it includes all of the static Parasolid library. The fbics_draw file is deceptively small since it uses the HOOPS shared object (dynamically linked) library. When the fbics_draw file is executed, the process it starts is much larger (78 Mb on a Sun computer with 512 Mb of RAM, for example). The processes started when the other six files are executed are of the same order of magnitude as the files.

14.3.7 Other

Three other types of file are required to run FBICS.

A shell script named “fbics” is used to start FBICS running. It copies options files and a configuration file into files whose names are hard-coded, and then starts the seven FBICS processes in separate terminal windows.

The NML configuration file “configure_nml” is read by all seven FBICS processes. NML communications are discussed in Section 5.4.2.

A file “cell_commands1” is used as a command argument for invoking the Fbics_Cell executable. That file contains the commands suggested as the first few commands to Fbics_Cell. The format of this file is simply text identical to commands the user might type in, one command per file line.

14.4 Error Handling

The code handwritten for FBICS checks for many types of errors and reports them. The commercial and ISD software packages used in FBICS also do extensive error checking.

In all the handwritten FBICS code, if an error is detected, an explicit error message is displayed and control is passed up the function call stack to the user interface level. No use is made of the C++ try and catch mechanism. To make the passing of control smooth, every function in which an error can occur returns an (unsigned integer) code which is either zero (if there is no error) or the number of an error. If an error code is returned from a check made in a called function, the calling function returns that code to its caller, and so on up the function call stack. In addition to returning a specific code in case of an error, each function prints its name before returning, so that the user can see what the function call stack was at the time the error occurred. Seeing the function call stack is, of course, useful primarily to users who are also FBICS system builders.

14.4.1 Automatic Generation of Error Software

Automatic generation of error software is implemented in FBICS for the four largest sets of source code (for Fbics_Cell, Fbics_Work, Fbics_Task, and Fbics_Model). Among them, these sets of code have a total of about 500 error messages. Automatic generation of error software works as follows.

While writing source code, when a section of code is checking for an error, the programmer decides what a suitable error message would be and creates a symbol by writing the error message in upper case letters, replacing spaces with underscores, and putting a module-specific prefix at the front. For example, the Fbics_Work prefix is “FWE_”, so if the desired error message is “Unknown manufacturing feature”, then the symbol would be

FWE_UNKNOWN_MANUFACTURING_FEATURE. Functions that check for this error return the symbolic value of this symbol if the error occurs. An error reporter then prints the error string associated with that value.

The assignment of values to error symbols and the association of values with strings are done automatically. A utility written for FBICS and invoked automatically when make-ing any of the FBICS processes reads the source code and looks for places where error symbols are used. It records the error symbols, alphabetizes them, and removes duplicates. Then it writes a header file and a code file. The header file is a list of assignments of values to symbols. The code file is an array of error strings extracted from the error symbols. Each string is created by deleting the prefix, changing underscores to spaces, and changing all but the first letter of the symbol to lower case (this is the reverse of the process by which the symbol was created). So, for example, the symbol FWE_UNKNOWN_MANUFACTURING_FEATURE leads to the string “Unknown manufacturing feature” being printed as the 194th entry in the array of strings defined in the work_err.c file. In the corresponding header file, fwe_code.h, the line “#define FWE_UNKNOWN_MANUFACTURING_FEATURE 0xB927L” appears. To retrieve the string from the array, given the symbol value, the array index of the string is calculated by subtracting the value of the first symbol (0xB865L, i.e., 47205 decimal) from 0xB927L (i.e., 47399 decimal) and getting 194.

14.4.2 Error Recovery

A modest effort has been made to be able to recover from errors.

Recovering from an illegal command typed at the user interface of any of the three controllers is trivially easy since the state of the system has not changed. The user is notified and no attempt is made to execute the illegal command.

Recovering from other errors is more difficult. The only strategy that has been implemented is to stop the work in progress and re-initialize.

References

[Albus1] Albus, James S.; *A Theory of Intelligent Systems*; Control and Dynamic Systems; Vol. 45; 1991; pp. 197-248

[Albus2] Albus, James S.; McCain, Harry G.; Lumia, Ronald; *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*; NIST Technical Note 1235, 1989 Edition; National Institute of Standards and Technology; April 1989

[Albus3] Albus, James S.; *A Reference Model Architecture for Intelligent Systems Design*; NISTIR 5502; National Institute of Standards and Technology; September 1994

[Albus4] Albus, James S; et al; *NIST Support to the Next Generation Controller Program: 1991 Final Technical Report*; NISTIR 4888; National Institute of Standards and Technology, Gaithersburg, MD; July 1992

[Allen Bradley] Allen Bradley; *RS274/NGC for the Low End Controller*; First Draft; Allen Bradley; August 1992

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

[Catron] Catron, Bryan; Ray, Steven R.; *ALPS — A Language for Process Specification*; International Journal of Computer Integrated Manufacturing; Vol. 4, No. 2; 1991; pp 105 -113

[EIA] Electronic Industries Association; *EIA Standard EIA-274-D Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/ Positioning Numerically Controlled Machines*; Electronic Industries Association; Washington, DC; February 1979

[Ingersoll] Ingersoll Milling Machine Company; *Octahedral Hexapod Machine Operation (Preliminary) Manual*; Ingersoll Milling Machine Company; 1996

[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

[ISO3] ISO 10303-224:1998; *Industrial automation systems and integration - Product data representation and exchange - Part 224: Application Protocol: Mechanical Product Definition for Process Planning Using Machining Features*; ISO; Geneva, Switzerland; 1998

[Jurrens] Jurrens, Kevin K.; Fowler, James E.; Algeo, Mary E.; *Modeling of Manufacturing Resource Information*; NISTIR 5707; National Institute of Standards and Technology; July 1995

[Kramer1] Kramer, Thomas R.; Jun, Jau-Shi; *Software for an Automated Machining Workstation*; Proceedings of the 1986 International Machine Tool Technical Conference; September 1986; Chicago, Illinois; National Machine Tool Builders Association; 1986; pp. 12-9 through 12-44

[Kramer2] Kramer, Thomas R.; *Process Plan Expression, Generation, and Enhancement for the Vertical Workstation Milling Machine in the Automated Manufacturing Research Facility at the National Bureau of Standards*; NBSIR 87-3678; National Institute of Standards and Technology (formerly National Bureau of Standards); November 1987

[Kramer3] Kramer, Thomas R.; Strayer, W. Timothy; *Error Prevention and Detection in Data Preparation for the Vertical Workstation Milling Machine in the Automated Manufacturing Research Facility at the National Bureau of Standards*; NBSIR 87-3677; National Institute of Standards and Technology (formerly National Bureau of Standards); November 1987

[Kramer4] Kramer, Thomas R.; Weaver, Rebecca E.; *The Data Execution Module of the Vertical Workstation of the Automated Manufacturing Research Facility at the National Bureau of Standards*; NBSIR 88-3704; National Institute of Standards and Technology (formerly National Bureau of Standards); January 1988

[Kramer5] Kramer, Thomas R.; Jun, Jau-Shi; *The Design Protocol, Part Design Editor, and Geometry Library of the Vertical Workstation of the Automated Manufacturing Research Facility at the National Bureau of Standards*; NBSIR 88-3717; National Institute of Standards and Technology (formerly National Bureau of Standards); January 1988

[Kramer6] Kramer, Thomas R.; *A Parser that Converts a Boundary Representation into a Features Representation*; International Journal of Computer Integrated Manufacturing; Vol. 2, No. 3, May-June 1989; pp. 154-163; (also published as NISTIR 88-3864; National Institute of Standards and Technology; September 1988)

[Kramer7] Kramer, Thomas R.; *Contour Outlines*; NISTIR 4586; National Institute of Standards and Technology; October 1989

[Kramer8] Kramer, Thomas R.; *The Off-Line Programming System (OLPS): A Prototype STEP-Based NC-Program Generator*, proceedings of a seminar Product Data Exchange for the 1990's; New Orleans, Louisiana; NCGA; February 1991; Vol. 2

[Kramer9] Kramer, Thomas R.; *A Library of Material Removal Shape Element Volumes (MRSEVs)*; NISTIR 4809; National Institute of Standards and Technology; March 1992

[Kramer10] Kramer, Thomas R.; *Issues Concerning Material Removal Shape Element Volumes (MRSEVs)*; International Journal of Computer Integrated Manufacturing; Vol. 7, No. 3; 1994; pp. 139-151; (also published as NISTIR 4804; National Institute of Standards and Technology; March 1992*)*

[Kramer11] Kramer, Thomas R.; Proctor, Frederick M.; Michaloski, John; *The NIST RS274/NGC Interpreter Version 1*; NISTIR 5416; National Institute of Standards and Technology, Gaithersburg, MD; April 1994

[Kramer12] Kramer, Thomas R.; Proctor, Frederick M.; *The NIST RS274KT Interpreter*; NISTIR 5738; National Institute of Standards and Technology, Gaithersburg, MD; October 1995

[Kramer13] Kramer, Thomas R.; Proctor, Frederick M.; *The NIST RS274/NGC Interpreter Version 2*; NISTIR 5739; National Institute of Standards and Technology, Gaithersburg, MD; October 1995

[Kramer14] Kramer, Thomas R.; Proctor, Frederick M.; *The NIST RS274/VGER Interpreter*; NISTIR 5754; National Institute of Standards and Technology, Gaithersburg, MD; November 1995

[Kramer15] Kramer, Thomas R.; Proctor, Federick M.; *Feature-based Control of a Machining Center*; NISTIR 5926; National Institute of Standards and Technology; December 1996

[Kramer16] 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; April 1997

[Kramer17] Kramer, Thomas R.; Proctor, Frederick M.; Rippey, William G.; Scott, Harry; *The NIST DMIS Interpreter Version 2*; NISTIR 6252; National Institute of Standards and Technology, Gaithersburg, MD; October 1998

[Kramer18] Kramer, Thomas R.; Proctor, Frederick M.; Messina, Elena; *The NIST RS274/NGC Interpreter - Version 3*; NISTIR 6556; National Institute of Standards and Technology, Gaithersburg, MD; August 2000

[Kramer19] Kramer, Thomas R.; Huang, Hui-Minh; Messina, Elena; Proctor, Frederick M.; Scott, Harry; *A Feature-Based Inspection and Machining System*; Computer-Aided Design, Vol. 33, Issue 9; pp. 653-669; August 2001

[Messina] Messina, E.; Horst, J.; Kramer, T.; Huang, H.; Tsai, T.; Amatucci, E.; *A Knowledge-Based Inspection Workstation*; Proceedings of the IEEE International Conference on Information, Intelligence, and Systems; Bethesda, MD; November 1999

[NCMS] National Center for Manufacturing Sciences; *The Next Generation Controller Part Programming Functional Specification (RS-274/NGC)*; Draft; NCMS; August 1994

[Proctor1] 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

[Proctor2] Proctor, Frederick M.; Kramer, Thomas R.; *A Feature-based Machining System using STEP*; Proceedings of SPIE Conference on Sensors and Controls for Intelligent Machining, Agile Manufacturing, and Mechatronics; November 1998; Boston, Massachusetts; SPIE Vol. 3518; pp. 156-163

[Proctor3] Proctor, Frederick M; Kramer, Thomas R.; *Canonical Machining Commands*; NISTIR 5970; National Institute of Standards and Technology; January 1997

[Shackleford] Shackleford, Will; *Real-Time Control Systems Library Documents*; http://isd.cme.nist.gov/proj/rcs_lib; 1996

[STEPTools1] STEP Tools, Inc.; *STEP Utilities Reference Manual*; STEP Tools, Inc.; Troy, New York; 1994

[STEPTools2] STEP Tools, Inc.; *ROSE Library Reference Manual*; STEP Tools, Inc.; Troy, New York; 1996

[Wallace] Wallace, Sarah; Senehi, M. K.; Barkmeyer, Ed; Ray, Steven; Wallace, Evan K.; *Control Entity Interface Specification*; NISTIR 5272; National Institute of Standards and Technology; September 1993

Appendix A An Example

This appendix provides a simple example of FBICS in action. A part with six features is used. The part is named “part1” and shown in Figure 22. Part1 has no function. Its shape is designed to demonstrate many of the capabilities of FBICS. In this example, the Fbics_Task2 process is simulating a machining center equipped with a touch probe and able to carry out both low-level machining commands and low-level inspection commands. The simulation of probing assumes everything is perfect; the simulated actual data for probe points is identical to the nominal data.

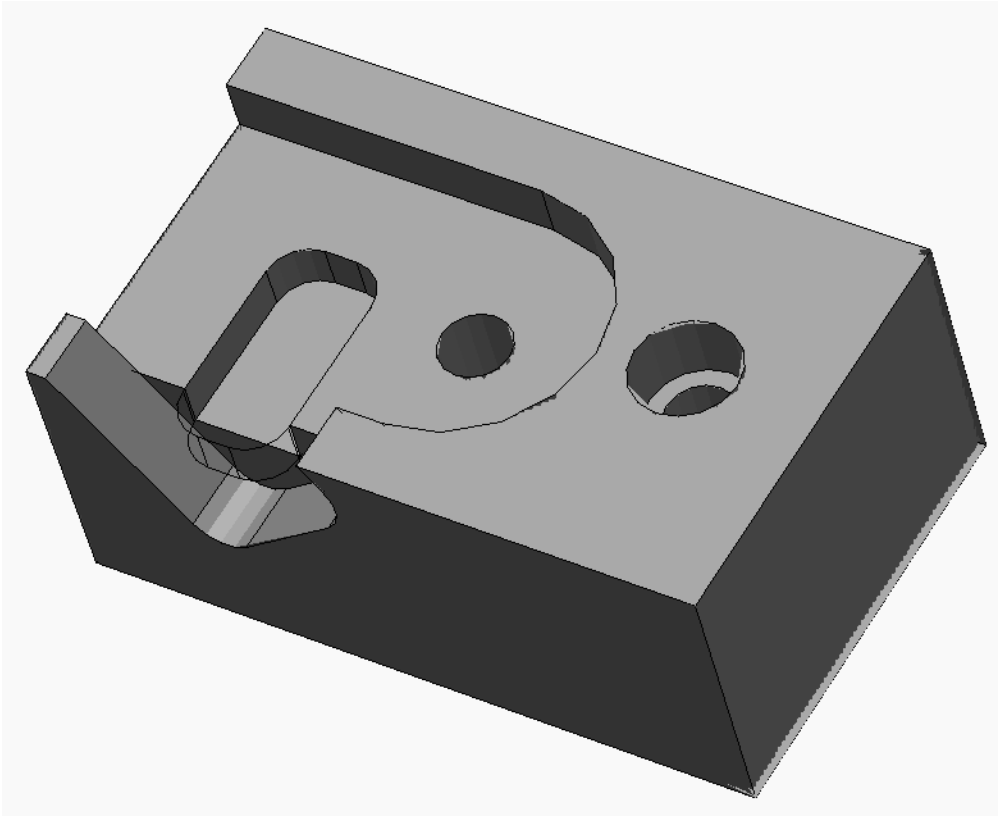


Figure 22. Part1

Part1 has several interesting aspects.

First, part1 has five features that must be machined from the top and one feature that must be machined from the side, so at least two setups are required.

Second, if the part is machined by approaching it from the top first, the lower pocket on top will not be accessible for machining (a wire frame view of the entire lower pocket is shown in Figure 22). FBICS detects this and, consequently, decides to run first the setup that makes the side pocket. This is manifested in the cell-level stage-one plan (Appendix A.4) by a sequential, rather than parallel, ordering of the two run_setup nodes, with the run_setup node that includes machining the side pocket coming first.

Third, FBICS sees that all the features to be made from the top can be made in a single setup, but

the counterbored hole must be drilled before it is counterbored, and the upper pocket must be made before either the lower pocket, or the hole at the bottom of the upper pocket. This leads to the following structure for the work-level stage-one plan for the second setup (Appendix A.11).

1. Start.
2. In any order, mill the top pocket and drill the top hole.
3. In any order, counterbore the top hole, drill the hole at the bottom of the upper pocket, and mill the pocket at the bottom of the top pocket.
4. End.

Fourth, the top hole and the top pocket are toleranced. The options in effect when FBICS did the work (see Appendix B) indicate that any feature with a tolerance should be inspected, so these two features are inspected after they are machined. Also, the tolerance on the corner radius of the top pocket (0.04 mm) is tighter than the shop options default tolerance (0.1 mm), so the Work Planner decides during stage-one planning to perform adaptive milling to make the pocket. The inspections and adaptive milling show up first in the work-level stage-one plan for the second setup (Appendix A.11). This is reflected later by several additional executable operation files (and corresponding NC code and DMIS code files) being generated at execution time.

To prepare the files shown in this appendix, FBICS was started, and two commands were given to the Cell Controller:

1. plan_part_machine(data/parts/part1/out.stp, OFF, data/parts/part1/in.stp, data/parts/part1/plans, data/parts/part1/feats, data/parts/part1/setups, 2)
2. run_part_plan2(data/parts/part1/plans)

Running on a Sun Ultra 60 with 512 Mbytes of memory, the first command took 40 seconds (of wall clock time) to execute. The second command took 50 seconds to execute.

The design file part1/out.stp shown in Appendix A.1 was written by hand. All the rest of the files in Appendix A were written by FBICS as it ran. The first ten files below of these were generated in response to the first command above. The files setups_1.stp and setups_2.stp were generated but are not shown here since they are almost identical to setups_1_keep.stp and setups_2_keep.stp, respectively. Comments, extra spaces, and some header strings have been deleted from the files in this appendix, but they are otherwise exactly as written by FBICS.

The reading and writing activity associated with the first ten files may be outlined in chronological order as follows. Details of how the Cell Planner and Work Planner operate are discussed in Section 7 and Section 8.

1. Cell Planner reads part_out file.
2. Cell Planner writes part_in file.
3. Cell Planner writes stage-one cell-level plan file.
4. Cell Planner writes first setup file and first features file.
5. Cell Planner writes second setup file and second features file.
6. Cell Planner reads stage-one cell-level plan file to start stage-two planning.
7. Cell Planner writes intermediate workpiece file, rewrites first setup file, and tells Work Planner to plan for first setup.
8. Work Planner reads rewritten first setup file.
9. Work Planner reads intermediate workpiece file, first features file, and part_in file.
10. Work Planner writes first stage-one work-level plan.

11. Cell Planner rewrites second setup file, and tells Work Planner to plan for second setup.
12. Work Planner reads rewritten second setup file.
13. Work Planner reads part_out file, second features file, and intermediate workpiece file.
14. Work Planner writes second stage-one work-level plan.
15. Cell Planner writes stage-two cell-level plan file.

The 12 files in Table 2, 68 files in Table 3, and one file in Appendix A.14, shown below, were generated in response to the second command. Table 2 includes the files written during the first setup of the part, and Table 3 includes the files written during the second setup. In both tables, the files on the left side are executable operation files generated by the Work Planner during execution of the work-level stage-one plan for the setup, and the files on the right side are NC code or DMIS code files generated by the Task Planner for carrying out the operations. The files were generated in the order shown in the tables, top to bottom, with the code file on any row being generated before the executable operation file on the next line down.

All the NC code and DMIS code files shown were interpreted by either the DMIS interpreter or the RS274/NGC interpreter in Fbics_Task2. This resulted in several hundred low-level machining and inspection commands being printed in the terminal window for that process, but they were not printed to a file. The RS274/NGC interpreter printed the file rs274ngc.var when it exited, but that is not shown here. The DMIS interpreter printed the output.dms file shown in Appendix A.14, and the Task Planner read part of it, as described in that section of the appendix.

The last file in this appendix is a sample graphics file. It is the “fbics_part_in_picture” graphics file for the block that is the base shape for part1. The Modeler wrote and rewrote this file and the other five graphics files many times. Most of the graphics files are five to ten times as large as the one shown.

A.1 Part_out Design File

This is the file out.stp giving the design of part1. It was written by hand.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION ((),‘1’);
FILE_NAME (‘out.stp’, ‘2001-08’, (‘T. Kramer’), (‘NIST’), ‘hand-written’, ‘NA’, ‘OK’);
FILE_SCHEMA ((‘ARM224’));
ENDSEC;

DATA;
  #1 = DIRECTION_ELEMENT ((0.0, 0.0, 1.0));
  #2 = DIRECTION_ELEMENT ((1.0, 0.0, 0.0));
  #3 = LOCATION_ELEMENT ((62.5, 37.5, 0.0));
  #4 = ORIENTATION (#1, #2, #3);
  #5 = NUMERIC_PARAMETER (‘block Y dimension’, 75.0, ‘mm’);
  #6 = NUMERIC_PARAMETER (‘block X dimension’, 125.0, ‘mm’);
  #7 = NUMERIC_PARAMETER (‘block Z dimension’, 50.0, ‘mm’);
  #8 = BLOCK_BASE_SHAPE (#7, #4, #5, #6);
  #9 = NUMERIC_PARAMETER (‘twist drill tip angle’, 118.0, ‘degrees’);
 #10 = NUMERIC_PARAMETER (‘floor radius for pockets’, 0.0, ‘mm’);
```

```
#11 = CONICAL_HOLE_BOTTOM (.T., #9, $); /* twist drill bottom */
#12 = DIRECTION_ELEMENT ((0.0, 1.0, 0.0));
#20 = PLUS_MINUS_VALUE (0.04, -0.04, 5.0);
#21 = NUMERIC_PARAMETER_WITH_TOLERANCE('drill hole diameter', 12.7, 'mm', #20);
#22 = CIRCULAR_CLOSED_PROFILE (#21);
#23 = NUMERIC_PARAMETER ('drill hole depth', 25.0, 'mm');
#24 = LOCATION_ELEMENT ((100.0, 40.0, 50.0));
#25 = ORIENTATION (#1, #2, #24);
#26 = ROUND_HOLE (#25, '12.7 hole in top', #11, $, #22, #23);
#27 = SHAPE_ASPECT ((), (), #26);
#29 = PLUS_MINUS_VALUE (0.04, -0.04, 5.0);
#31 = NUMERIC_PARAMETER_WITH_TOLERANCE('corner_radius of profile', 25.0, 'mm', #29);
#32 = NUMERIC_PARAMETER ('width of profile', 50.0, 'mm');
#33 = NUMERIC_PARAMETER ('length of profile', 110.0, 'mm');
#34 = LOCATION_ELEMENT ((30.0, 37.5, 50.0));
#35 = ORIENTATION (#1, #2, #34);
#36 = RECTANGULAR_CLOSED_PROFILE (#31, #32, #33);
#37 = LOCATION_ELEMENT ((0.0, 0.0, -10.0));
#38 = PLANAR_POCKET_BOTTOM_CONDITION (.T., #37, #1, #10);
#39 = RECTANGULAR_CLOSED_POCKET (#35, 'upper pocket on top', #38, $, #36);
#40 = SHAPE_ASPECT ((), (), #39);
#41 = NUMERIC_PARAMETER('corner_radius of profile', 7.0, 'mm');
#42 = NUMERIC_PARAMETER ('width of profile', 20.0, 'mm');
#43 = NUMERIC_PARAMETER ('length of profile', 40.0, 'mm');
#44 = LOCATION_ELEMENT ((30.0, 25.0, 40.0));
#45 = ORIENTATION (#1, #12, #44);
#46 = RECTANGULAR_CLOSED_PROFILE (#41, #42, #43);
#47 = LOCATION_ELEMENT ((0.0, 0.0, -8.0));
#48 = PLANAR_POCKET_BOTTOM_CONDITION (.T., #47, #1, #10);
#49 = RECTANGULAR_CLOSED_POCKET (#45, 'lower pocket on top', #48, $, #46);
#50 = SHAPE_ASPECT ((), (), #49);
#61 = NUMERIC_PARAMETER('drill hole diameter', 12.7, 'mm');
#62 = CIRCULAR_CLOSED_PROFILE (#61);
#63 = NUMERIC_PARAMETER ('drill hole depth', 25.0, 'mm');
#64 = LOCATION_ELEMENT ((60.0, 37.5, 40.0));
#65 = ORIENTATION (#1, #2, #64);
#66 = ROUND_HOLE (#65, '12.7 hole in top pocket', #11, $, #62, #63);
#67 = SHAPE_ASPECT ((), (), #66);
#80 = FLAT_HOLE_BOTTOM (.T.); /* counterbore hole bottom */
#81 = NUMERIC_PARAMETER ('counterbore hole diameter', 19.0, 'mm');
#82 = CIRCULAR_CLOSED_PROFILE (#81);
#83 = NUMERIC_PARAMETER ('counterbore hole depth', 12.5, 'mm');
#84 = LOCATION_ELEMENT ((100.0, 40.0, 50.0));
#85 = ORIENTATION (#1, #2, #84);
#86 = ROUND_HOLE (#85, 'counterbore1', #80, $, #82, #83);
#87 = COUNTERBORE_HOLE ('19.0 counterbore hole in top', #26, #86);
#88 = SHAPE_ASPECT ((), (), #87);
#89 = DIRECTION_ELEMENT ((0.0, -1.0, 0.0));
#90 = DIRECTION_ELEMENT ((-0.6,  0.0, 0.8));
#91 = NUMERIC_PARAMETER('corner_radius of profile', 7.0, 'mm');
#92 = NUMERIC_PARAMETER ('width of profile', 38.0, 'mm');
#93 = NUMERIC_PARAMETER ('length of profile', 50.0, 'mm');
#94 = LOCATION_ELEMENT ((29.0, 0.0, 48.0));
#95 = ORIENTATION (#89, #90, #94);
```

```
 #96 = RECTANGULAR_CLOSED_PROFILE (#91, #92, #93);
 #97 = LOCATION_ELEMENT ((0.0, 0.0, -12.5));
 #98 = PLANAR_POCKET_BOTTOM_CONDITION (.T., #97, #1, #10);
 #99 = RECTANGULAR_CLOSED_POCKET (#95, 'pocket on side', #98, $, #96);
#100 = SHAPE_ASPECT ((), (), #99);
#110 = SHAPE ((#27, #40, #50, #67, #88, #100), #8, ());
#120 = MATERIAL('aluminum', 'soft aluminum', $, (), ());
#200 = PART('out', 'rev1', '', 'simple part', 'insecure', (), #110,
        (), (), (), $, (), (#120), (), ());
ENDSEC;
END-ISO-10303-21;
```

A.2 Part_in

This is the part_in file, in.stp, generated by the Cell Controller as it started work.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((''), '2;1');
FILE_NAME('in', '2003-08', (''), (''), '', '', '');
FILE_SCHEMA (('ARM224'));
ENDSEC;

DATA;
#10=MATERIAL('aluminum','soft aluminum',$,(),());
#11=PART('out','rev1','','simple part','insecure',(),#12,(),(),(),$,(),(#10),(),());
#12=SHAPE((),#20,());
#13=LOCATION_ELEMENT((62.5,37.5,0.));
#14=DIRECTION_ELEMENT((0.,0.,1.));
#15=DIRECTION_ELEMENT((1.,0.,0.));
#16=ORIENTATION(#14,#15,#13);
#17=NUMERIC_PARAMETER('block Z dimension',50.,'mm');
#18=NUMERIC_PARAMETER('block Y dimension',75.,'mm');
#19=NUMERIC_PARAMETER('block X dimension',125.,'mm');
#20=BLOCK_BASE_SHAPE(#17,#16,#18,#19);
ENDSEC;
END-ISO-10303-21;
```

A.3 Intermediate Workpiece Shape

This is the file out_keep_1.stp representing the shape of the workpiece as it came out of the first fixturing and as it went into the second fixturing. The shape includes only the base shape and the pocket on the side of part1.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION( (''), '2;1');
FILE_NAME( 'out_keep_1','2003-08', (''), (''), '', '', '');
FILE_SCHEMA (('ARM224'));
ENDSEC;

DATA;
#10=MATERIAL('aluminum','soft aluminum',$,(),());
```

```
#11=BLOCK_BASE_SHAPE(#17,#30,#18,#19);
#12=RECTANGULAR_CLOSED_PROFILE(#14,#15,#16);
#13=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm');
#14=NUMERIC_PARAMETER('corner_radius of profile',7.,'mm');
#15=NUMERIC_PARAMETER('width of profile',38.,'mm');
#16=NUMERIC_PARAMETER('length of profile',50.,'mm');
#17=NUMERIC_PARAMETER('block Z dimension',50.,'mm');
#18=NUMERIC_PARAMETER('block Y dimension',75.,'mm');
#19=NUMERIC_PARAMETER('block X dimension',125.,'mm');
#20=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#22,#26,#13);
#21=LOCATION_ELEMENT((29.,0.,48.));
#22=LOCATION_ELEMENT((0.,0.,-12.5));
#23=LOCATION_ELEMENT((62.5,37.5,0.));
#24=DIRECTION_ELEMENT((0.,-1.,0.));
#25=DIRECTION_ELEMENT((-0.6,0.,0.8));
#26=DIRECTION_ELEMENT((0.,0.,1.));
#27=DIRECTION_ELEMENT((0.,0.,1.));
#28=DIRECTION_ELEMENT((1.,0.,0.));
#29=ORIENTATION(#24,#25,#21);
#30=ORIENTATION(#27,#28,#23);
#31=RECTANGULAR_CLOSED_POCKET(#29,'pocket on side',#20,$,#12);
#32=SHAPE_ASPECT((),(),#31);
#33=SHAPE((#32),#11,());
#34=PART('out','rev1','','simple part','insecure',(),#33,(),(),(),$,(),(#10),(),());
ENDSEC;
END-ISO-10303-21;
```

A.4 Cell-level Stage-one Plan

This is the file plans_1cell.stp written by the Cell Controller when it completed stage-one planning.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((''),'2;1');
FILE_NAME('plans_1cell', '2003-08', (''), (''), '', '', '');
FILE_SCHEMA (('FBICS_COMBO','FBICS_ALPS'));
ENDSEC;

DATA;
#10=INTERNAL_REAL(#18,'setup_total',.T.,$,2.);
#11=END_PLAN_NODE(#18,4,$,$,(),());
#12=RUN_SETUP(#18,2,$,$,(#13),(),1,'data/parts/part1/setups_1.stp',());
#13=RUN_SETUP(#18,3,$,$,(#11),(),1,'data/parts/part1/setups_2.stp',());
#14=START_PLAN_NODE(#18,1,$,$,(#12),());
#15=INTERNAL_STRING(#18,'length_units',.T.,'mm');
#16=INTERNAL_STRING(#18,'part_in_file_name',.T.,'data/parts/part1/in.stp');
#17=INTERNAL_STRING(#18,'part_out_file_name',.T.,'data/parts/part1/out.stp');
#18=PLAN('data/parts/part1/plans_1cell','version 1',(),(#15,#16,#17,#10),
    'CELL','a part','generated by FBICS',(#14,#12,#13,#11));
ENDSEC;
END-ISO-10303-21;
```

A.5 Cell-level Stage-two Plan

This is the file plans_2cell.stp written by the Cell Controller when it completed stage-two planning.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((‘’), ‘2;1’);
FILE_NAME(‘plans_2cell’, ‘2003-08’, (‘’), (‘’), ‘’, ‘’, ‘’);
FILE_SCHEMA ((‘FBICS_COMBO’));
ENDSEC;

DATA;
#10=ONE_OPERATION(7,’data/parts/part1/plans_1_1work.stp’);
#11=ONE_OPERATION(7,’data/parts/part1/plans_2_1work.stp’);
#12=OPERATION_PLAN((#10,#11));
ENDSEC;
END-ISO-10303-21;
```

A.6 Setup File for First Setup

This is the file setups_1_keep.stp written by the Cell Controller when it performed stage-two planning for the first setup. This is nearly identical to the file setups_1.stp (not shown) written by the Cell Controller when it completed stage-one planning. Only two strings have been modified.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((‘’), ‘2;1’);
FILE_NAME(‘setups_1_keep’, ‘2003-08’, (‘’), (‘’), ‘’, ‘’, ‘’);
FILE_SCHEMA ((‘SETUP’));
ENDSEC;

DATA;
#10=BOX_SETUP(#23,#24);
#11=DIRECTION_SETUP(0.,1.,0.);
#12=DIRECTION_SETUP(1.,0.,0.);
#13=DIRECTION_SETUP(0.,1.,0.);
#14=DIRECTION_SETUP(1.,0.,0.);
#15=DIRECTION_SETUP(0.,1.,0.);
#16=DIRECTION_SETUP(1.,0.,0.);
#17=DIRECTION_SETUP(0.,0.,1.);
#18=DIRECTION_SETUP(1.,0.,0.);
#19=CARTESIAN_POINT_SETUP(0.,0.,75.);
#20=CARTESIAN_POINT_SETUP(0.,0.,75.);
#21=CARTESIAN_POINT_SETUP(0.,0.,75.);
#22=CARTESIAN_POINT_SETUP(-39.1,0.,0.);
#23=CARTESIAN_POINT_SETUP(0.,0.,0.);
#24=CARTESIAN_POINT_SETUP(125.,50.,75.);
#25=AXIS2_PLACEMENT_SETUP(#19,#11,#12);
#26=AXIS2_PLACEMENT_SETUP(#20,#13,#14);
#27=AXIS2_PLACEMENT_SETUP(#21,#15,#16);
#28=AXIS2_PLACEMENT_SETUP(#22,#17,#18);
#29=FILE_NAMES(’data/parts/part1/setups_1_keep.stp’,’data/parts/part1/out_keep_1.stp’,
```

```
    'data/large_vise_half.stp','data/parts/part1/plans_1.stp','data/parts/part1/feats_1.stp','data/parts/part1/in.stp');
#30=SETUP_SPEC(#29,#25,#26,#27,#28,#10,.F.,'mm');
ENDSEC;
END-ISO-10303-21;
```

A.7 Setup File for Second Setup

This is the file setups_2_keep.stp written by the Cell Controller when it performed stage-two planning for the second setup. This is nearly identical to the file setups_2.stp (not shown) written by the Cell Controller when it completed stage-one planning. Only two strings have been modified.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((''), '2;1');
FILE_NAME('setups_2_keep', '2003-08', (''), (''), '', '', '');
FILE_SCHEMA (('SETUP'));
ENDSEC;

DATA;
#10=BOX_SETUP(#23,#24);
#11=DIRECTION_SETUP(0.,0.,1.);
#12=DIRECTION_SETUP(1.,0.,0.);
#13=DIRECTION_SETUP(0.,0.,1.);
#14=DIRECTION_SETUP(1.,0.,0.);
#15=DIRECTION_SETUP(0.,0.,1.);
#16=DIRECTION_SETUP(1.,0.,0.);
#17=DIRECTION_SETUP(0.,0.,1.);
#18=DIRECTION_SETUP(1.,0.,0.);
#19=CARTESIAN_POINT_SETUP(0.,0.,0.);
#20=CARTESIAN_POINT_SETUP(0.,0.,0.);
#21=CARTESIAN_POINT_SETUP(0.,0.,0.);
#22=CARTESIAN_POINT_SETUP(-39.1,0.,0.);
#23=CARTESIAN_POINT_SETUP(0.,0.,0.);
#24=CARTESIAN_POINT_SETUP(125.,75.,50.);
#25=AXIS2_PLACEMENT_SETUP(#19,#11,#12);
#26=AXIS2_PLACEMENT_SETUP(#20,#13,#14);
#27=AXIS2_PLACEMENT_SETUP(#21,#15,#16);
#28=AXIS2_PLACEMENT_SETUP(#22,#17,#18);
#29=FILE_NAMES('data/parts/part1/setups_2_keep.stp','data/parts/part1/out.stp','data/large_vise_half.stp',
    'data/parts/part1/plans_2.stp','data/parts/part1/feats_2.stp','data/parts/part1/out_keep_1.stp');
#30=SETUP_SPEC(#29,#25,#26,#27,#28,#10,.F.,'mm');
ENDSEC;
END-ISO-10303-21;
```

A.8 Features File for First Setup

This is the features file for the first setup, feats_1.stp.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((''), '2;1');
FILE_NAME('feats_1', '2003-08', (''), (''), '', '', '');
```

```
FILE_SCHEMA ((‘ARM224’));
ENDSEC;

DATA;
#10=MATERIAL(’aluminum’,’soft aluminum’,$,(),());
#11=BLOCK_BASE_SHAPE(#17,#29,#18,#19);
#12=RECTANGULAR_CLOSED_PROFILE(#14,#15,#16);
#13=NUMERIC_PARAMETER(’floor radius for pockets’,0.,’mm’);
#14=NUMERIC_PARAMETER(’corner_radius of profile’,7.,’mm’);
#15=NUMERIC_PARAMETER(’width of profile’,38.,’mm’);
#16=NUMERIC_PARAMETER(’length of profile’,50.,’mm’);
#17=NUMERIC_PARAMETER(’block Z dimension’,50.,’mm’);
#18=NUMERIC_PARAMETER(’block Y dimension’,75.,’mm’);
#19=NUMERIC_PARAMETER(’block X dimension’,125.,’mm’);
#20=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#22,#26,#13);
#21=LOCATION_ELEMENT((29.,0.,48.));
#22=LOCATION_ELEMENT((0.,0.,-12.5));
#23=LOCATION_ELEMENT((62.5,37.5,0.));
#24=DIRECTION_ELEMENT((0.,-1.,0.));
#25=DIRECTION_ELEMENT((-0.6,0.,0.8));
#26=DIRECTION_ELEMENT((0.,0.,1.));
#27=DIRECTION_ELEMENT((1.,0.,0.));
#28=ORIENTATION(#24,#25,#21);
#29=ORIENTATION(#26,#27,#23);
#30=RECTANGULAR_CLOSED_POCKET(#28,’pocket on side’,#20,$,#12);
#31=SHAPE_ASPECT((),(),#30);
#32=SHAPE((#31),#11,());
#33=PART(’out’,’rev1’,’’,’simple part’,’insecure’,(),#32,(),(),(),$,(),(#10),(),());
ENDSEC;
END-ISO-10303-21;
```

A.9 Features File for Second Setup

This is the features file for the second setup, feats_2.stp.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((‘’), ‘2;1’);
FILE_NAME(‘feats_2’, ‘2003-08’, (‘’), (‘’), ‘’, ‘’, ‘’);
FILE_SCHEMA ((‘ARM224’));
ENDSEC;

DATA;
#10=MATERIAL(’aluminum’,’soft aluminum’,$,(),());
#11=BLOCK_BASE_SHAPE(#39,#59,#40,#41);
#12=FLAT_HOLE_BOTTOM(.T.);
#13=COUNTERBORE_HOLE(’19.0 counterbore hole in top’,#60,#61);
#14=RECTANGULAR_CLOSED_PROFILE(#23,#30,#31);
#15=RECTANGULAR_CLOSED_PROFILE(#36,#37,#38);
#16=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#45,#51,#29);
#17=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#49,#51,#29);
#18=RECTANGULAR_CLOSED_POCKET(#55,’upper pocket on top’,#16,$,#14);
#19=RECTANGULAR_CLOSED_POCKET(#58,’lower pocket on top’,#17,$,#15);
```

```
#20=PLUS_MINUS_VALUE(0.04,-0.04,5.);
#21=PLUS_MINUS_VALUE(0.04,-0.04,5.);
#22=NUMERIC_PARAMETER_WITH_TOLERANCE('drill hole diameter',12.7,'mm',#20);
#23=NUMERIC_PARAMETER_WITH_TOLERANCE('corner_radius of profile',25.,'mm',#21);
#24=CIRCULAR_CLOSED_PROFILE(#22);
#25=CIRCULAR_CLOSED_PROFILE(#32);
#26=CIRCULAR_CLOSED_PROFILE(#34);
#27=NUMERIC_PARAMETER('twist drill tip angle',118.,'degrees');
#28=NUMERIC_PARAMETER('drill hole depth',25.,'mm');
#29=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm');
#30=NUMERIC_PARAMETER('width of profile',50.,'mm');
#31=NUMERIC_PARAMETER('length of profile',110.,'mm');
#32=NUMERIC_PARAMETER('counterbore hole diameter',19.,'mm');
#33=NUMERIC_PARAMETER('counterbore hole depth',12.5,'mm');
#34=NUMERIC_PARAMETER('drill hole diameter',12.7,'mm');
#35=NUMERIC_PARAMETER('drill hole depth',25.,'mm');
#36=NUMERIC_PARAMETER('corner_radius of profile',7.,'mm');
#37=NUMERIC_PARAMETER('width of profile',20.,'mm');
#38=NUMERIC_PARAMETER('length of profile',40.,'mm');
#39=NUMERIC_PARAMETER('block Z dimension',50.,'mm');
#40=NUMERIC_PARAMETER('block Y dimension',75.,'mm');
#41=NUMERIC_PARAMETER('block X dimension',125.,'mm');
#42=CONICAL_HOLE_BOTTOM(.T.,#27,$);
#43=LOCATION_ELEMENT((100.,40.,50.));
#44=LOCATION_ELEMENT((30.,37.5,50.));
#45=LOCATION_ELEMENT((0.,0.,-10.));
#46=LOCATION_ELEMENT((100.,40.,50.));
#47=LOCATION_ELEMENT((60.,37.5,40.));
#48=LOCATION_ELEMENT((30.,25.,40.));
#49=LOCATION_ELEMENT((0.,0.,-8.));
#50=LOCATION_ELEMENT((62.5,37.5,0.));
#51=DIRECTION_ELEMENT((0.,0.,1.));
#52=DIRECTION_ELEMENT((1.,0.,0.));
#53=DIRECTION_ELEMENT((0.,1.,0.));
#54=ORIENTATION(#51,#52,#43);
#55=ORIENTATION(#51,#52,#44);
#56=ORIENTATION(#51,#52,#46);
#57=ORIENTATION(#51,#52,#47);
#58=ORIENTATION(#51,#53,#48);
#59=ORIENTATION(#51,#52,#50);
#60=ROUND_HOLE(#54,'12.7 hole in top',#42,$,#24,#28);
#61=ROUND_HOLE(#56,'counterbore1',#12,$,#25,#33);
#62=ROUND_HOLE(#57,'12.7 hole in top pocket',#42,$,#26,#35);
#63=SHAPE_ASPECT((),(),#60);
#64=SHAPE_ASPECT((),(),#18);
#65=SHAPE_ASPECT((),(),#13);
#66=SHAPE_ASPECT((),(),#62);
#67=SHAPE_ASPECT((),(),#19);
#68=SHAPE((#63,#64,#65,#66,#67),#11,());
#69=PART('out','rev1','','simple part','insecure',(),#68,(),(),(),$,(),(#10),(),());
ENDSEC;
END-ISO-10303-21;
```

A.10 Work-level Stage-one Plan for First Setup

This is the work-level stage-one plan for the first setup, plans_1_1work.stp.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((‘’), ‘2;1’);
FILE_NAME(‘plans_1_1work’, ‘2003-08’, (‘’), (‘’), ‘’, ‘’, ‘’);
FILE_SCHEMA ((‘FBICS_COMBO’,’FBICS_ALPS’));
ENDSEC;

DATA;
#10=END_PLAN_NODE(#15,3,$,$,(),());
#11=FINISH_MILL(#15,2,’pocket on side’,$,(#10),(),1,(),’END-MILL-10.0-2
    ’,0,4366,436.606716536686,.T.,.F.,5.,5.);
#12=START_PLAN_NODE(#15,1,$,$,(#11),());
#13=INTERNAL_STRING(#15,’length_units’,.T.,’mm’);
#14=INTERNAL_STRING(#15,’setup_file_name’,.T.,’data/parts/part1/setups_1_keep.stp’);
#15=PLAN(’data/parts/part1/plans_1.stp’,’version 1’,(),(#13,#14),’WORK’,
    ’a part’,’generated by FBICS’,(#12,#11,#10));
ENDSEC;
END-ISO-10303-21;
```

A.11 Work-level Stage-one Plan for Second Setup

This is the work-level stage-one plan for the second setup, plans_2_1work.stp.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION((‘’), ‘2;1’);
FILE_NAME(‘plans_2_1work’, ‘2003-08’, (‘’), (‘’), ‘’, ‘’, ‘’);
FILE_SCHEMA ((‘FBICS_COMBO’,’FBICS_ALPS’));
ENDSEC;

DATA;
#10=END_PLAN_NODE(#25,13,$,$,(),());
#11=FINISH_MILL(#25,12,’lower pocket on top’,$,(#19),(),1,(),
    ’END-MILL-10.0-2’,4,4366,436.606716536686,.T.,.F.,5.,5.);
#12=COUNTERBORING(#25,10,’19.0 counterbore hole in top’,$,(#19),(),1,(),
    ’END-MILL-19.0-2’,2,2297,436.606716536686,.T.,.F.);
#13=FINISH_MILL_ADAPTIVE(#25,6,’upper pocket on top’,$,(#15),(),1,(),’END-MILL-1.0-2’,
    1,1718,436.606716536686,.T.,.F.,12.7,12.7,’PROBE-20.0-4.0’,1000.,.MEDIUM.);
#14=INSPECT_FEATURE_GEOMETRY(#25,5,’inspect 12.7 hole in top’,$,(#18),(),
    1,.MEDIUM.,(),’PROBE-20.0-4.0’,0,1000.);
#15=INSPECT_FEATURE_GEOMETRY(#25,7,’inspect upper pocket on top’,$,(#18),
    (),1,.MEDIUM.,(),’PROBE-20.0-4.0’,1,1000.);
#16=TWIST_DRILLING(#25,4,’12.7 hole in top’,$,(#14),(),1,(),
    ’DRILL-0.5-2’,0,2100,133.407607830654,.T.,.F.,12.7);
#17=TWIST_DRILLING(#25,11,’12.7 hole in top pocket’,$,(#19),(),1,(),
    ’DRILL-0.5-2’,3,2100,133.407607830654,.T.,.F.,12.7);
#18=PATH_JOIN_NODE(#25,3,$,$,(#21),());
#19=PATH_JOIN_NODE(#25,9,$,$,(#10),());
#20=PARAMETERIZED_SPLIT_NODE(#25,2,$,$,(#16,#13),(),0,.SPLIT_TIMING_SERIAL.);
```

```
#21=PARAMETERIZED_SPLIT_NODE(#25,8,$,$,(#12,#17,#11),(),0,.SPLIT_TIMING_SERIAL.);
#22=START_PLAN_NODE(#25,1,$,$,(#20),());
#23=INTERNAL_STRING(#25,'length_units',.T.,'mm');
#24=INTERNAL_STRING(#25,'setup_file_name',.T.,'data/parts/part1/setups_2_keep.stp');
#25=PLAN('data/parts/part1/plans_2.stp','version 1',(),(#23,#24),'WORK','a part',
    'generated by FBICS',(#22,#20,#18,#16,#14,#13,#15,#21,#19,#12,#17,#11,#10));
ENDSEC;
END-ISO-10303-21;
```

A.12 Executable Operation and Code Files for First Setup

The table below shows on the left the 6 executable operation files generated by the Work Planner during the first setup for part1. On the right the table shows the corresponding NC code files generated by the Task Planner. The file names are given in *italics*. Only the data sections of the executable operation files are shown; the header sections are omitted.

Table 2.

Executable Operation File (data only)	NC or DMIS Code File
plans_1_1work_001.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL($,2,'pocket on side',$,($),($),1,(), 'END-MILL-10.0-2', 0,4366,436.606716536686,.T.,.F.,5.,5.); #12=START_NC_EX(#11);	*plans_1_1work_001.nc* (start program) G21 G92.2 G10 L2 P6 x0.000000 y0.000000 z0.000000 G59 G0 z81.000000 G0 x0.000000 y0.000000
plans_1_1work_002.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL($,2,'pocket on side',$,($),($),1,(), 'END-MILL-10.0-2', 0,4366,436.606716536686,.T.,.F.,5.,5.); #12=NC_CHANGE_EX(#11,'END-MILL-10.0-2-3');	*plans_1_1work_002.nc* M5 G0 G53 z-10.000000 T15 M6 G43 H15 (MSG,Load tool 15) M0 G0 z81.000000
plans_1_1work_003.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL($,2,'pocket on side',$,($),($),1,(), 'END-MILL-10.0-2', 0,4366,436.606716536686,.T.,.F.,5.,5.); #12=COOLANT_EX(#11,.T.,.F.);	*plans_1_1work_003.nc* M8

Table 2.

Executable Operation File (data only)	NC or DMIS Code File
plans_1_1work_004.stp #10=RECTANGULAR_CLOSED_PROFILE(#12,#13,#14); #11=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm'); #12=NUMERIC_PARAMETER('corner_radius of profile',7.,'mm'); #13=NUMERIC_PARAMETER('width of profile',38.,'mm'); #14=NUMERIC_PARAMETER('length of profile',50.,'mm'); #15=NUMERIC_PARAMETER('units_marker',0.,'mm'); #16=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#18, #21,#11); #17=LOCATION_ELEMENT((29.,48.,75.)); #18=LOCATION_ELEMENT((0.,0.,-12.5)); #19=DIRECTION_ELEMENT((0.,0.,1.)); #20=DIRECTION_ELEMENT((-0.6,0.8,0.)); #21=DIRECTION_ELEMENT((0.,0.,1.)); #22=ORIENTATION(#19,#20,#17); #23=RECTANGULAR_CLOSED_POCKET(#22, 'pocket on side',#16,$,#10); #24=FINISH_MILL($,2,'pocket on side',$,($),($),1,(), 'END-MILL-10.0-2',0, 4366,436.606716536686,.T.,.F.,5.,5.); #25=FINISH_MILL_EX(#24,#23,'END-MILL-10.0-2-3',1);	*plans_1_1work_004.nc* S4366.000000 M3 G0 x7.950000 y56.900000 G0 z80.000000 F218.303358 G1 z62.500000 F436.606717 G1 x31.650000 y25.300000 G1 x35.650000 y28.300000 G1 x11.950000 y59.900000 G1 x15.950000 y62.900000 G1 x39.650000 y31.300000 G1 x43.650000 y34.300000 G1 x19.950000 y65.900000 G1 x23.950000 y68.900000 G1 x47.650000 y37.300000 G0 z80.000000 G0 x29.200000 y69.400000 G0 z80.000000 G1 z67.500000 G3 x29.400000 y70.800000 z62.500000 r1.000000 G3 x26.600000 y71.200000 r2.000000 G1 x7.400000 y56.800000 G3 x7.000000 y54.000000 r2.000000 G1 x28.600000 y25.200000 G3 x31.400000 y24.800000 r2.000000 G1 x50.600000 y39.200000 G3 x51.000000 y42.000000 r2.000000 G1 x29.400000 y70.800000 G3 x28.000000 y71.000000 z67.500000 r1.000000 G0 z81.000000
plans_1_1work_005.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=END_PLAN_NODE($,3,$,$,(),($)); #12=COOLANT_EX(#11,.F.,.F.);	*plans_1_1work_005.nc* M9
plans_1_1work_006.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=END_PLAN_NODE($,3,$,$,(),($)); #12=END_NC_EX(#11);	*plans_1_1work_006.nc* M5 G0 G53 z0.000000 G0 G53 x100.000000 y100.000000 M2

A.13 Executable Operation and Code Files for Second Setup

The table below shows on the left the 34 executable operation files generated by the Work Planner during the second setup for part1. On the right the table shows the corresponding NC or DMIS code files generated by the Task Planner. The file names are given in *italics*. Only the data sections of the executable operation files are shown; the header sections are omitted. Numbers with many decimal places have been shortened to reduce the amount of space needed.

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_001.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=TWIST_DRILLING($,4,'12.7 hole in top',$,($),($),1, (),'DRILL-0.5-2',0,2100,133.407607830654,.T.,.F.,12.7); #12=START_NC_EX(#11);	*plans_2_1work_001.nc* (start program) G21 G92.2 G10 L2 P6 x0.000000 y0.000000 z0.000000 G59 G0 z56.000000 G0 x0.000000 y0.000000
plans_2_1work_002.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=TWIST_DRILLING($,4,'12.7 hole in top',$,($),($),1, (),'DRILL-0.5-2',0,2100,133.407607830654,.T.,.F.,12.7); #12=NC_CHANGE_EX(#11,'DRILL-0.5-2-3');	*plans_2_1work_002.nc* G0 G53 z-10.000000 T5 M6 G43 H5 (MSG,Load tool 5) M0 G0 z56.000000
plans_2_1work_003.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=TWIST_DRILLING($,4,'12.7 hole in top',$,($),($),1, (),'DRILL-0.5-2',0,2100,133.407607830654,.T.,.F.,12.7); #12=COOLANT_EX(#11,.T.,.F.);	*plans_2_1work_003.nc* M8
plans_2_1work_004.stp #10=PLUS_MINUS_VALUE(0.04,-0.04,5.); #11=NUMERIC_PARAMETER_WITH_TOLERANCE('drill hole diameter',12.7,'mm',#10); #12=CIRCULAR_CLOSED_PROFILE(#11); #13=NUMERIC_PARAMETER('twist drill tip angle', 118.,'degrees'); #14=NUMERIC_PARAMETER('drill hole depth',25.,'mm'); #15=NUMERIC_PARAMETER('units_marker',0.,'mm'); #16=CONICAL_HOLE_BOTTOM(.T.,#13,$); #17=LOCATION_ELEMENT((100.,40.,50.)); #18=DIRECTION_ELEMENT((0.,0.,1.)); #19=DIRECTION_ELEMENT((1.,0.,0.)); #20=ORIENTATION(#18,#19,#17); #21=ROUND_HOLE(#20,'12.7 hole in top',#16,$,#12,#14); #22=TWIST_DRILLING($,4,'12.7 hole in top',$,($),($),1, (),'DRILL-0.5-2',0,2100,133.407607830654,.T.,.F.,12.7); #23=TWIST_DRILLING_EX(#22,#21,'DRILL-0.5-2-3',1);	*plans_2_1work_004.nc* S2100.000000 M3 F133.407608 G83 x100.0000 y40.0000 z25.0000 r55.0000 q12.7000 G0 z56.000000
plans_2_1work_005.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=INSPECT_FEATURE_GEOMETRY($,5,'inspect 12.7 hole in top',$,($),($),1, .MEDIUM.,(),'PROBE-20.0-4.0',0,1000.); #12=COOLANT_EX(#11,.F.,.F.);	*plans_2_1work_005.nc* M9

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_006.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=INSPECT_FEATURE_GEOMETRY($,5,'inspect 12.7 hole in top',$,($),($),1, .MEDIUM.,(),'PROBE-20.0-4.0',0,1000.); #12=START_INSPECT_EX(#11);	*plans_2_1work_006.dmis* DMISMN/ 'FBICS DMIS program' FILNAM/ 'FBICS DMIS output' UNITS/ MM, ANGDEC
plans_2_1work_007.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=INSPECT_FEATURE_GEOMETRY($,5,'inspect 12.7 hole in top',$,($),($),1, .MEDIUM.,(),'PROBE-20.0-4.0',0,1000.); #12=NC_CHANGE_EX(#11,'PROBE6');	*plans_2_1work_007.nc* M5 G0 G53 z-10.000000 T19 M6 G43 H19 (MSG,Load tool 19) M0 G0 z56.000000
plans_2_1work_008.stp #10=PLUS_MINUS_VALUE(0.04,-0.04,5.); #11=NUMERIC_PARAMETER_WITH_TOLERANCE('drill hole diameter',12.7,'mm',#10); #12=CIRCULAR_CLOSED_PROFILE(#11); #13=NUMERIC_PARAMETER('twist drill tip angle',118.,'degrees'); #14=NUMERIC_PARAMETER('drill hole depth',25.,'mm'); #15=NUMERIC_PARAMETER('units_marker',0.,'mm'); #16=CONICAL_HOLE_BOTTOM(.T.,#13,$); #17=LOCATION_ELEMENT((100.,40.,50.)); #18=DIRECTION_ELEMENT((0.,0.,1.)); #19=DIRECTION_ELEMENT((1.,0.,0.)); #20=ORIENTATION(#18,#19,#17); #21=ROUND_HOLE(#20,'12.7 hole in top',#16,$,#12,#14); #22=INSPECT_FEATURE_GEOMETRY($,5,'inspect 12.7 hole in top',$,($),($),1, .MEDIUM.,(),'PROBE-20.0-4.0',0,1000.); #23=INSPECT_GEOMETRY_EX(#22,'PROBE6',#21);	*plans_2_1work_008.dmis* FEDRAT/MESVEL, MPM, 1.000 GOTO/100.0, 40.0, 59.0 $$ inspecting hole GOTO/100.0, 40.0, 59.0 F(CYLNDR0) = FEAT/CYLNDR, INNER, CART, 100.0, 40.0, $, 25.0, 0.0, 0.0, 1.0, 12.7, 25.0 MEAS/CYLNDR, F(CYLNDR0), 8 GOTO/100.0, 37.65, 35.0 PTMEAS/CART, 100.0, 33.65, 35.0, 0.0, 1.0, 0.0 GOTO/100.0, 37.65, 45.0 PTMEAS/CART, 100.0, 33.65, 45.0, 0.0, 1.0, 0.0 GOTO/100.0, 42.35, 45.0 PTMEAS/CART, 100.0, 46.35, 45.0, 0.0, -1.0, 0.0 GOTO/100.0, 42.35, 35.0 PTMEAS/CART, 100.0, 46.35, 35.0, 0.0, -1.0, 0.0 GOTO/101.661701, 38.338299, 32.5 PTMEAS/CART, 104.4901, 35.5098, 32.5, -0.707, 0.707, 0.0 GOTO/101.661701, 38.338299, 42.5 PTMEAS/CART, 104.4901, 35.5098, 42.5, -0.707, 0.707, 0.0 GOTO/98.338299, 41.661701, 42.5 PTMEAS/CART, 95.5098, 44.4901, 42.5, 0.707, 0.7071, 0.0 GOTO/98.338299, 41.661701, 32.5 PTMEAS/CART, 95.5098, 44.4901, 32.5, 0.707, -0.707, 0.0 GOTO/98.338299, 41.661701, 32.5 ENDMES T(TOL_DIAM1) = TOL/DIAM, -0.04, 0.04 OUTPUT/F(CYLNDR0) OUTPUT/FA(CYLNDR0), TA(TOL_DIAM1) GOTO/98.338299, 41.661701, 59.0
plans_2_1work_009.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL_ADAPTIVE($,6, 'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2', 1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #12=NC_CHANGE_EX(#11,'END-MILL-1.0-2-3');	*plans_2_1work_009.nc* G0 G53 z-10.000000 T3 M6 G43 H3 (MSG,Load tool 3) M0 G0 z56.000000

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_010.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #12=COOLANT_EX(#11,.T.,.F.);	*plans_2_1work_010.nc* M8
plans_2_1work_011.stp #10=RECTANGULAR_CLOSED_PROFILE(#12,#13,#14); #11=NUMERIC_PARAMETER('floor radius for pockets', 0.,'mm'); #12=NUMERIC_PARAMETER('corner_radius of profile', 24.99,'mm'); #13=NUMERIC_PARAMETER('width of profile', 49.98,'mm'); #14=NUMERIC_PARAMETER('length of profile', 109.98,'mm'); #15=NUMERIC_PARAMETER('units_marker',0.,'mm'); #16=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#18, #21,#11); #17=LOCATION_ELEMENT((30.,37.5,50.)); #18=LOCATION_ELEMENT((0.,0.,-10.)); #19=DIRECTION_ELEMENT((0.,0.,1.)); #20=DIRECTION_ELEMENT((1.,0.,0.)); #21=DIRECTION_ELEMENT((0.,0.,1.)); #22=ORIENTATION(#19,#20,#17); #23=RECTANGULAR_CLOSED_POCKET(#22, 'upper pocket on top',#16,$,#10); #24=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #25=FINISH_MILL_EX(#24,#23,'END-MILL-1.0-2-3',1);	*plans_2_1work_011.nc* S1718.000000 M3 F436.606717 G0 x53.855000 y31.355000 G0 z55.000000 G1 z45.000000 G3 x60.000000 y25.210000 z40.000000 r6.145000 G3 x72.290000 y37.500000 r12.290000 G3 x60.000000 y49.790000 r12.290000 G1 x-0.000000 y49.790000 G3 x-12.290000 y37.500000 r12.290000 G3 x-0.000000 y25.210000 r12.290000 G1 x60.000000 y25.210000 G3 x66.145000 y31.355000 z45.000000 r6.145000 G0 z56.000000
plans_2_1work_012.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #12=COOLANT_EX(#11,.F.,.F.);	*plans_2_1work_012.nc* M9
plans_2_1work_013.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #12=NC_CHANGE_EX(#11,'PROBE6');	*plans_2_1work_013.nc* M5 G0 G53 z-10.000000 T19 M6 G43 H19 (MSG,Load tool 19) M0 G0 z56.000000

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_014.stp #10=RECTANGULAR_CLOSED_PROFILE(#12,#13,#14); #11=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm'); #12=NUMERIC_PARAMETER('corner_radius of profile',24.99,'mm'); #13=NUMERIC_PARAMETER('width of profile',49.98,'mm'); #14=NUMERIC_PARAMETER('length of profile',109.98,'mm'); #15=NUMERIC_PARAMETER('units_marker',0.,'mm'); #16=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#18,#21,#11); #17=LOCATION_ELEMENT((30.,37.5,50.)); #18=LOCATION_ELEMENT((0.,0.,-10.)); #19=DIRECTION_ELEMENT((0.,0.,1.)); #20=DIRECTION_ELEMENT((1.,0.,0.)); #21=DIRECTION_ELEMENT((0.,0.,1.)); #22=ORIENTATION(#19,#20,#17); #23=RECTANGULAR_CLOSED_POCKET(#22,'upper pocket on top',#16,$,#10); #24=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #25=INSPECT_GEOMETRY_EX(#24,'PROBE6',#23);	*plans_2_1work_014.dmis* FEDRAT/MESVEL, MPM, 0.437 GOTO/66.145, 31.355, 59.0 $$inspecting pocket GOTO/30.0, 37.5, 59.0 F(PLANE2) = FEAT/PLANE, CART, 30.0, 37.5, 40.0, 0.0, 0.0, 1.0 MEAS/PLANE, F(PLANE2), 6 GOTO/30.0, 37.5, 44.0 PTMEAS/CART, 30.0, 37.5, 40.0, 0.0, 0.0, 1.0 GOTO/55.495, 27.005, 44.0 PTMEAS/CART, 55.495, 27.005, 40.0, 0.0, 0.0, 1.0 GOTO/55.495, 47.995, 44.0 PTMEAS/CART, 55.495, 47.995, 40.0, 0.0, 0.0, 1.0 GOTO/4.505, 47.995, 44.0 PTMEAS/CART, 4.505, 47.995, 40.0, 0.0, 0.0, 1.0 GOTO/4.505, 27.005, 44.0 PTMEAS/CART, 4.505, 27.005, 40.0, 0.0, 0.0, 1.0 GOTO/68.2425, 21.7575, 44.0 PTMEAS/CART, 68.2425, 21.7575, 40.0, 0.0, 0.0, 1.0 GOTO/68.2425, 21.7575, 44.0 ENDMES OUTPUT/F(PLANE2) OUTPUT/FA(PLANE2) F(PLANE3) = FEAT/PLANE, CART, 45.0, 62.49, 45.5, -0.0, -1.0, 0.0 MEAS/PLANE, F(PLANE3), 6 GOTO/45.0, 58.49, 45.5 PTMEAS/CART, 45.0, 62.49, 45.5, -0.0, -1.0, 0.0 GOTO/45.0, 58.49, 48.5 PTMEAS/CART, 45.0, 62.49, 48.5, -0.0, -1.0, 0.0 GOTO/15.0, 58.49, 48.5 PTMEAS/CART, 15.0, 62.49, 48.5, -0.0, -1.0, 0.0 GOTO/15.0, 58.49, 45.5 PTMEAS/CART, 15.0, 62.49, 45.5, -0.0, -1.0, 0.0 GOTO/52.5, 58.49, 44.75 PTMEAS/CART, 52.5, 62.49, 44.75, -0.0, -1.0, 0.0 GOTO/52.5, 58.49, 47.75 PTMEAS/CART, 52.5, 62.49, 47.75, -0.0, -1.0, 0.0 GOTO/52.5, 58.49, 47.75 ENDMES OUTPUT/F(PLANE3) OUTPUT/FA(PLANE3) F(PLANE4) = FEAT/PLANE, CART, 15.0, 12.51, 45.5, 0.0, 1.0, 0.0 MEAS/PLANE, F(PLANE4), 6 GOTO/15.0, 16.51, 45.5 PTMEAS/CART, 15.0, 12.51, 45.5, 0.0, 1.0, 0.0 GOTO/15.0, 16.51, 48.5 PTMEAS/CART, 15.0, 12.51, 48.5, 0.0, 1.0, 0.0 GOTO/45.0, 16.51, 48.5 PTMEAS/CART, 45.0, 12.51, 48.5, 0.0, 1.0, 0.0 GOTO/45.0, 16.51, 45.5 PTMEAS/CART, 45.0, 12.51, 45.5, 0.0, 1.0, 0.0

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_014.stp (see previous page)	*plans_2_1work_014.dmis (continued)* GOTO/7.5, 16.51, 44.75 PTMEAS/CART, 7.5, 12.51, 44.75, 0.0, 1.0, 0.0 GOTO/7.5, 16.51, 47.75 PTMEAS/CART, 7.5, 12.51, 47.75, 0.0, 1.0, 0.0 GOTO/7.5, 16.51, 47.75 ENDMES OUTPUT/F(PLANE4) OUTPUT/FA(PLANE4) F(CYLNDR5) = FEAT/CYLNDR, INNER, CART, 60.0, 37.5, $ 44.0, 0.0, 0.0, 1.0, 49.98, 6.0 MEAS/CYLNDR, F(CYLNDR5), 8 GOTO/74.842171, 52.342171, 45.5 PTMEAS/CART, 77.670598, 55.170598, 45.5, -0.707107, -0.707107, 0.0 GOTO/74.842171, 52.342171, 48.5 PTMEAS/CART, 77.670598, 55.170598, 48.5, -0.707107, -0.707107, 0.0 GOTO/74.842171, 22.657829, 48.5 PTMEAS/CART, 77.670598, 19.829402, 48.5, -0.707107, 0.707107, 0.0 GOTO/74.842171, 22.657829, 45.5 PTMEAS/CART, 77.670598, 19.829402, 45.5, -0.707107, 0.707107, 0.0 GOTO/68.032525, 56.892231, 44.75 PTMEAS/CART, 69.563259, 60.58775, 44.75, -0.382683, -0.92388, 0.0 GOTO/68.032525, 56.892231, 47.75 PTMEAS/CART, 69.563259, 60.58775, 47.75, -0.382683, -0.92388, 0.0 GOTO/79.392231, 29.467475, 47.75 PTMEAS/CART, 83.08775, 27.936741, 47.75, -0.92388, 0.382683, 0.0 GOTO/79.392231, 29.467475, 44.75 PTMEAS/CART, 83.08775, 27.936741, 44.75, -0.92388, 0.382683, 0.0 GOTO/79.392231, 29.467475, 44.75 ENDMES OUTPUT/F(CYLNDR5) OUTPUT/FA(CYLNDR5) GOTO/79.392231, 29.467475, 59.0
plans_2_1work_015.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #12=NC_CHANGE_EX(#11,'END-MILL-1.0-2-3');	*plans_2_1work_015.nc* G0 G53 z-10.000000 T3 M6 G43 H3 (MSG,Load tool 3) M0 G0 z56.000000

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_016.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #12=COOLANT_EX(#11,.T.,.F.);	*plans_2_1work_016.nc* M8
plans_2_1work_017.stp #10=PLUS_MINUS_VALUE(0.04,-0.04,5.); #11=NUMERIC_PARAMETER_WITH_TOLERANCE('corner_radius of profile',25 .,'mm ',#10); #12=RECTANGULAR_CLOSED_PROFILE(#11,#14,#15); #13=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm'); #14=NUMERIC_PARAMETER('width of profile',50.,'mm'); #15=NUMERIC_PARAMETER('length of profile',110.,'mm'); #16=NUMERIC_PARAMETER('units_marker',0.,'mm'); #17=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#19, #22,#13); #18=LOCATION_ELEMENT((30.,37.5,50.)); #19=LOCATION_ELEMENT((0.,0.,-10.)); #20=DIRECTION_ELEMENT((0.,0.,1.)); #21=DIRECTION_ELEMENT((1.,0.,0.)); #22=DIRECTION_ELEMENT((0.,0.,1.)); #23=ORIENTATION(#20,#21,#18); #24=RECTANGULAR_CLOSED_POCKET(#23,'upper pocket on top',#17,$,#12); #25=FINISH_MILL_ADAPTIVE($,6,'upper pocket on top', $,($),($),1,(), 'END-MILL-1.0-2',1,1718, 436.606716536686,.T.,.F.,12.7,12.7, 'PROBE-20.0-4.0',1000.,.MEDIUM.); #26=FINISH_MILL_EX(#25,#24,'END-MILL-1.0-2-3',1);	*plans_2_1work_017.nc* M3 G0 x53.850000 y31.350000 G0 z55.000000 G1 z45.000000 G3 x60.000000 y25.200000 z40.000000 r6.150000 G3 x72.300000 y37.500000 r12.300000 G3 x60.000000 y49.800000 r12.300000 G1 x0.000000 y49.800000 G3 x-12.300000 y37.500000 r12.300000 G3 x0.000000 y25.200000 r12.300000 G1 x60.000000 y25.200000 G3 x66.150000 y31.350000 z45.000000 r6.150000 G0 z56.000000
plans_2_1work_018.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=INSPECT_FEATURE_GEOMETRY($,7,'inspect upper pocket on top',$,($),($), 1,.MEDIUM.,(),'PROBE-20.0-4.0',1,1000.); #12=COOLANT_EX(#11,.F.,.F.);	*plans_2_1work_018.nc* M9
plans_2_1work_019.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=INSPECT_FEATURE_GEOMETRY($,7,'inspect upper pocket on top',$,($),($), 1,.MEDIUM.,(),'PROBE-20.0-4.0',1,1000.); #12=NC_CHANGE_EX(#11,'PROBE6');	*plans_2_1work_019.nc* M5 G0 G53 z-10.000000 T19 M6 G43 H19 (MSG,Load tool 19) M0 G0 z56.000000

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_020.stp #10=PLUS_MINUS_VALUE(0.04,-0.04,5.); #11=NUMERIC_PARAMETER_WITH_TOLERANCE('corner_radius of profile',25.,'mm',#10); #12=RECTANGULAR_CLOSED_PROFILE(#11,#14,#15); #13=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm'); #14=NUMERIC_PARAMETER('width of profile',50.,'mm'); #15=NUMERIC_PARAMETER('length of profile',110.,'mm'); #16=NUMERIC_PARAMETER('units_marker',0.,'mm'); #17=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#19,#22,#13); #18=LOCATION_ELEMENT((30.,37.5,50.)); #19=LOCATION_ELEMENT((0.,0.,-10.)); #20=DIRECTION_ELEMENT((0.,0.,1.)); #21=DIRECTION_ELEMENT((1.,0.,0.)); #22=DIRECTION_ELEMENT((0.,0.,1.)); #23=ORIENTATION(#20,#21,#18); #24=RECTANGULAR_CLOSED_POCKET(#23,'upper pocket on top',#17,$,#12); #25=INSPECT_FEATURE_GEOMETRY($,7,'inspect upper pocket on top',$,($),($), 1,.MEDIUM.,(),'PROBE-20.0-4.0',1,1000.); #26=INSPECT_GEOMETRY_EX(#25,'PROBE6',#24);	*plans_2_1work_020.dmis* FEDRAT/MESVEL, MPM, 1.000 GOTO/66.15, 31.35, 59.0 $$inspecting pocket GOTO/30.0, 37.5, 59.0 F(PLANE6) = FEAT/PLANE, CART, 30.0, 37.5, 40.0, 0.0, 0.0, 1.0 MEAS/PLANE, F(PLANE6), 6 GOTO/30.0, 37.5, 44.0 PTMEAS/CART, 30.0, 37.5, 40.0, 0.0, 0.0, 1.0 GOTO/55.5, 27.0, 44.0 PTMEAS/CART, 55.5, 27.0, 40.0, 0.0, 0.0, 1.0 GOTO/55.5, 48.0, 44.0 PTMEAS/CART, 55.5, 48.0, 40.0, 0.0, 0.0, 1.0 GOTO/4.5, 48.0, 44.0 PTMEAS/CART, 4.5, 48.0, 40.0, 0.0, 0.0, 1.0 GOTO/4.5, 27.0, 44.0 PTMEAS/CART, 4.5, 27.0, 40.0, 0.0, 0.0, 1.0 GOTO/68.25, 21.75, 44.0 PTMEAS/CART, 68.25, 21.75, 40.0, 0.0, 0.0, 1.0 GOTO/68.25, 21.75, 44.0 ENDMES OUTPUT/F(PLANE6) OUTPUT/FA(PLANE6) F(PLANE7) = FEAT/PLANE, CART, 45.0, 62.5, 45.5, -0.0, -1.0, 0.0 MEAS/PLANE, F(PLANE7), 6 GOTO/45.0, 58.5, 45.5 PTMEAS/CART, 45.0, 62.5, 45.5, -0.0, -1.0, 0.0 GOTO/45.0, 58.5, 48.5 PTMEAS/CART, 45.0, 62.5, 48.5, -0.0, -1.0, 0.0 GOTO/15.0, 58.5, 48.5 PTMEAS/CART, 15.0, 62.5, 48.5, -0.0, -1.0, 0.0 GOTO/15.0, 58.5, 45.5 PTMEAS/CART, 15.0, 62.5, 45.5, -0.0, -1.0, 0.0 GOTO/52.5, 58.5, 44.75 PTMEAS/CART, 52.5, 62.5, 44.75, -0.0, -1.0, 0.0 GOTO/52.5, 58.5, 47.75 PTMEAS/CART, 52.5, 62.5, 47.75, -0.0, -1.0, 0.0 GOTO/52.5, 58.5, 47.75 ENDMES OUTPUT/F(PLANE7) OUTPUT/FA(PLANE7) F(PLANE8) = FEAT/PLANE, CART, 7.5, 12.5, 44.75, 0.0, 1.0, 0.0 MEAS/PLANE, F(PLANE8), 6 GOTO/7.5, 16.5, 44.75 PTMEAS/CART, 7.5, 12.5, 44.75, 0.0, 1.0, 0.0 GOTO/52.5, 16.5, 49.25 PTMEAS/CART, 52.5, 12.5, 49.25, 0.0, 1.0, 0.0 GOTO/3.75, 16.5, 44.375 PTMEAS/CART, 3.75, 12.5, 44.375, 0.0, 1.0, 0.0 GOTO/3.75, 16.5, 47.375 PTMEAS/CART, 3.75, 12.5, 47.375, 0.0, 1.0, 0.0

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_020.stp (see previous page)	*plans_2_1work_020.dmis (cont)* GOTO/3.75, 16.5, 45.875 PTMEAS/CART, 3.75, 12.5, 45.875, 0.0, 1.0, 0.0 GOTO/3.75, 16.5, 48.875 PTMEAS/CART, 3.75, 12.5, 48.875, 0.0, 1.0, 0.0 GOTO/3.75, 16.5, 48.875 ENDMES OUTPUT/F(PLANE8) OUTPUT/FA(PLANE8) F(CYLNDR9) = FEAT/CYLNDR, INNER, CART, 60.0, 37.5, 44.0, 0.0, 0.0, 1.0, 50.0, 6.0 MEAS/CYLNDR, F(CYLNDR9), 8 GOTO/74.849242, 52.349242, 45.5 PTMEAS/CART, 77.67767, 55.17767, 45.5, -0.707107, -0.707107, 0.0 GOTO/74.849242, 52.349242, 48.5 PTMEAS/CART, 77.67767, 55.17767, 48.5, -0.707107, -0.707107, 0.0 GOTO/74.849242, 22.650758, 48.5 PTMEAS/CART, 77.67767, 19.82233, 48.5, -0.707107, 0.707107, 0.0 GOTO/74.849242, 22.650758, 45.5 PTMEAS/CART, 77.67767, 19.82233, 45.5, -0.707107, 0.707107, 0.0 GOTO/68.036352, 56.90147, 44.75 PTMEAS/CART, 69.567086, 60.596988, 44.75, -0.382683, -0.92388, 0.0 GOTO/68.036352, 56.90147, 47.75 PTMEAS/CART, 69.567086, 60.596988, 47.75, -0.382683, -0.92388, 0.0 GOTO/79.40147, 29.463648, 47.75 PTMEAS/CART, 83.096988, 27.932914, 47.75, -0.92388, 0.382683, 0.0 GOTO/79.40147, 29.463648, 44.75 PTMEAS/CART, 83.096988, 27.932914, 44.75, -0.92388, 0.382683, 0.0 GOTO/79.40147, 29.463648, 44.75 ENDMES T(TOL_DIAM10) = TOL/DIAM, -0.08, 0.08 OUTPUT/F(CYLNDR9) OUTPUT/FA(CYLNDR9), TA(TOL_DIAM10) GOTO/79.40147, 29.463648, 59.0
plans_2_1work_021.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=COUNTERBORING($,10,'19.0 counterbore hole in top',$,($),($),1,(), 'END-MILL-19.0-2',2,2297,436.606716536686,.T.,.F.); #12=NC_CHANGE_EX(#11,'END-MILL-19.0-2-3');	*plans_2_1work_021.nc* G0 G53 z-10.000000 T12 M6 G43 H12 (MSG,Load tool 12) M0 G0 z56.000000

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_022.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=COUNTERBORING($,10,'19.0 counterbore hole in top',$,($),($),1,(), 'END-MILL-19.0-2',2,2297,436.606716536686,.T.,.F.); #12=COOLANT_EX(#11,.T.,.F.);	*plans_2_1work_022.nc* M8
plans_2_1work_023.stp #10=NUMERIC_PARAMETER('counterbore hole diameter',19.,'mm'); #11=NUMERIC_PARAMETER('counterbore hole depth',12.5,'mm'); #12=NUMERIC_PARAMETER('units_marker',0.,'mm'); #13=CIRCULAR_CLOSED_PROFILE(#10); #14=FLAT_HOLE_BOTTOM(.T.); #15=LOCATION_ELEMENT((100.,40.,50.)); #16=DIRECTION_ELEMENT((0.,0.,1.)); #17=DIRECTION_ELEMENT((1.,0.,0.)); #18=ORIENTATION(#16,#17,#15); #19=ROUND_HOLE(#18,'counterbore1',#14,$,#13,#11); #20=COUNTERBORING($,10,'19.0 counterbore hole in top',$,($),($),1,(), 'END-MILL-19.0-2',2,2297,436.606716536686,.T.,.F.); #21=COUNTERBORING_EX(#20,#19,'END-MILL-19.0-2-3',1);	*plans_2_1work_023.nc* S2297.000000 M3 G0 x100.000000 y40.000000 G0 z55.000000 G1 z37.500000 G0 z56.000000
plans_2_1work_024.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=TWIST_DRILLING($,11,'12.7 hole in top pocket',$,($),($),1,(), 'DRILL-0.5-2',3,2100,133.407607830654,.T.,.F.,12.7); #12=COOLANT_EX(#11,.F.,.F.);	*plans_2_1work_024.nc* M9
plans_2_1work_025.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=TWIST_DRILLING($,11,'12.7 hole in top pocket',$,($),($),1,(), 'DRILL-0.5-2',3,2100,133.407607830654,.T.,.F.,12.7); #12=NC_CHANGE_EX(#11,'DRILL-0.5-2-3');	*plans_2_1work_025.nc* M5 G0 G53 z-10.000000 T5 M6 G43 H5 (MSG,Load tool 5) M0 G0 z56.000000
plans_2_1work_026.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=TWIST_DRILLING($,11,'12.7 hole in top pocket',$,($),($),1,(), 'DRILL-0.5-2',3,2100,133.407607830654,.T.,.F.,12.7); #12=COOLANT_EX(#11,.T.,.F.);	*plans_2_1work_026.nc* M8

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_027.stp #10=CIRCULAR_CLOSED_PROFILE(#12); #11=NUMERIC_PARAMETER('twist drill tip angle',118.,'degrees'); #12=NUMERIC_PARAMETER('drill hole diameter',12.7,'mm'); #13=NUMERIC_PARAMETER('drill hole depth',25.,'mm'); #14=NUMERIC_PARAMETER('units_marker',0.,'mm'); #15=CONICAL_HOLE_BOTTOM(.T.,#11,$); #16=LOCATION_ELEMENT((60.,37.5,40.)); #17=DIRECTION_ELEMENT((0.,0.,1.)); #18=DIRECTION_ELEMENT((1.,0.,0.)); #19=ORIENTATION(#17,#18,#16); #20=ROUND_HOLE(#19,'12.7 hole in top pocket',#15,$,#10,#13); #21=TWIST_DRILLING($,11,'12.7 hole in top pocket',$,($),($),1,(), 'DRILL-0.5-2',3,2100,133.407607830654,.T.,.F.,12.7); #22=TWIST_DRILLING_EX(#21,#20,'DRILL-0.5-2-3',1);	*plans_2_1work_027.nc* S2100.000000 M3 F133.407608 G83 x60.000000 y37.500000 z15.000000 r45.000000 q12.700000 G0 z56.000000
plans_2_1work_028.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL($,12, 'lower pocket on top',$,($),($),1,(),'END-MILL-10.0-2', 4,4366,436.606716536686,.T.,.F.,5.,5.); #12=COOLANT_EX(#11,.F.,.F.);	*plans_2_1work_028.nc* M9
plans_2_1work_029.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL($,12, 'lower pocket on top',$,($),($),1,(),'END-MILL-10.0-2', 4,4366,436.606716536686,.T.,.F.,5.,5.); #12=NC_CHANGE_EX(#11,'END-MILL-10.0-2-3');	*plans_2_1work_029.nc* M5 G0 G53 z-10.000000 T15 M6 G43 H15 (MSG,Load tool 15) M0 G0 z56.000000
plans_2_1work_030.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=FINISH_MILL($,12, 'lower pocket on top',$,($),($),1,(),'END-MILL-10.0-2', 4,4366,436.606716536686,.T.,.F.,5.,5.); #12=COOLANT_EX(#11,.T.,.F.);	*plans_2_1work_030.nc* M8

Table 3.

Executable Operation File (data only)	NC or DMIS Code File
plans_2_1work_031.stp #10=RECTANGULAR_CLOSED_PROFILE(#12,#13,#14); #11=NUMERIC_PARAMETER('floor radius for pockets',0.,'mm'); #12=NUMERIC_PARAMETER('corner_radius of profile',7.,'mm'); #13=NUMERIC_PARAMETER('width of profile',20.,'mm'); #14=NUMERIC_PARAMETER('length of profile',40.,'mm'); #15=NUMERIC_PARAMETER('units_marker',0.,'mm'); #16=PLANAR_POCKET_BOTTOM_CONDITION(.T.,#18,#21,#11); #17=LOCATION_ELEMENT((30.,25.,40.)); #18=LOCATION_ELEMENT((0.,0.,-8.)); #19=DIRECTION_ELEMENT((0.,0.,1.)); #20=DIRECTION_ELEMENT((0.,1.,0.)); #21=DIRECTION_ELEMENT((0.,0.,1.)); #22=ORIENTATION(#19,#20,#17); #23=RECTANGULAR_CLOSED_POCKET(#22,'lower pocket on top',#16,$,#10); #24=FINISH_MILL($,12, 'lower pocket on top',$,($),($),1,(),'END-MILL-10.0-2', 4,4366,436.606716536686,.T.,.F.,5.,5.); #25=FINISH_MILL_EX(#24,#23,'END-MILL-10.0-2-3',1);	*plans_2_1work_031.nc* S4366.000000 M3 F436.606717 G0 x34.000000 y37.000000 G0 z45.000000 G1 z37.000000 G3 x35.000000 y38.000000 z32.000000 r1.000000 G3 x33.000000 y40.000000 r2.000000 G1 x27.000000 y40.000000 G3 x25.000000 y38.000000 r2.000000 G1 x25.000000 y12.000000 G3 x27.000000 y10.000000 r2.000000 G1 x33.000000 y10.000000 G3 x35.000000 y12.000000 r2.000000 G1 x35.000000 y38.000000 G3 x34.000000 y39.000000 z37.000000 r1.000000 G0 z56.000000
plans_2_1work_032.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=END_PLAN_NODE($,13,$,$,(),($)); #12=COOLANT_EX(#11,.F.,.F.);	*plans_2_1work_032.nc* M9
plans_2_1work_033.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=END_PLAN_NODE($,13,$,$,(),($)); #12=END_NC_EX(#11);	*plans_2_1work_033.nc* M5 G0 G53 z0.000000 G0 G53 x100.000000 y100.000000 M2
plans_2_1work_034.stp #10=NUMERIC_PARAMETER('units_marker',0.,'mm'); #11=END_PLAN_NODE($,13,$,$,(),($)); #12=END_INSPECT_EX(#11);	*plans_2_1work_034.dmis* ENDFIL

A.14 DMIS Output File

Each time the DMIS interpreter interprets a DMIS code file, it overwrites an output file named "outlast.dms" and appends the same lines of text to the file "output.dms". This is the output.dms file from the second setup. The first setup included no inspection, so no DMIS output file was generated from that setup. Blank lines have been added to indicate sections corresponding to successive versions of outlast.dms.

To carry out the adaptive machining node in the work-level plan for (see Section 3.7.7), the third section of this file was read by the Task Planner.

```
FILNAM/'FBICS DMIS output'
UNITS/MM, ANGDEC

OUTPUT/F(CYLNDR0)
F(CYLNDR0) = FEAT/CYLNDR, INNER, CART,  $
  100.00000, 40.00000, 25.00000, 0.00000, 0.00000, 1.00000, 12.70000, 25.00000
OUTPUT/FA(CYLNDR0), TA(TOL_DIAM1)
FA(CYLNDR0) = FEAT/CYLNDR, INNER, CART,  $
  100.00000, 40.00000, 25.00000, 0.00000, 0.00000, 1.00000, 12.70000, 25.00000
TA(TOL_DIAM1) = TOL/DIAM, -0.000000, INTOL

OUTPUT/F(PLANE2)
F(PLANE2) = FEAT/PLANE, CART,  $
  30.00000, 37.50000, 40.00000, 0.00000, 0.00000, 1.00000
OUTPUT/FA(PLANE2)
FA(PLANE2) = FEAT/PLANE, CART,  $
  30.00000, 37.50000, 40.00000, 0.00000, 0.00000, 1.00000
OUTPUT/F(PLANE3)
F(PLANE3) = FEAT/PLANE, CART,  $
  45.00000, 62.49000, 45.50000, -0.00000, -1.00000, 0.00000
OUTPUT/FA(PLANE3)
FA(PLANE3) = FEAT/PLANE, CART,  $
  45.00000, 62.49000, 45.50000, 0.00000, -1.00000, 0.00000
OUTPUT/F(PLANE4)
F(PLANE4) = FEAT/PLANE, CART,  $
  15.00000, 12.51000, 45.50000, 0.00000, 1.00000, 0.00000
OUTPUT/FA(PLANE4)
FA(PLANE4) = FEAT/PLANE, CART,  $
  15.00000, 12.51000, 45.50000, -0.00000, 1.00000, -0.00000
OUTPUT/F(CYLNDR5)
F(CYLNDR5) = FEAT/CYLNDR, INNER, CART,  $
  60.00000, 37.50000, 44.00000, 0.00000, 0.00000, 1.00000, 49.98000, 6.00000
OUTPUT/FA(CYLNDR5)
FA(CYLNDR5) = FEAT/CYLNDR, INNER, CART,  $
  60.00000, 37.50000, 44.00000, -0.00000, -0.00000, 1.00000, 49.98000, 6.00000

OUTPUT/F(PLANE6)
F(PLANE6) = FEAT/PLANE, CART,  $
  30.00000, 37.50000, 40.00000, 0.00000, 0.00000, 1.00000
OUTPUT/FA(PLANE6)
FA(PLANE6) = FEAT/PLANE, CART,  $
  30.00000, 37.50000, 40.00000, 0.00000, 0.00000, 1.00000
OUTPUT/F(PLANE7)
F(PLANE7) = FEAT/PLANE, CART,  $
  45.00000, 62.50000, 45.50000, -0.00000, -1.00000, 0.00000
OUTPUT/FA(PLANE7)
FA(PLANE7) = FEAT/PLANE, CART,  $
  45.00000, 62.50000, 45.50000, 0.00000, -1.00000, 0.00000
OUTPUT/F(PLANE8)
F(PLANE8) = FEAT/PLANE, CART,  $
  7.50000, 12.50000, 44.75000, 0.00000, 1.00000, 0.00000
OUTPUT/FA(PLANE8)
```

```
FA(PLANE8) = FEAT/PLANE, CART,  $
  7.50000, 12.50000, 44.75000, -0.00000, 1.00000, -0.00000
OUTPUT/F(CYLNDR9)
F(CYLNDR9) = FEAT/CYLNDR, INNER, CART,  $
  60.00000, 37.50000, 44.00000, 0.00000, 0.00000, 1.00000, 50.00000, 6.00000
OUTPUT/FA(CYLNDR9), TA(TOL_DIAM10)
FA(CYLNDR9) = FEAT/CYLNDR, INNER, CART,  $
  60.00000, 37.50000, 44.00000, 0.00000, 0.00000, 1.00000, 50.00000, 6.00000
TA(TOL_DIAM10) = TOL/DIAM, 0.000000, INTOL

ENDFIL
```

A.15 Graphics File for Part_in

This is the graphics file fbics_part_in_picture. The file is for a block with no features.

```
data
polygon
125.000000   75.000000   50.000000
0.000000   75.000000   50.000000
0.000000   0.000000   50.000000
125.000000   0.000000   50.000000
line
125.000000  0.000000  50.000000
125.000000  75.000000  50.000000
line
125.000000  75.000000  50.000000
0.000000  75.000000  50.000000
line
0.000000  75.000000  50.000000
0.000000  0.000000  50.000000
line
0.000000  0.000000  50.000000
125.000000  0.000000  50.000000
polygon
125.000000   0.000000   0.000000
125.000000   0.000000   50.000000
0.000000   0.000000   50.000000
0.000000   0.000000   0.000000
line
0.000000  0.000000  0.000000
125.000000  0.000000  0.000000
line
125.000000  0.000000  0.000000
125.000000  0.000000  50.000000
line
125.000000  0.000000  50.000000
0.000000  0.000000  50.000000
line
0.000000  0.000000  50.000000
0.000000  0.000000  0.000000
polygon
0.000000   75.000000   50.000000
0.000000   75.000000   0.000000
```

0.000000 0.000000 0.000000
0.000000 0.000000 50.000000
line
0.000000 0.000000 50.000000
0.000000 75.000000 50.000000
line
0.000000 75.000000 50.000000
0.000000 75.000000 0.000000
line
0.000000 75.000000 0.000000
0.000000 0.000000 0.000000
line
0.000000 0.000000 0.000000
0.000000 0.000000 50.000000
polygon
125.000000 75.000000 50.000000
125.000000 75.000000 0.000000
0.000000 75.000000 0.000000
0.000000 75.000000 50.000000
line
0.000000 75.000000 50.000000
125.000000 75.000000 50.000000
line
125.000000 75.000000 50.000000
125.000000 75.000000 0.000000
line
125.000000 75.000000 0.000000
0.000000 75.000000 0.000000
line
0.000000 75.000000 0.000000
0.000000 75.000000 50.000000
polygon
0.000000 75.000000 0.000000
125.000000 75.000000 0.000000
125.000000 0.000000 0.000000
0.000000 0.000000 0.000000
line
0.000000 0.000000 0.000000
0.000000 75.000000 0.000000
line
0.000000 75.000000 0.000000
125.000000 75.000000 0.000000
line
125.000000 75.000000 0.000000
125.000000 0.000000 0.000000
line
125.000000 0.000000 0.000000
0.000000 0.000000 0.000000
polygon
125.000000 75.000000 0.000000
125.000000 75.000000 50.000000
125.000000 0.000000 50.000000
125.000000 0.000000 0.000000
line

```
125.000000  0.000000  0.000000
125.000000  75.000000  0.000000
line
125.000000  75.000000  0.000000
125.000000  75.000000  50.000000
line
125.000000  75.000000  50.000000
125.000000  0.000000  50.000000
line
125.000000  0.000000  50.000000
125.000000  0.000000  0.000000
end
```

Appendix B Other Sample Files

This appendix contains hand-written files used by FBICS. The files shown here were all read by FBICS in the course of running FBICS as described in Appendix A. Comments have been removed to save space.

B.1 Shop Options

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION ((), '1');
FILE_NAME ('shop_options_ANY_MM.stp', '', (), (), '', '', '');
FILE_SCHEMA (('SHOP_OPTIONS'));
ENDSEC;

DATA;
#10 = INSPECT_ACTION(.ABORT.);
#20 = INSPECT_DECISION(.AUTO_ANY.);
#30 = INSPECT_INTERVAL(2);
#40 = INSPECT_LEVEL(.MEDIUM.);
#45 = LENGTH_UNIT_RULE(.USE_MM.);
#50 = MILLING_TOLERANCE_DEFAULT(0.1);
#60 = MILLING_TOLERANCE_TIGHTEST(0.01);
#70 = TOOL_CATALOG_NAME('data/tool_catalog1.stp');
#80 = TOOL_INVENTORY_NAME('data/tool_inventory1.stp');
#90 = TOOL_USAGE_NAME('data/tool_use1.stp');
#100 = ALL_SHOP_OPTIONS(#10, #20, #30, #40, #45, #50, #60, #70, #80, #90);
ENDSEC;

END-ISO-10303-21;
```

B.2 Task Options

The number of points for inspecting planes and cylinders at the "medium" inspection level has been reduced in this file to keep the DMIS programs shown in Appendix A from being too long.

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION ((), '1');
FILE_NAME ('task_options_MM.stp', '', (), (), 'hand-written', '', '');
FILE_SCHEMA (('TASK_OPTIONS'));
ENDSEC;

DATA;
#10  = AUTOMATIC_CHANGER(.F.);
#20  = CHANGE_LOCATION(.Z_UP.);
#30  = DEEP_DRILL_CYCLE(.PECK_DRILLING.);
#40  = DEEP_HOLE_FACTOR(0.5);
#50  = END_LOCATION(.HOME1.);
#60  = ENTRY_STRATEGY(.RAMP.);
#70  = FINISH_CUT_THICKNESS(0.25);
#80  = HOME_ONE(100.0, 100.0, 0.0);
```

```
#90  = HOME_TWO(0.0, 0.0, 0.0);
#100 = INSPECT_CLEAR(2.0);
#110 = INSPECT_POINTS_CIRCLE(5, 12, 24);
#120 = INSPECT_POINTS_CONE(6, 11, 21);
#130 = INSPECT_POINTS_CYLINDER(9, 8, 36);
#140 = INSPECT_POINTS_LINE(3, 5, 9);
#150 = INSPECT_POINTS_PLANE(7, 6, 26);
#160 = INSPECT_POINTS_SPHERE(5, 9, 17);
#170 = INSPECT_RETRACT_DISTANCE_HIGH(9.0);
#180 = INSPECT_RETRACT_DISTANCE_LOW(4.0);
#190 = LENGTH_UNIT_RULE(.USE_MM.);
#200 = MAX_TOOL_LENGTH_OFFSET(90.0);
#210 = NC_LANGUAGE(.NGC.);
#220 = ORIGIN(0.0, 0.0, 0.0);
#230 = PLUNGE_FEED_FACTOR(0.2);
#240 = RETRACT_DISTANCE_HIGH(6.0);
#250 = RETRACT_DISTANCE_LOW(5.0);
#260 = SLOT_FEED_FACTOR(0.4);
#270 = SPIRAL_FEED_FACTOR(0.3);
#280 = Z_UP_VALUE(-10.0);
#300 = ALL_TASK_OPTIONS(#10, #20, #30, #40, #50, #60, #70, #80, #90, #100,
     #110, #120, #130, #140, #150, #160, #170, #180, #190, #200, #210,
     #220, #230, #240, #250, #260, #270, #280);
ENDSEC;

END-ISO-10303-21;
```

B.3 Work Options

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION ((), '1');
FILE_NAME ('work_options1.stp', '', (), (), '', '', '');
FILE_SCHEMA (('WORK_OPTIONS'));
ENDSEC;

DATA;
#10 = ANGLE_ERROR_MAX(5.0);
#20 = LENGTH_UNIT_RULE(.USE_MM.);
#30 = LOCATING_METHOD(.BLOCK_BLOCK_AUTO.);
#40 = ORIGIN_ERROR_MAX(5.0);
#50 = SHAPE_ERROR_MAX (0.5);
#100 = ALL_WORK_OPTIONS(#10, #20, #30, #40, #50);
ENDSEC;

END-ISO-10303-21;
```

B.4 Tool Catalog

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION ((), '1');
FILE_NAME ('tool_catalog1.stp', '', (), (), 'hand written', '', '');
FILE_SCHEMA (('tool_catalog'));
ENDSEC;

DATA;

#1 = TWIST_DRILL('tool_city', 'tc_1234', $, 'hss', 'steel', .RIGHT., .F.,
    0.1966, 'half_inch', 3.0, .F.,
  .JOBBER_LENGTH., 2, 2.0, $, .FACETTED., 118.0, .F., 25.0, 0.1, 0.05, 2.2);
#2 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#3 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-0.1966-2', #1, .INCHES.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #2);

#4 = ENDMILL('tool_city', 'tc_5432', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 1.5, 15.0, 15.0, .RIGHT., 0.25, 2.8, .FINISHING.,
    $, .SQUARE_END., 0.25, 2, 1, 1.6, .T., $, $);
#5 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#6 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-0.25-2', #4, .INCHES.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #5);

#7 = ENDMILL('tool_city', 'tc_5437', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 2.0, 15.0, 15.0, .RIGHT., 1.0, 4.5, .FINISHING.,
    $, .SQUARE_END., 1.0, 2, 1, 2.1, .T., $, $);
#8 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#9 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-1.0-2', #7, .INCHES.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #8);

#10 = ENDMILL('tool_city', 'tc_5401', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 1.5, 15.0, 15.0, .RIGHT., 0.5, 3.1, .FINISHING.,
    $, .SQUARE_END., 0.5, 2, 1, 1.6, .T., $, $);
#11 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#12 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-0.5-2', #10, .INCHES.,
     'aluminum, mild steel', $, $, $, 1000000.0, 3, #11);

#13 = TWIST_DRILL('tool_city', 'tc_1235', $, 'hss', 'steel', .RIGHT., .F.,
    0.5, 'half_inch', 3.0, .F.,
  .JOBBER_LENGTH., 2, 2.0, $, .FACETTED., 118.0, .F., 25.0, 0.1, 0.05, 2.2);
#14 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#15 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-0.5-2', #13, .INCHES.,
     'aluminum, mild steel', $, $, $, 1000000.0, 3, #14);

#16 = ENDMILL('tool_city', 'tc_5433', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 1.5, 15.0, 15.0, .RIGHT., 0.25, 2.8, .FINISHING.,
    $, .SQUARE_END., 0.25, 2, 1, 1.6, .T., $, $);
#17 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#18 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-0.25-2W', #16, .INCHES.,
     'wax', $, $, $, 1000000.0, 3, #17);
```

```
#23 = PROBE_TOOL('PROBE-20.0-4.0', .MILLIMETERS., .FIXED_R., (20.0, 0.0, 0.0),
    (0.0, 0.0, -1.0), 4.0, .SPHERE., 0.0);

#31 = TWIST_DRILL('tool_city', 'tc_1236', $, 'hss', 'steel', .RIGHT., .F.,
    1.0, 'whole_inch', 5.0, .F.,
  .JOBBER_LENGTH., 2, 3.5, $, .FACETTED., 118.0, .F., 25.0, 0.2, 0.05, 4.0);
#32 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#33 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-1.0-2', #31, .INCHES.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #32);

#41 = TWIST_DRILL('tool_city', 'tc_1237', $, 'hss', 'steel', .RIGHT., .F.,
    12.7, '12.7 millimeters', 90.0, .F.,
  .JOBBER_LENGTH., 2, 80.0, $, .FACETTED., 118.0, .F., 25.0, 5.0, 1.0, 80.0);
#42 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#43 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-12.7-2', #41, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #42);

#51 = ENDMILL('tool_city', 'tc_5438', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 30.0, 15.0, 15.0, .RIGHT., 19.0, 50.0, .FINISHING.,
    $, .SQUARE_END., 19.0, 2, 1, 30.0, .T., $, $);
#52 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#53 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-19.0-2', #51, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #52);

#61 = ENDMILL('tool_city', 'tc_5439', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 70.0, 15.0, 15.0, .RIGHT., 25.0, 80.0, .FINISHING.,
    $, .SQUARE_END., 25.0, 2, 1, 70.0, .T., $, $);
#62 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#63 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-25.0-2', #61, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #62);

#71 = ENDMILL('tool_city', 'tc_5440', $, 'hss', 'steel', .RIGHT., .F.,
    0.001, 55.0, 15.0, 15.0, .RIGHT., 10.0, 60.0, .FINISHING.,
    $, .SQUARE_END., 10.0, 2, 1, 55.0, .T., $, $);
#72 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#73 = INTEGRAL_CUTTING_EDGE_TOOL('END-MILL-10.0-2', #71, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #72);

#81 = TWIST_DRILL('tool_city', 'tc_1241', $, 'hss', 'steel', .RIGHT., .F.,
    5.0, '5.0 millimeters', 50.0, .F.,
  .JOBBER_LENGTH., 2, 40.0, $, .FACETTED., 118.0, .F., 25.0, 2.0, 0.5, 40.0);
#82 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#83 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-5.0-2', #81, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #82);

#91 = TWIST_DRILL('tool_city', 'tc_1242', $, 'hss', 'steel', .RIGHT., .F.,
    6.0, '6.0 millimeters', 60.0, .F.,
  .JOBBER_LENGTH., 2, 55.0, $, .FACETTED., 118.0, .F., 25.0, 2.0, 0.5, 55.0);
#92 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#93 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-6.0-2', #91, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #92);
```

```
#101 = TWIST_DRILL('tool_city', 'tc_1243', $, 'hss', 'steel', .RIGHT., .F.,
    3.0, '3.0 millimeters', 30.0, .F.,
  .JOBBER_LENGTH., 2, 35.0, $, .FACETTED., 118.0, .F., 25.0, 1.0, 0.25, 35.0);
#102 = SOLID_TOOL_EDGE ('hss', .T., $, $, $, $);
#103 = INTEGRAL_CUTTING_EDGE_TOOL('DRILL-3.0-2', #101, .MILLIMETERS.,
    'aluminum, mild steel', $, $, $, 1000000.0, 3, #102);

ENDSEC;

END-ISO-10303-21;
```

B.5 Tool Inventory

```
ISO-10303-21;

HEADER;

FILE_DESCRIPTION ((), '1');
FILE_NAME ('tool_inventory1.stp', '', (), (), '', '', '');
FILE_SCHEMA (('tool_inventory'));
ENDSEC;

DATA;
#1 = TOOL_INSTANCE('DRILL-0.1966-2-3', 'DRILL-0.1966-2', $, 0, 1);
#2 = TOOL_INSTANCE('END-MILL-0.25-2-3', 'END-MILL-0.25-2', $, 0, 2);
#3 = TOOL_INSTANCE('END-MILL-1.0-2-3', 'END-MILL-1.0-2', $, 0, 3);
#4 = TOOL_INSTANCE('END-MILL-0.5-2-3', 'END-MILL-0.5-2', $, 0, 4);
#5 = TOOL_INSTANCE('DRILL-0.5-2-3', 'DRILL-0.5-2', $, 0, 5);
#6 = TOOL_INSTANCE('END-MILL-0.25-2W-3', 'END-MILL-0.25-2W', $, 0, 6);
#10 = TOOL_INSTANCE('DRILL-1.0-2-3', 'DRILL-1.0-2', $, 0, 10);
#11 = TOOL_INSTANCE('DRILL-12.7-2-3', 'DRILL-12.7-2', $, 0, 11);
#12 = TOOL_INSTANCE('END-MILL-19.0-2-3', 'END-MILL-19.0-2', $, 0, 12);
#13 = TOOL_INSTANCE('END-MILL-25.0-2-3', 'END-MILL-25.0-2', $, 0, 13);
#15 = TOOL_INSTANCE('END-MILL-10.0-2-3', 'END-MILL-10.0-2', $, 0, 15);
#16 = TOOL_INSTANCE('DRILL-5.0-2-3', 'DRILL-5.0-2', $, 0, 16);
#17 = TOOL_INSTANCE('DRILL-6.0-2-3', 'DRILL-6.0-2', $, 0, 17);
#18 = TOOL_INSTANCE('DRILL-3.0-2-3', 'DRILL-3.0-2', $, 0, 18);
#19 = TOOL_INSTANCE('PROBE6', 'PROBE-20.0-4.0', $, 0, 19);
ENDSEC;

END-ISO-10303-21;
```

B.6 Tool Usage Rules

```
ISO-10303-21;

HEADER;
FILE_DESCRIPTION ((), '1');
FILE_NAME ('tool_use1.stp', '', (), (), '', '', '');
FILE_SCHEMA (('expressions', 'fbics_combo'));
ENDSEC;
```

```
DATA;
#10  = CLASS_INSTANCES('material', ('aluminum', 'brass', 'steel', 'wax'));
#15  = CLASS_INSTANCES('tool_type', ('ballnose_endmill', 'bullnose_endmill',
       'center_drill', 'cutting_tap', 'corner_rounding_endmill',
       'countersink', 'endmill', 'facemill', 'fly_cutter', 'reamer',
       'roll_form_tap', 'twist_drill'));
#20  = IF_THEN_EXPRESSION(
    /* IF      */ ((('material', .EQUAL., 'aluminum'), .B_AND.,
                 ('tool_type', .EQUAL., 'twist_drill')),
    /* THEN    */  ((275.0, .TIMES., (12.0, .TIMES., 25.4)), .DIVI.,
                  (3.1415, .TIMES., ('diameter', .TIMES., 'factor')))),
    /* START ELSE IFS */ (
    /* ELSE IF */  ((('material', .EQUAL., 'aluminum'), .B_AND.,
                  ('tool_type', .EQUAL., 'endmill')), /* not slot */
    /* THEN    */   ((450.0, .TIMES., (12.0, .TIMES., 25.4)), .DIVI.,
                  (3.1415, .TIMES., ('diameter', .TIMES., 'factor')))),
    /* ELSE IF */ ((('material', .EQUAL., 'brass'), .B_AND.,
                 ('tool_type', .EQUAL., 'twist_drill')),
    /* THEN    */  ((275.0, .TIMES., (12.0, .TIMES., 25.4)), .DIVI.,
                  (3.1415, .TIMES., ('diameter', .TIMES., 'factor')))),
    /* ELSE IF */  ((('material', .EQUAL., 'brass'), .B_AND.,
                  ('tool_type', .EQUAL., 'endmill')), /* not slot */
    /* THEN    */   ((350.0, .TIMES., (12.0, .TIMES., 25.4)), .DIVI.,
                  (3.1415, .TIMES., ('diameter', .TIMES., 'factor')))),
    /* ELSE IF */ ((('material', .EQUAL., 'steel'), .B_AND.,
                 ('tool_type', .EQUAL., 'twist_drill')),
    /* THEN    */  ((65.0, .TIMES., (12.0, .TIMES., 25.4)), .DIVI.,
                  (3.1415, .TIMES., ('diameter', .TIMES., 'factor')))),
    /* ELSE IF */  ((('material', .EQUAL., 'steel'), .B_AND.,
                  ('tool_type', .EQUAL., 'endmill')), /* not slot */
    /* THEN    */   ((200.0, .TIMES., (12.0, .TIMES., 25.4)), .DIVI.,
                  (3.1415, .TIMES., ('diameter', .TIMES., 'factor')))),
    /* ELSE IF */  (('material', .EQUAL., 'wax'),
    /* THEN    */   (5000))),
    /* END ELSE IFS */
    /* ELSE    */ (1111));
#21  = TOOL_USE_RULE(.SPEED., #20);
#22  = IF_THEN_EXPRESSION(
   /* IF      */     (('speed', .BIGR., 5200),
   /* THEN    */      (5200)),
   /* NO ELSE IFS */ (),
   /* ELSE    */     ('speed'));
#23  = TOOL_USE_RULE(.SPEED., #22);
/#30 = IF_THEN_EXPRESSION(
   /* IF      */ ((('material', .EQUAL., 'aluminum'), .B_AND.,
                 ('tool_type', .EQUAL., 'twist_drill')),
   /* THEN    */  ((('speed', .TIMES., 0.005), .TIMES., 'diameter'))),
   /* START ELSE IFS */ (
   /* ELSE IF */  ((((('material', .EQUAL., 'aluminum'), .B_AND.,
                  ('tool_type', .EQUAL., 'endmill')), .B_AND.,
                  (('diameter', .TIMES., 'factor'), .BIGR., 25.4)),
   /* THEN    */   ((('speed', .TIMES., 0.005),
                  .TIMES., 'number_of_flutes')),
```

```
   /* ELSE IF */  (((‘material’, .EQUAL., ‘aluminum’), .B_AND.,
                    (‘tool_type’, .EQUAL., ‘endmill’)),
   /* THEN    */  (((‘speed’, .TIMES., 0.005), .TIMES., ‘diameter’),
                   .TIMES., ‘number_of_flutes’)),
   /* ELSE IF */  ((‘material’, .EQUAL., ‘wax’),
   /* THEN    */  (((‘speed’, .TIMES., 0.01), .TIMES., ‘diameter’),
                   .TIMES., ‘number_of_flutes’))),
   /* END ELSE IFS */
   /* ELSE    */  (2.222));

#31  = TOOL_USE_RULE(.FEED., #30);

/* stepover_rule */
#40  = IF_THEN_EXPRESSION(
   /* IF      */    ((‘material’, .EQUAL., ‘wax’),
   /* THEN    */     (‘diameter’, .TIMES., 0.8)),
   /* NO ELSE IFS */ (),
   /* ELSE    */    (‘diameter’, .TIMES., 0.5));
#41  = TOOL_USE_RULE(.STEPOVER., #40);
#50  = IF_THEN_EXPRESSION(
   /* IF      */    ((‘material’, .EQUAL., ‘wax’),
   /* THEN    */     (‘diameter’, .TIMES., 2.0)),
   /* START ELSE IFS */ (
   /* ELSE IF */    ((‘tool_type’, .EQUAL., ‘twist_drill’),
   /* THEN    */     (‘diameter’)),
   /* ELSE IF */    (((‘tool_type’, .EQUAL., ‘endmill’), .B_OR.,
                    (‘tool_type’, .EQUAL., ‘ballnose_endmill’)),
   /* THEN    */     (‘diameter’, .TIMES., 0.5))),
   /* END ELSE IFS */
   /* ELSE    */   (4.444));
#51  = TOOL_USE_RULE(.PASS_DEPTH., #50);
#60  = IF_THEN_EXPRESSION(
   /* IF      */    ((‘material’, .EQUAL., ‘wax’),
   /* THEN    */     (0)),
   /* NO ELSE IFS */ (),
   /* ELSE    */    (1));
#61  = TOOL_USE_RULE(.FLOOD., #60);
#70  = IF_THEN_EXPRESSION(
   /* IF      */    ((1),
   /* THEN    */     (0)),
   /* NO ELSE IFS */ (),
   /* ELSE    */    (0));
#71  = TOOL_USE_RULE(.MIST., #70);
#80  = TOOL_USE_RULES((#10, #15), (#21, #23, #31, #41, #51, #61, #71));
ENDSEC;

END-ISO-10303-21;
```