



PeopleSoft PepperCode

PeopleSoft PepperCode

SKU MTPCr8SP1B 1200

PeopleBooks Contributors: Teams from PeopleSoft Product Documentation and Development.

Copyright © 2001 by PeopleSoft, Inc. All rights reserved.

Printed in the United States of America.

All material contained in this documentation is proprietary and confidential to PeopleSoft, Inc. and is protected by copyright laws. No part of this documentation may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, including, but not limited to, electronic, graphic, mechanical, photocopying, recording, or otherwise without the prior written permission of PeopleSoft, Inc.

This documentation is subject to change without notice, and PeopleSoft, Inc. does not warrant that the material contained in this documentation is free of errors. Any errors found in this document should be reported to PeopleSoft, Inc. in writing.

The copyrighted software that accompanies this documentation is licensed for use only in strict accordance with the applicable license agreement which should be read carefully as it governs the terms of use of the software and this documentation, including the disclosure thereof.

PeopleSoft, the PeopleSoft logo, PeopleTools, PS/nVision, PeopleCode, PeopleBooks, and Vantive are registered trademarks, and *PeopleTalk* and "People power the internet." are trademarks of PeopleSoft, Inc. All other company and product names may be trademarks of their respective owners.

Contents

About This PeopleBook

Before You Begin	xiv
Related Documentation	xiv
Documentation on the Internet.....	xiv
Documentation on CD-ROM	xv
Hardcopy Documentation	xv
Typographical Conventions and Visual Cues.....	xv
Comments and Suggestions.....	xvii

Chapter 1

Understanding PepperCode

Defining PepperCode.....	1-1
Comparing PepperCode and C/C++	1-3
Comparing PepperCode and C/C++ Classes.....	1-3
Comparing PepperCode Actions and C/C++ Functions.....	1-4
Customizing Planning Software	1-5

Chapter 2

Getting Started with PepperCode

Writing Sample PepperCode Constructs	2-1
Understanding Program Elements	2-4

Chapter 3

Understanding PepperCode Basics

Writing .spl Files	3-1
Writing PepperCode #include Statements	3-1
Rules for Inclusion and Writing #include Statements.....	3-2
Using two files that include each other	3-4
Using #include instead of forward declarations.....	3-4
#include and pre-8.0 versions.....	3-5
Understanding Scopes and Identifiers	3-6
Writing PepperCode Comments	3-7
Writing PepperCode Documentation Comments	3-7

Understanding #document error messages.....	3-8
Format for #document comments	3-9
Using --doc and --header with documentation comments.....	3-11
Writing PepperCode Notice Comments	3-11
Understanding PepperCode Data Types	3-11
Understanding PepperCode Performance Considerations	3-14
Using PepperCode Naming Conventions	3-15

Chapter 4

Understanding PepperCode Classes

Writing New Class Definitions.....	4-3
Understanding Default Values	4-5
Understanding Multiple Inheritance.....	4-5
Specializing Slots	4-5
Understanding Dot Notation	4-6
Declaring Classes.....	4-6
Forward Class Declarations	4-6
Slot Clause List Statements.....	4-7
Slot Declaration Statements	4-8
Understanding Instance and Class Slots	4-9
Writing Temporary Objects	4-9
Using Predefined Classes	4-10
Using Instance Names	4-12

Chapter 5

Understanding PepperCode Actions

Writing Action Definitions	5-2
Incomplete and Forward Declarations	5-4
Avoiding Incomplete Declarations.....	5-6
Matching Parameters and Parameter Lists	5-6
Using context:, no_context:, and readonly:.....	5-6
Writing Action Parameters	5-7
Using required: Keyword as Explicit Default Value.....	5-8
Understanding non-local action parameters	5-9
Understanding Parameter Defaults.....	5-9
Understanding How Parameters Behave With Execute	5-11
Action Parameters are No Longer Static.....	5-12
Writing Schemas.....	5-13
Action Schema Declarations and Definitions	5-15
Declaring Actions: Forward (or Incomplete) Action Declarations.....	5-15

Executing Actions.....	5-18
New Rule for Invoking Action.....	5-19
Passing Output in execute Statement	5-20
Writing Methods.....	5-21
Implementing A Method: Example 1	5-21
Implementing A Method: Example 2.....	5-23
Implementing A Method: Example 3	5-27
Understanding Context	5-33
Accessing Action Status	5-40
Understanding How Actions Are Executed.....	5-41
Action Execution & Definitions.....	5-42
Using Transaction Logs	5-42

Chapter 6

Writing Control Statements

Writing Assignment Statements	6-1
Writing if-else Statements	6-2
Writing while Statements	6-2
Writing foreach Statements	6-3
Writing execute Statements	6-5
Writing succeed, fail, or leave Statements.....	6-5
Writing break and continue in Loops	6-6
Writing Enumerations in Loops.....	6-7
Using Dot Notation in Expressions	6-9

Chapter 7

Writing Osets

Writing Osets with Action Parameters	7-4
Writing Osets in Loops.....	7-5
Writing Osets with the foreach Statement	7-6

Chapter 8

Writing Arrays

Writing Associative Arrays	8-1
Writing Nonassociative Arrays.....	8-3
Understanding Array operations.....	8-5
Writing Arrays of Arrays.....	8-5
Writing Statements Involving Arrays	8-8
Writing Array Accesses	8-8
Writing Arrays Indexed by Float	8-8

Chapter 9

Understanding Histories And Side Effects

Understanding the History Abstract Data Structure	9-1
Representing Availability of a Capacity Resource.....	9-2
Understanding The History Data Structure	9-4
Understanding History Data Structure Elements	9-4
Understanding A History Elements List	9-4
Understanding An Example of History Object	9-5
Understanding An Example of Interval Implementation	9-5
Understanding GetValue Implementation.....	9-5
Finding Maximum.....	9-6
Understanding Side Effects and Persistence.....	9-8
Understanding The Effect of Supply/Constraint and Capacity/Inventory on Side Effects	9-8
Understanding the Scheduling Classes: Resource and Task	9-9
Understanding the Resource Class.....	9-9
Understanding Tasks.....	9-10
Understanding Resource Supplies and Constraints.....	9-10
Understanding the Effect of Resource Supplies and Constraints on Histories	9-11
Programming for Side Effects: The side_effect Keyword	9-12
Understanding Schedules.....	9-13

Chapter 10

Understanding Operators And Functions

Understanding Infix and Intrinsic Operators and Functions.....	10-1
Understanding SET_EPSILON and SET_FLOAT_FORMAT	10-6
Using EQ With Strings.....	10-8
Accessing C/C++ Functions	10-8
PepperCode Data Types in cpp_function Statements	10-9
Rules for Passing Arguments	10-9
Typedefs Used With C++ Functions.....	10-10
Using PepperCode Runtime Functions	10-11
GET_NAME_OF_CLASS.....	10-22
TYPEP Example	10-23
Using Expression Comparisons	10-23
Using Upstairs Objects Functions	10-24
Using String Functions for National Language Support	10-27
Using Postpone Side Effects Functions	10-30
Using Functions That Query From PepperCode	10-33

Using History Functions	10-35
Using Dump Functions	10-44

Chapter 11

Writing PepperCode Applications

Writing a PepperCode Class	11-1
Naming A Class	11-1
Naming Class Slots	11-1
Adding An Action To A Class	11-1
Adding Default Values To A Class	11-2
Specializing Class Slots	11-2
Using Casting	11-5
Writing a PepperCode Action	11-6
Using no_context	11-7
Avoiding Static Parameters	11-7
Checking The Output Variable On An Action	11-8
Grouping Action Parameters	11-9
Writing A PepperCode Transaction	11-11
Starting Transaction Names With transaction_	11-12
Using The Action Schema Transaction	11-12
Putting Minimal Code Into A Transaction	11-12
Including No Instances, Classes, Histories, Or Actions	11-12
Using Default Values For Input Parameters	11-12
Performing Error Checking	11-13
Using #document and #end_document	11-17
Writing A PepperCode Method	11-18
Writing Actions That Dispatch The Method	11-18
Implementing Input And Output Parameters	11-19
Including The Object As An Argument	11-19
Casting The Inner Object To The Class	11-19
Writing a C++ Utility	11-20
Checking That A Corresponding Function Is Not Defined	11-20
Putting C++ Code In The Proper Location	11-20
Capitalizing C++ Function Names	11-21
Providing Meaningful PepperCode Types	11-21
Using RPS_IMPORT When Defining External C++ Functions	11-22
Adding and Retrieving Documentation	11-23
Using #include Files	11-24
Customizing and Displaying Class Names	11-25
Customizing PepperCode Methods And Actions	11-26

Replacing Standard Method Actions	11-26
Adding Method Slots	11-28
Adding A Constraint.....	11-32
Creating the Class Shipset_Milestone_Constraint	11-33
Writing an Action to Display Information	11-34
Writing An Action To Define The Penalty	11-34
Writing An Action To Specify The Repair	11-37
Writing An Action To Specify The Time Interval	11-40
Writing An Action That Creates A Constraint Object	11-40

Chapter 12

Compiling And Linking PepperCode

Setting Up and Using Your Own PepperCode Sandbox	12-1
Running the Compiler.....	12-2
Solaris example	12-3
HP-UX example	12-4
Digital Unix (OSF/1) and Linux examples	12-4
NT example	12-4
Command-line rules in detail	12-6
Installation and Configuration Issues	12-6
LD_LIBRARY_PATH.....	12-6
List of Necessary Files	12-7
.splrc	12-7
Compiler Options (For Use During Installation)	12-8
PepperCode Compiler Reference.....	12-10
Command-line Rules in Detail.....	12-11
Most-Used Compiler Options	12-11
Options That Dictate Which Compiler or Linker to Run	12-12
Options Used When Compiling PepperCode.....	12-13
Options Used Only When Compiling C++ Source Code.....	12-16
Options to be Used With --make_program Option	12-16
Machine-Specific Escape Clause	12-17
Options for Compiler Maintenance.....	12-17
Using Hush	12-18

Chapter 13

Understanding PepperCode Syntax

Chapter 14

Debugging PepperCode

Avoiding Common Mistakes	14-1
Troubleshooting Guide	14-6
Compiler Frequently Asked Questions (FAQ).....	14-6
Q: How does one compile PepperCode files that #include each other?.....	14-6
Q: Why doesn't an enumeration constant have an integer value?	14-7
Q: The rules have changed for declaring actions locally. What about classes?	14-7
Error Message Reference	14-9
Errors (That Stop Compilation)	14-9
Warnings (These Don't Stop Compilation)	14-20
Using Debugging Tools	14-20
Using The Action Interpreter	14-21
Using Action Debug Tracing	14-23
Using The Action Debug Tracing Transaction and C++ Function	14-24
Setting Action Debug Tracing Behavior	14-24
Enabling and Disabling Action Debug Tracing	14-25
Understanding Action Debug Tracing Output	14-26
Creating Debug Messages With The MSG Function.....	14-29
Using Debugging Functions.....	14-31
describe	14-32
describe_all	14-32
describe_one.....	14-34
describe_by_name.....	14-36
describe_by_uid	14-37
how_many	14-39
list_objects	14-39
display_rhistory.....	14-40
display_rinitial_history	14-40
display_ahistory	14-40
display_chistory	14-40
Other Debugging Functions	14-41
Using Debugging Actions.....	14-41
Understanding Key Terms	14-41
Setting The Debugging Message Level	14-42
Running The Debugging Actions.....	14-42
Understanding The Debugging Action Categories	14-43
Displaying PepperCode Instance Information	14-43
Displaying History Information	14-43

Displaying Task Reschedule Information	14-43
Debugging Side Effects.....	14-44
Displaying Time Period Information	14-44
Miscellaneous Debugging Actions	14-44
Deciding Which Debugging Action To Use	14-45
Understanding Debugging Action Descriptions.....	14-46
transaction_describe_all.....	14-46
transaction_describe_one	14-46
transaction_describe_by_name	14-47
transaction_describe_by_uid.....	14-47
transaction_how_many	14-47
transaction_list_objects.....	14-47
transaction_display_rhistory	14-47
transaction_display_rinitial_history	14-48
transaction_display_ahistory.....	14-48
transaction_display_chistory.....	14-48
transaction_set_intersect_debug_level.....	14-49
transaction_printf	14-49
transaction_printf_with_current_time.....	14-49
transaction_start_of_day	14-49
transaction_end_of_day	14-49
transaction_start_of_week.....	14-50
transaction_end_of_week.....	14-50
transaction_start_of_month.....	14-50
transaction_end_of_month.....	14-50
display_violated_constraints	14-50
display_resource_constraints	14-51
display_resource_supplies.....	14-51
retract_resource_constraint	14-51
assert_resource_constraint	14-51
retract_resource_supply	14-52
assert_resource_supply	14-52
retract_task_side_effects	14-52
assert_task_side_effects	14-52
retract_resource_side_effects.....	14-53
assert_resource_side_effects	14-53
repair_me	14-53
object_is_alive.....	14-53
resource_info.....	14-54
create_some_objects	14-54

delete_some_objects	14-54
Understanding Debug Command Files	14-55
Using Sanity Checks.....	14-61
Understanding What Sanity Checks Do and Don't Do.....	14-61
Using Sanity Checks	14-61
Understanding Each Sanity Check	14-62
A Parent Task Must Have Subtasks	14-62
Work Duration Check For Unsplittable Leaf Tasks.....	14-62
A Calendar Must Have Computed Legal Time.....	14-63
A Resource Constraint Must Have Quantity >= 0.0	14-63
A Resource Supply Must Have Quantity >= 0.0.....	14-63
Start And End Time Checks Of Effective Entries.....	14-63
A Routing Entry Must Have Quantity >= 0.0	14-64
A Routing Entry Must Match A Routing Step	14-64
A Bor Entry Must Have A Valid Equipment Class	14-64
A Build Option Must Have At Least One Routing Step	14-65
A Build Option Must Supply An Item (Part) For All Time	14-65
A Build Option Should Have Only One Primary Order Bor	14-65
A Build Option Should Have Only One Primary Operation Bor Per Routing Step.....	14-65
An Inventory Item Must Have A Way To Be Replenished	14-65
A Sales Order Must Have Sales Order Lines	14-65
A Purchase Order Must Have Purchase Order Lines	14-66
An Equipment Resource Must Have Enough Capacity To Repair Any One Of Its Equipment Constraints	14-66
Understanding Potential Sanity Checks	14-66
Every Product Must Map To An Inventory Item	14-66
Sourcing Logic Checks	14-66
Understanding Sanity Check Output.....	14-67
transaction_mfg_sanity_check (:verbose 0 :filename "")	14-67
transaction_mfg_sanity_check(:verbose 0 :filename "")	14-68
transaction_mfg_sanity_check(:verbose 1 :filename "")	14-69

Index

ABOUT THIS PEOPLEBOOK

This PeopleBook, PeopleSoft PepperCode, provides you with the information you need to write programs in the PepperCode programming language.

You should be familiar with navigating around the system and adding, updating, and deleting information using PeopleSoft windows, menus, and pages. You should also be comfortable using the Microsoft® Windows 95 or Windows NT graphical user interface.

Because we assume you already know how to navigate around the PeopleSoft system, much of the information in this book is not procedural. That is, it does not typically provide step-by-step instructions on using tables, pages, and menus. Instead we provide you with all the information you need to use the system most effectively, and to customize the documentation to your organizational or departmental needs. This book expands on the material covered in PeopleSoft training classes.

Understanding PepperCode provides an overview of PepperCode, also known as the Scheduling Programming Language (SPL).

Getting Started with PepperCode introduces you to PepperCode with a sample program that creates two objects representing bicycles. The program tests itself by printing information about them.

Understanding PepperCode Basics gives basic information about how to write PepperCode (.spl) files, #include statements, and comments, and it discusses PepperCode data types, performance considerations, and naming conventions.

Understanding PepperCode Classes explains PepperCode classes, which provide definitions of the data stored in an object.

Understanding PepperCode Actions explain PepperCode actions, which are similar to C functions or Pascal procedures, but have very different semantics for memory allocation and the lifetimes of variables and changes to variables.

Writing Control Statements explains how to write PepperCode control statements, such as assignment (=), if-else, and while.

Writing Osets explains how to write PepperCode osets, which behave like a list.

Writing Arrays explains how to write PepperCode arrays, which behave like a list.

Understanding Histories And Side Effects explains histories and side effects, and describes how and when to use the side_effect keyword in slot declarations.

Understanding Operators And Functions lists and describes how to use the PepperCode intrinsic operators and functions. It also describes how to access and use C/C++ functions.

Writing PepperCode Applications provides guidelines for creating PepperCode applications, such as writing a PepperCode class, action, transaction, and method.

Compiling And Linking PepperCode describes how to compile your code and link it with existing Planning software for testing purposes.

Understanding PepperCode Syntax describes the PepperCode syntax recognized by the parser in the current PepperCode compiler.

Debugging PepperCode explains the debugging tools for PepperCode, including symbolic debuggers for C++, the debugging functions that you can use to print methods and their descriptions, and other methods.

This section describes information you should know before you begin working with PeopleSoft products and documentation, including PeopleSoft-specific documentation conventions, information specific to PeopleTools, how to order additional copies of our documentation, and so on.

Before You Begin

To benefit fully from the information covered in this book, you need to have a basic understanding of how to use PeopleSoft applications. We recommend that you complete at least one PeopleSoft introductory training course.

You should be familiar with navigating around the system and adding, updating, and deleting information using PeopleSoft windows, menus, and pages. You should also be comfortable using the World Wide Web and the Microsoft® Windows or Windows NT graphical user interface.

Related Documentation

To add to your knowledge of PeopleSoft applications and tools, you may want to refer to the documentation of the specific PeopleSoft applications your company uses. You can access additional documentation for this release from PeopleSoft Customer Connection (www.peoplesoft.com). We post updates and other items on Customer Connection, as well. In addition, documentation for this release is available on CD-ROM and in hard copy.



Important! Before upgrading, it is *imperative* that you check PeopleSoft Customer Connection for updates to the upgrade instructions. We continually post updates as we refine the upgrade process.

Documentation on the Internet

You can order printed, bound versions of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM. You can order additional copies of the PeopleBooks CDs through the Documentation section of the PeopleSoft Customer Connection Web site:
<http://www.peoplesoft.com/>

You'll also find updates to the documentation for this and previous releases on Customer Connection. Through the Documentation section of Customer Connection, you can download files to add to your PeopleBook library. You'll find a variety of useful and timely materials, including updates to the full PeopleSoft documentation delivered on your PeopleBooks CD.

Documentation on CD-ROM

Complete documentation for this PeopleTools release is provided in HTML format on the PeopleTools PeopleBooks CD-ROM. The documentation for the PeopleSoft applications you have purchased appears on a separate PeopleBooks CD for the product line.

Hardcopy Documentation

To order printed, bound volumes of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM, visit the PeopleSoft Press Web site from the Documentation section of PeopleSoft Customer Connection. The PeopleSoft Press Web site is a joint venture between PeopleSoft and Consolidated Publications Incorporated (CPI), our book print vendor.

We make printed documentation for each major release available shortly after the software is first shipped. Customers and partners can order printed PeopleSoft documentation using any of the following methods:

Internet

From the main PeopleSoft Internet site, go to the Documentation section of Customer Connection. You can find order information under the Ordering PeopleBooks topic. Use a Customer Connection ID, credit card, or purchase order to place your order.

PeopleSoft Internet site: <http://www.peoplesoft.com/>.

Telephone

Contact Consolidated Publishing Incorporated (CPI) at **800 888 3559**.

Email

Email CPI at callcenter@conpub.com.

Typographical Conventions and Visual Cues

To help you locate and interpret information, we use a number of standard conventions in our online documentation.

Please take a moment to review the following typographical cues:

`monospace font`

Indicates PeopleCode.

Bold

Indicates field names and other page elements, such as buttons and group box labels, when these elements are documented below the page on which they appear. When we refer to these elements elsewhere in the documentation, we set them in Normal style (not in bold).

We also use boldface when we refer to navigational paths, menu names, or process actions (such as **Save** and **Run**).

Italics

Indicates a PeopleSoft or other book-length publication. We also use italics for *emphasis* and to indicate specific field values. When we cite a field value under the page on which it appears, we use this style: *field value*.

We also use italics when we refer to words as words or letters as letters, as in the following: Enter the number *0*, not the letter *O*.

KEY+KEY

Indicates a key combination action. For example, a plus sign (+) between keys means that you must hold down the first key while you press the second key. For ALT+W, hold down the ALT key while you press W.

Jump links

Indicates a jump (also called a link, hyperlink, or hypertext link). Click a jump to move to the jump destination or referenced section.

Cross-references

The phrase For more information indicates where you can find additional documentation on the topic at hand. We include the navigational path to the referenced topic, separated by colons (:). Capitalized titles in *italics* indicate the title of a PeopleBook; capitalized titles in normal font refer to sections and specific topics within the PeopleBook. Cross-references typically begin with a jump link. Here's an example:

For more information, see Documentation on CD-ROM in *About These PeopleBooks*: Related Documentation.

- Topic list

Contains jump links to all the topics in the section. Note that these correspond to the heading levels you'll find in the Contents window.



Name of Page or
Dialog Box

Opens a pop-up window that contains the named page or dialog box. Click the icon to display the image. Some screen shots may also appear inline (directly in the text).



Text in this bar indicates information that you should pay particular attention to as you work with your PeopleSoft system. If the note is preceded by **Important!**, the note is crucial and includes information that concerns what you need to do for the system to function properly.



Text in this bar indicates For more information cross-references to related or additional information.



Text within this bar indicates a crucial configuration consideration. Pay very close attention to these warning messages.

Comments and Suggestions

Your comments are important to us. We encourage you to tell us what you like, or what you would like changed about our documentation, PeopleBooks, and other PeopleSoft reference and training materials. Please send your suggestions to:

PeopleTools Product Documentation Manager
PeopleSoft, Inc.
4460 Hacienda Drive
Pleasanton, CA 94588

Or send comments by email to the authors of the PeopleSoft documentation at:

DOC@PEOPLESOFT.COM

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions. We are always improving our product communications for you.

CHAPTER 1

Understanding PepperCode

This section provides an overview of PepperCode, also known as the Scheduling Programming Language (SPL).

Defining PepperCode

PepperCode is a high-level, object-oriented programming language. The language is optimized for use with Planning scheduling applications. It has the following features:

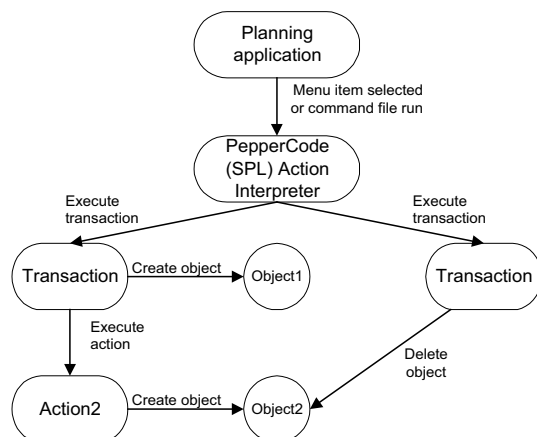
- PepperCode is a high-level language, so you can spend more time writing the application and less time worrying about low-level coding details. For example:
 - Pointers and pointer manipulation are invisible to you, which eliminates bugs related to uninitialized pointers, dangling pointers, and misallocated memory.
 - A construct similar to a linked list is built into the language.
 - Memory allocation and deallocation are hidden from you.
 - All data is automatically initialized so that execution is predictable.
- PepperCode provides some constructs aimed specifically at scheduling algorithms. A mechanism called context makes it easy to try out various combinations of values before making global changes to the state of the entire system. Each experiment is independent of the others; when experiments are complete, changes that were part of the failed experiments can be discarded while changes that represent the optimal combination of values are accepted.
- PepperCode provides a more dynamic object-oriented programming environment than statically compiled languages like C++. You can create new classes by creating subclasses at execution time, change the default values of members at execution time, and query any class to get a list of subclasses or instances that currently exist.
- PepperCode code is easy to transport to any system that provides a standard C++ compiler, since PepperCode code is automatically converted to C++ code during compilation.
- PepperCode offers a feature called *side effects*, where a dependent member is computed from independent members through a side-effect function. Any change to an independent member causes the dependent member to be recomputed.
- Built-in features support a client-server architecture.

In addition, Planning software, which is written in PepperCode, has the following features that make it easy to customize:

- An extensive group of predefined object hierarchies and functions are provided that enable the easy specialization of the application software.
- Application menus can be changed simply by placing a command in a file.
- Transaction logging enables you to restore a system to its previous state when needed.

A Planning application runs PepperCode code. The application can run PepperCode code when a user selects a menu item or uses the menu system to run a command file. In addition, during debugging phases, a diagnostic tool enables you to execute PepperCode code from a command line.

PepperCode code has two main constructs: classes and actions. Classes define objects. Actions can create, delete, and modify objects, and can execute other actions. When a PepperCode application wants to perform an operation, it instructs the PepperCode (or SPL) Action Interpreter to run an action, as in the following example. An action that is designed to be run in a command file or to be used by a programmer customizing the software is called a transaction—to distinguish it from actions that are meant to be hidden within the system. From the viewpoint of PepperCode, however, there is no distinction between an action and a transaction.



Example of running PepperCode code

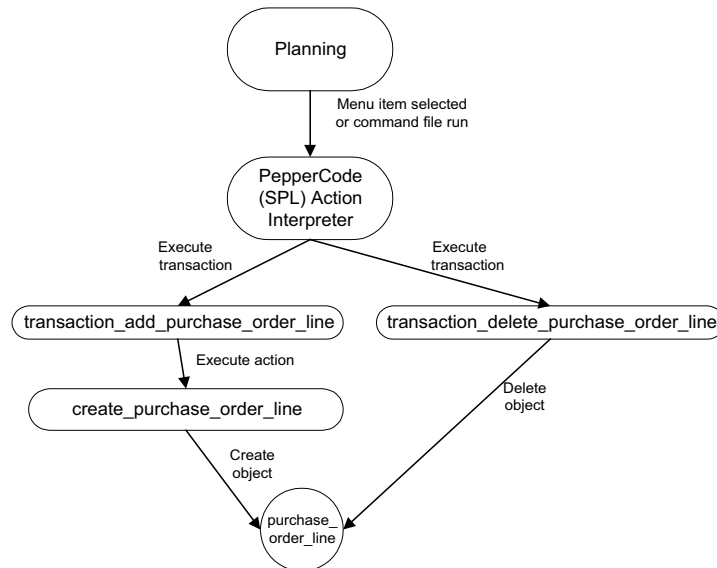
Actions can change the values of a class or create a new class through subclassing. However, there is no way to delete a class during runtime.



Because classes can be modified during runtime, you could think of a class as an object. However, in this documentation, the term object applies only to an instance of a class.

An action can query a parent class to retrieve its child classes, which makes it easier to keep track of classes in your code and reduces the possibility of errors.

Following is an example of how a manufacturing application might enable a user to add a line to a purchase order. The user chooses a menu item that causes the Action Interpreter to execute `transaction_add_purchase_order_line`. This transaction executes the action `create_purchase_order_line`, which creates the object `purchase_order_line`. Later, the user chooses a menu item to delete the line, which causes the Action Interpreter to execute `transaction_delete_purchase_order_line`. This transaction deletes the object `purchase_order_line`.



Manufacturing example

Comparing PepperCode and C/C++

Because PepperCode code is similar to C++ code, it's easier to look at the differences instead of the ways they are alike.

Comparing PepperCode and C/C++ Classes

PepperCode classes differ from C++ classes in the following ways:

- PepperCode uses the term *slot* instead of the C++ *data member* or *member function* terms. `A slot can hold:
 - data (like a C++ data member)
 - an action (like a C++ member function)
 - an instance of a class (like a C++ member of type class)
 - another class

- Unlike C++, methods are stored on ordinary slots like data values are. All PepperCode slots are public.
- A PepperCode class slot is like a C++ static data member or static member function. All instances of the class read and write the same value for that slot.
- As mentioned earlier, PepperCode is more abstract than C++. It doesn't provide ways to specify how class slots are represented in memory in terms of offsets, ordering, addresses, and so on. Also, PepperCode has no bit fields or unions. As a result, a program is less likely to have a memory error, since most memory allocation and deallocation is handled by PepperCode.
- Another way PepperCode is safer than C++ is that it has no uninitialized slots. Every slot has a default value, either specified explicitly in the code or provided by the compiler.

Comparing PepperCode Actions and C/C++ Functions

A PepperCode action is similar to a function in C++, C, or Pascal. As mentioned previously, if a PepperCode class has a slot of type action, the action on that slot acts like a C++ member function or method.

PepperCode actions differ from C++ functions in the following ways:

- A declaration for a PepperCode action must list local parameters along with other parameters.
- A PepperCode action can have more than one output parameter, while a C++ function can have only one return value.
- An action cannot appear within an expression. It must be invoked with the execute statement.
- In an action invocation, the parameters can be listed in any order. Before each parameter is a keyword, which is the name for the parameter as it appears in the definition of the action.
- Input parameters can have default values, so it's not necessary to provide all of them. But defaults behave differently in PepperCode than in C++.
- Actions are invoked within other actions by a parameter of type input or local, not by their original name.
- Output values aren't copied from a formal argument to an actual argument, as in C++. Instead, the action behaves as if it were a class and the outputs behave as if they were slots on a class.
- As mentioned earlier, PepperCode has a powerful feature called context. Changes to the values of slots on objects can be accepted or rejected at the appropriate time. Binding an action to multiple contexts lets your application try independent experiments and postpone selecting the optimum outcome until the experiment is complete.

Customizing Planning Software

Your Planning software, written in PepperCode, is customizable so it can meet your own specialized requirements. You can add new menu items or tailor existing menu items. In addition, you can create your own custom software, compile it, and link it to the existing Planning Scheduler product, MFG product, or both.

You can do the following in your custom software:

- Create new transactions. Wrapper transactions extend the behavior of existing transactions: you simply create a new transaction that calls the existing Planning transaction and add the additional functionality you need.
- Create new classes that inherit behavior from existing classes in the Planning Scheduler product, MFG product, or both.
- Create new software to override inherited behavior in classes. This software simply needs to use the schema specified for the action slot in the class.

This documentation describes how to create actions, classes, and methods for your custom software.

CHAPTER 2

Getting Started with PepperCode

This section introduces you to PepperCode with a sample program that creates two objects representing bicycles. The program tests itself by printing information about them.

The example contains some constructs that are familiar to C++ programmers, other constructs that may be unfamiliar to C++ programmers but familiar to programmers of other object-oriented languages, and one construct that is unique to PepperCode. PepperCode files use a .spl extension.

Writing Sample PepperCode Constructs

```
// Include the .spl file containing PepperCode runtime functions.

// By convention, these functions appear in all uppercase letters in code.

#include "cpp_utility.spl"

// Create an enumeration containing the possible bike materials.

enum material { STEEL, ALUMINUM, CARBON_FIBER, TITANIUM, OTHER };

// Define a basic class for a vehicle.

class Vehicle : Base_Class {

    int serial_number

    int passengers

    int price

};

slot Vehicle.passengers { default: 4 };

// Derive the class Bicycle from the class Vehicle.

// Add two new slots, in addition to those from the Vehicle class.

// Override the default number of passengers to a more realistic value

// for a bike.

class Bicycle: Vehicle {

    string model_name
```

```

    enum<material> frame_material

};

slot Bicycle.frame_material{ default: STEEL };

slot Bicycle.passengers{ default: 1 };

// Define a procedure to create an instance of the Bicycle class
// (or one of its subclasses). The instance of the class is an object.

action create_bicycle

    (input: int serial_number,

     input: string model_name,

     input: string class_name,

     output: instance<Bicycle> new_bike,

     no_context:)
{
    // Create an object with the CREATE_OBJECT function.

    // model_name is the name of the object.

    // (All named objects must have a unique name.)

    // class_name is the name of the class the object belongs to.

    new_bike = CREATE_OBJECT(model_name, class_name);

    new_bike.serial_number = serial_number;

    new_bike.model_name = model_name;

    succeed();
}

// Create instances of the classes Atb and Bicycle.

// Test it by generating a list of the instances of Bicycle and iterating
// through the list—printing the serial number of each Bicycle or Atb.

action spl_main

    (input: int argc,

     input: oset[string] argv,

     input: string identity,

```

```

        local: oset[instance<Bicycle>] list)
{
    // Create a subclass at runtime. Atb is the name of the new class,
    // which is a subclass of Bicycle.
    CREATE_SUBCLASS("Atb", "Bicycle");

    // Create the bicycles. The object names are stumphopper and vamenos.
    execute create_bicycle(:serial_number 44475656,
        :model_name "stumphopper",
        :class_name "Atb");

    execute create_bicycle(:serial_number 55572323,
        :model_name "vamenos",
        :class_name "Bicycle");

    // As a test, print information about the objects.
    // list is an oset of instances that are "filled in."
    // The next argument is the name of the class whose descendants
    // you want to list.
    // 1 specifies that you want instances (as opposed to 0 for classes).
    GET_DESCENDANTS(list, Bicycle, 1);

    foreach item in list
        PRINTF("%s serial number=%d, model_name=%s\n",
            item.class_name, item.serial_number, item.model_name);

    succeed();
}

```

When the program runs, it prints the class an object is derived from, followed by the serial number and model name (which in this case is the object name) for the object, as follows:

```

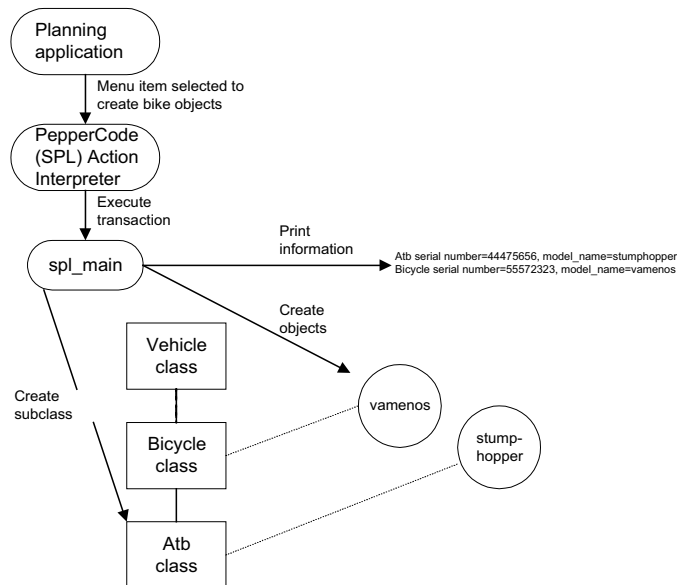
Atb serial number=44475656, model_name=stumphopper

Bicycle serial number=55572323, model_name=vamenos

```

Understanding Program Elements

If this code was compiled and then linked with Planning software modules, and a menu item was created to execute spl_main, the program would run as illustrated here.



Running the sample code

The first line of the program should be familiar to C++ programmers:

```
#include "cpp_utility.spl"
```

This statement causes the file `cpp_utility.spl` to be included in the program. This file contains declarations, including those for PepperCode runtime functions such as `CREATE_OBJECT`, `CREATE_SUBCLASS`, `GET_DESCENDANTS`, and `PRINTF`.

C++ programmers should also recognize the declaration of the enumerated type:

```
enum material { STEEL, ALUMINUM, CARBON_FIBER, TITANIUM, OTHER };
```

The Bicycle class defined later in the code uses the enumeration for its material value.

Next, two classes are defined: first a basic class for any vehicle and then a more specific class for bicycles that is derived from the Vehicle class. Classes are the prototypes for object instances; they can inherit from one or more other classes to provide greater specialization. The data values, or slots, defined for the Vehicle class apply to the Bicycle class, which inherits from it. So the Bicycle class contains slots for `serial_number`, `passengers`, `price`, `model_name`, and `frame_material`.

```
class Vehicle : Base_Class {
    int serial_number
    int passengers
```

```

        int price
    };

    slot Vehicle.passengers { default: 4 };

    class Bicycle: Vehicle {

        string model_name

        enum<material> frame_material

    };

    slot Bicycle.frame_material{ default: STEEL };

    slot Bicycle.passengers{ default: 1 };

```

There are no uninitialized slots in PepperCode. If you don't provide a default, PepperCode provides one for you. In this code, a default is defined for passengers in the Vehicle class. In the Bicycle class definition, this default is changed to a more appropriate value for a bike; a default is also provided for frame_material. The default initialization of slots is similar to that for C++ members.

Slots are referred to using dot notation—for example, Vehicle.passengers refers to the passengers slot in the Vehicle class, while Bicycle.passengers refers to the passengers slot in the Bicycle class.

In addition to classes, the sample program also has two actions, which resemble C++ functions. Actions can contain declarations for both input parameters and local variables, followed by the action body containing PepperCode code. To exit an action, a succeed or fail statement is supplied. This is how the context mechanism is implemented: you can try out various combinations of values before making global changes to the state of the entire system. Changes that were part of the failed experiments can be discarded with the fail statement.

The action create_bicycle is used in the spl_main action to create objects of the Bicycle class:

```

action create_bicycle

    (input: int serial_number,

     input: string model_name,

     input: string class_name,

     output: instance<Bicycle> new_bike,

     no_context:)

{

    new_bike = CREATE_OBJECT(model_name, class_name);

    new_bike.serial_number = serial_number;

    new_bike.model_name = model_name;

```

```

        succeed();
    }

    action spl_main

        (input: int argc,

         input: oset[string] argv,

         input: string identity,

         local: oset[instance<Bicycle>] list)

    {

        CREATE_SUBCLASS("Atb", "Bicycle");

        execute create_bicycle(:serial_number 44475656,

                               :model_name "stumphopper",

                               :class_name "Atb");

        execute create_bicycle(:serial_number 55572323,

                               :model_name "vamenos",

                               :class_name "Bicycle");

        GET_DESCENDANTS(list, Bicycle, 1);

        foreach item in list

            PRINTF("%s serial number=%d, model_name=%s\n",

                  item.class_name),

                  item.serial_number, item.model_name);

        succeed();
    }

```

C++ programmers are likely to be surprised that the program creates a new derived class at execution time (the call to `CREATE_SUBCLASS`). And although creating instances of a class at execution time isn't particularly unusual in C++ (the call to `CREATE_OBJECT` is similar to the C++ operator `new`), the semantics of an instance are very different: C++ has no global or local variables of type class. All instances of classes are allocated dynamically, and instead of naming and accessing them with fixed identifiers at compilation time, you can name them and access them with strings at execution time. The local variable `new_bike` resembles a C++ pointer in the sense that you can't use it to access an instance of the class `Bicycle` until you initialize it to point to one. In this example, `CREATE_OBJECT` initializes `new_bike` to point to a newly created instance; alternatively, a call to the `GET_INSTANCE_BY_NAME` function could determine which instance to point to.



Note: You should normally not use `CREATE_OBJECT` as in this example. Use the example shown in *Using Instance Names* for a better way.

The second action, `spl_main`, holds a few more surprises for C++ programmers. First, the action `create_bicycle` creates two bicycles (the `execute` statements), one of them using the subclass `Atb`. The code doesn't keep a list of the instances of `Bicycle` that were created. Instead, when it comes time to test the program with `print` statements, `GET_DESCENDANTS` provides a list of instances. The code uses a `foreach` statement to iterate over the list. Notice that you aren't limited to printing out the data that lies in the slots of an instance, but can also determine the class an instance belongs to, and can use the identity of that class as an extra piece of information about the instance. Instead of using a string or enumeration slot to remember that one of the bicycles is an all-terrain bicycle (`ATB`), you can rely on the ability of `PepperCode` to query its class membership directly.

Users of languages like `LISP` or `SmallTalk` may not find these aspects particularly novel, except for the `succeed` statement in each action that hints at the mechanism for committing or rolling back changes that the action has made to the database.

The following sections cover `PepperCode` in greater detail. The simple example in this section doesn't show all the constructs of the language, but it does provide an overview of the material that the remaining sections will cover.

CHAPTER 3

Understanding PepperCode Basics

This section gives basic information about the PepperCode language.

Writing .spl Files

PepperCode code should be in files that end in .spl. By convention, the following types of files are used for PepperCode code; they should be placed in this order in a makefile to ensure that action declarations are compiled before the action definitions, which depend on those declarations:

- `cpp_filename.spl`: C/C++ function declarations
- *filename*.spl: class definitions
- `dcl_filename.spl`: action declarations
- *filename*.spl: action definitions
- `transaction_filename.spl`: transaction definitions
- *filename*_ui_actions.spl: PC client form actions

In this guide, the different elements of the PepperCode language are discussed without specifically referring to this file architecture. You should keep in mind that when you write code, you need to place it in these different files.

Writing PepperCode #include Statements

Pre-compiled header files represent the interface to classes, actions, and enums in a particular module. This representation is in a very compact form that is quicker and easier for the compiler to parse.

The following is the syntax of a #include statement:

```
#include "filename"

#include <filename>
```

These statements are equivalent. They behave as if the compiler reads in declarations for the symbols exported by the named file, without actually compiling that file.

PepperCode has two compiler directives, called `#include` and `#remote_include`, that allow you to import declarations and actions from other source files.

The compiler opens the source file and inserts its contents where the statement appears. As with C/C++ code, filenames in double quotes (" ") should be in the current directory or in the path specified with the `-I` command line option, and files enclosed in angle brackets (< >) should be in a location specified by a search path in the environment.



When placing filenames in double quotes (" ") for Windows NT, you can use a file delimiter of "/" instead of "\".

If an included.spl file includes other files, the directory containing the “outer” included.spl file is used in the search list for the nested files instead of the current working directory.

Like C/C++, if the PepperCode compiler cannot find the files in the current working directory, it will search the directories specified with the `-I` command-line option of the PepperCode compiler.

PepperCode includes are different from C/C++ includes in the following ways:

- No white space may precede the pound sign (#).
- The file must end in .spl.
- You cannot start a syntactic construct inside an included file and end it in the outer file, or vice-versa.

Rules for Inclusion and Writing #include Statements

If you use an action or class in a *.spl file, and the action or class is defined in another *.spl file, you must `#include` the other *.spl file.

The following rules apply to `#include` statements:

- If <filename> begins with "/" (or, on Windows NT systems, with "\" or a single character followed by a colon), it is an absolute name; otherwise, it is a relative name.

For example, an absolute path would be `"/home/jfarris"` on UNIX and `"\\Sanma-file-01\c:"` on Windows NT. A relative path would be `"/jfarris"` on UNIX and `"c:\autoexec.bat"` on Windows NT.

- If <filename> is relative, then the compiler searches for the file using a list of directories. The first directory in the list is the directory in which the compiler found the file containing the `#include` statement. The remaining directories are provided by the `--include` and `-I` options specified on the compiler command line, in order. Note that the current working directory does not explicitly appear in the list unless you specify it with `--include` or `-I`, or the file containing the `#include` statement was found in the current working directory.

Example:

Suppose you're compiling xyz.spl, a file in "/home/jfarris/spl", and you're including abc.spl. When you compile xyz.spl, the compiler will look for abc.pchs because the compiler uses pre-compiled header files (*.pchs) when including *.spl files. If you're using the default compiler option as follows:

```
> spl xyz.spl
```

the compiler will have no problem finding abc.pchs if it is in the same directory as xyz.spl, "/home/jfarris/spl", because it looks in the directory containing the file you are compiling, by default. However, if abc.pchs is in another directory (say "/home/jfarris"), the compiler will not be able to find abc.pchs, and you will get a file-not-found error. If you use the --include compiler option to include "/home/jfarris" in the include path as follows:

```
> spl --include "/home/jfarris" xyz.spl //absolute pathname
```

or

```
> spl --include ".." xyz.spl //relative pathname
```

the compiler will be able to find abc.pchs, and in the absense of any other problems, will compile xyz.spl and the included code.

An alternative to using the --include compiler option (following this example) is to place a path to abc.pchs in the include statement as follows:

```
#include "../spl_parent.spl" //relative pathname
```

or

```
#include "/home/jfarris/spl_parent.spl" //absolute pathname
```

The name under which the compiler finds the file is called the "full name". A full name may be absolute or relative. For example, the "full name" of spl_parent.spl in the above example is "/home/jfarris/spl_parent.spl".

The name of just the file without the path is called the "base name". For example, the "base name" of spl_parent.spl is "spl_parent.spl".

To break cycles in the graph of file inclusions, the compiler refuses to read two files having the same "base name". Thus, if two different files in two different directories have the same name, a compilation will include only one of them, directly or indirectly.

Example:

Suppose you had the following #include statements in xyz.spl:

```
#include "../spl_parent.spl"

#include "../doc/spl_parent.spl"
```

In this case, the compiler will read one #include statement and ignore the other. A further complication is that there is no absolute rule for determining which #include statement the compiler chooses. Of course, this aspect of the compiler will only cause a problem if there are two different *.spl files with the same name in your #include statements.

The "include" statement is implemented by reading in "precompiled header" files rather than by reading the actual PepperCode and parsing it, although in concept either implementation would have the same effect. If a PepperCode source file is named x.spl, the corresponding precompiled header file is named x.pchs. PepperCode precompiled headers differ from those used by Borland and Microsoft C++ compilers: the file x.pchs is independent of changes in any source file except x.spl.

A PepperCode compilation normally generates a precompiled header file unless you use the "--no_header" option on the command line.

Precompiled header files are architecture-independent.

Using two files that include each other

Normally you must compile any included *.spl file before you compile the file containing the "#include" statement. If two files include each other, directly or indirectly, you must use the command line option "--header_only" to generate the precompiled header for the first one; then you can compile the second, and after that you'll be able to compile the first.



For more information and an example that shows what to do when two source files include each other, see the --header_only compiler option.

Using #include instead of forward declarations

If you use an action or class in a *.spl file, and the action or class is defined in another *.spl file, you must #include the other *.spl file. The only exception to this is the use of a forward action declaration in lieu of a #include statement. This is not recommended but is allowed to maintain backward compatibility.



For more information on the use of forward action declarations, see Declaring Actions: Forward (or Incomplete) Action Declarations.

Action Example:

Using the example from Declaring Actions: Forward (or Incomplete) Action Declarations, #include columns.spl is used instead of using a forward action declaration:

user.spl

```
#include "columns.spl"

action spl_main(input: float supply, input: float demand,
    local: float difference)
{
```

```

    execute print_three_columns(:a supply, :b demand, :c (demand - supply));
}

```

columns.spl:

```

action print_three_columns(input: float a, input: float b, input: float c)
{
    PRINTF("%15.5e\t%15.5e\t%15.5e", a, b, c);
}

```

Classes Example:

In the following example, you must `#include` the *.spl file containing the definition for `Parent_Class`. You cannot use a forward class declaration in a situation like this because you are referring to slots on the class. Also, please note that you do not need a forward class declaration for `Firstborn_Class`. Starting in Release 8.0, it is unnecessary to have forward class or action declarations for classes or actions that are defined in the same *.spl file.

If `Parent_Class` were defined in `spl_parent.spl`, the code would look like this:

```

#include "spl_parent.spl"

action spl_main()
{
    Firstborn_Class.person_name = "Cain";

    Parent_Class.person_name = "Adam";

    PRINTF("%s is the parent of %s.\n",
    Parent_Class.person_name, Firstborn_Class.person_name);
}

class Firstborn_Class : Parent_Class {
};

```

The file `spl_parent.spl` could be as simple as this:

```

class Parent_Class {string person_name};

```

#include and pre-8.0 versions

Included files are processed much differently by the Release 8.0 Compiler than they were in previous versions. *.h files no longer need to be included because the libraries corresponding to

*.h files are now included in the substrate. Including *.h files will cause no compiler problems, but it will generate a warning.



For more information on the warning, see No longer necessary to include C++ files ending in .h.

In Release 7.5, you had to include *.h files to describe human-written C++ functions. In both Releases 7.5 and 8.0, you have to include *.spl files to describe PepperCode (SPL) functions; however, in Release 8.0 the apparent inclusion of a *.spl file causes the compiler to actually use the corresponding *.pch file. Inclusion of *.spl and *.pch files is unrelated to the use of .h files in previous versions of PepperCode.

These changes are largely the result of the 8.0 Compiler's new compilation method. Starting in Release 8.0, the compiler makes two passes over the source code. In the first pass, the compiler processes inclusions using pre-compiled header (*.pch) files. Then, in the second pass, the compiler compiles the source code to object code.

Understanding Scopes and Identifiers

There are four different kinds of scope:

- **Predefined scope:** This includes all identifiers which are intrinsic to the language, such as "Base_Class" or "ADD"
- **Global scope:** This includes all identifiers declared at the outer level, such as action declarations, enum declarations, and class declarations. It is nested within the predefined scope.
- **Local scope:** each action creates a local scope. All parameters defined in an action belong to its local scope. As such, they can only be used in that action or actions that are children of that action.
- **Foreach scope:** Each "foreach" statement creates a nested local scope and declares its index variable within that scope.

An identifier declared within a nested scope may duplicate one which is declared in an outer scope, in which case it hides the outer declaration for the duration of the nested scope.

Within a scope, an identifier cannot be associated with two different definitions, except in the case of enumeration constants, as described below.

The program may not refer to an identifier that is not associated with a definition which is currently in scope, except in the case of slot default values, as described below.

Scope Example:

In the following example, xyz has local scope (3) and foreach scope (4) in action abc. The action variable abc.o has global scope (2). PRINTF has pre-defined scope (1).

```
action abc(input: int i,
```

```

output: int o,

local: int xyz = 17,

local: oset[string] os)
{
    PRINTF("Outside the loop xyz is an integer: %d", xyz);

    foreach xyz in os
    {
        PRINTF("Inside the loop xyz is a string: %s", xyz);
    }

    PRINTF("Now xyz is an integer again: %d", xyz);
}

```

Writing PepperCode Comments

You can format your comments in C or C++ styles. Following is an example of C-style comments:

```

/* This is
   a multiple line
   comment. */

a = /* A partial line comment */ 5;

```

Here is an example of C++-style comments:

```

// This is
// a multiple line
// comment.

a = 5; // Cannot be embedded

```

Writing PepperCode Documentation Comments

You can use `#document` and `#end_document` to form comments. These comments are a PepperCode documentation feature. The compiler ignores the text between `#document` and `#end_document`, so these comments do not have to be C or C++ style comments. It then writes these comments to a `*.doc` file using the `--doc` compiler option. Each `*.spl` file containing

comments formed with `#document` and `#end_document` gets a corresponding `.doc` file when compiled.

Here is an example of `#document` and `#end_document`.

```
#document transaction_create_or_set_chip_env

    You put your comments here.

    These comments do not have to be enclosed in /* and */.

    These comments do not have to begin with //.

    These comments can continue for as many lines as you like.

#end_document transaction_create_or_set_chip_env
```

Notice that `#document` is followed by the name of the transaction (in this case, `transaction_create_or_set_chip_env`). When you include the name of the transaction, the compiler can associate these comments with that transaction, ensuring that the `*.doc` file tells you which transaction these comments apply to.

Understanding `#document` error messages

You must include the transaction name in the `#document` and `#end_document` statements. If you don't, you will get the error "Missing transaction name after `"#document"`." If you include the transaction name with `#document` but not with `#end_document`, you will get a mismatch error.



For more information on these errors, see Missing transaction name after `"%s"..` and Mismatch between `"#document %s"` at line `%d` and `"#end_document %s"...`

If you forget the `#end_document` statement, you may not get an error at all, but the compiler will consider the remainder of your `*.spl` file to be a comment. This could cause logic problems that may be difficult to diagnose. Conversely if you forget the `#document` statement, the compiler will consider your comment text to be code. This could also cause logic problems that may be difficult to diagnose. Here is the error received when the `#document` statement was commented out:

```
hello_world.spl:8: parse error

spl_main is a pre-defined action that is
^

    (Skipping to ';' at 16:28)

hello_world.spl:17: parse error

    }
    ^
```


If you use the --doc compiler option, the compiler writes the text between #document and #end_document to a file named <filename>.doc for all of the #document blocks in the *.spl file, where <filename>.spl is the name of your source file. This file will contain the name of the source file and the line number at which the documentation comments start, so you do not have to put this into the comments.

Example: Hello World!

hello_world.spl:

```
#document spl_main

spl_main is a pre-defined action that is

executed automatically when the program is run.

It does not have to be called.

#end_document spl_main


action spl_main()

{

    PRINTF("Hello, world!\n");

}
```

hello_world.doc:

```
#document spl_main, "hello_world.spl", 6:

spl_main is a pre-defined action that is

executed automatically when the program is run.

It does not have to be called.

#end_document spl_main
```

Format for #document comments

When you write #document comments for a particular transaction, you should place them just before the code for that transaction. Use the following format:

Description: Describe the purpose of the transaction here.

Inputs: List the input parameters. For each parameter, list its name and data type, and describe the parameter.

Required Inputs: List the names of the required inputs here.

Input Defaults: List the names and default values for each input parameter that has a default value.

Outputs: List the output parameters. For each parameter, list its name and data type, and describe the parameter.

Instances Updated: List object instances in the server that are updated or whose slot values are updated.

Here is an example.

```
#document transaction_add_mfg_attribute
```

```
Description:  Adds an attribute.
```

```
Inputs:  mfg_attribute_name      STRING
```

```
        The mfg attribute identifier.
```

```
        class_name                STRING
```

```
        The class name of the attribute instance required.  Must be a subclass of
Mfg_Attribute.
```

```
Required Inputs:  mfg_attribute_name
```

```
Input Defaults:  class_name "Mfg_Attribute"
```

```
Outputs:  exit_msg  STRING
```

```
        An exit message to be read by the user.
```

```
Instances updated:  Mfg_Attribute instance added.
```

```
#end_document transaction_add_mfg_attribute
```

Typically, #document documentation blocks like those in the examples are used only for transactions. However, they can be used to document anything, and anything documented with #document blocks will be written to the *.doc files when you use the --doc compiler option.

Using --doc and --header with documentation comments

Starting in Release 8.0, *.doc files are generated from #document blocks using the --doc compiler option instead of the -d option. You can also generate them in standalone mode without actually compiling the source code containing the #document blocks.

To generate only a *.doc file for a particular *.spl file, use the --header_only and --no_header compiler options in conjunction with this compiler option.

Example:

To generate the file xyz.doc without actually compiling xyz.spl:

```
> spl --no_header --header_only --doc xyz.spl
```

To generate the file xyz.doc along with compiling xyz.spl (to an object file):

```
> spl --doc xyz.spl
```

Writing PepperCode Notice Comments

You can use #notice and #end_notice to enclose comments. Use these comments to form a copyright notice. If you don't include the #notice and #end_notice statements, the compiler will give a warning message.

Here is an example of #notice and #end_notice.

```
#notice

Copyright 1994-1998 by Peoplesoft, Inc.

All U.S. and World rights reserved.

#end_notice
```

If you do not include a #notice statement block, you will receive a warning message.



For more information about the warning message received, see Source file should have a "#notice" statement..

Understanding PepperCode Data Types

PepperCode has the data types shown in the following table. They are used in slot or action parameter definitions.

PepperCode Data Types

Data type	Syntax	C/C++ comparison	Description
int	int name	Corresponds to C long In C++, would be passed by value	Stores integer values. Signed two's complement integer
float	float name	Corresponds to C double In C++, would be passed by value	Stores float values. Doubleprecision IEEE floating point
time	time name	No C equivalent In C++, would be passed by value	Represents a relative time value, in seconds, that could be added to or subtracted from the date type. It can be a positive or negative value.
date	date name	No C equivalent In C++, would be passed by value	Stores an absolute point in time. You cannot add date values together, but you can add a date and time value to get a new date; for example, value = ADD(time, date);.
string	string name	Equivalent to C char * In C++, would be passed by value	Stores a sequence of characters (null-terminated) that are bounded by double quotes (" "). Usually used for text. Null-terminated array of bytes encoded in the UTF-8 version of Unicode.
enum	declaration: enum enum_name {values}; reference: enum<enum_name> name	Similar to C enum, but PepperCode enums can be compared to enums only, while C enums can be compared to integers, too.	* Stores a logical grouping of named constants, for example, enum Boolean { TRUE, FALSE }; It can be used to specify the only valid values for a slot.
instance	instance<class_name> name	Corresponds to C++ object of type class In C++, would be passed by reference or copy into	** Pointer to an instance of a PepperCode class (object instance). Gives quick access to the slots of an object and is generally used to link objects together.

class	class<class_name> name	Corresponds to C++ type class In C++, would be passed by reference	** Pointer to a PepperCode class (object class). Most objects don't need to point to a class, since they can query for their direct parent class.
oset	oset[data_type] name	Like a linked list	Stores an ordered set (list) of a specified data type—for example, oset[int], oset[instance<Base_Type>], and so on. It can reference any data type besides oset. An alternative to an oset of osets is an oset of instances that point to osets. Note that the list is ordered, but not sorted.
action	For action parameters: action<action_name> name For action parameters and slots of a class: action<schema_name> name	Similar to C functions and Pascal procedures, but provide more than the procedural abstraction of functions and procedures	Points to a PepperCode action, which is an encapsulation of PepperCode statements with input and output to the statements. Or points to an action schema that actions can be based on. To implement a method in a class, you must specify a schema in the slot.
history	history<data_type> name	No equivalent in any language	Stores time-varying data of type history<int>, history<float>, or history<string>. The values of histories cannot be changed by PepperCode statements directly; instead, they are changed by a side effect mechanism.

* Starting in Release 8.0, two enumerations may use the same constant name(s).

Example:

```
enum Week { SUN, MON, TUES, WED, THU, FRI, SAT };

enum WorkWeek { MON, TUES, WED, THU, FRI };
```

** The term "pointer to" in the previous descriptions means that when you declare a variable of type "instance" (for example) you do not create a PepperCode instance; you merely create a variable capable of referring to a PepperCode instance which exists independently.

Example of creating a PepperCode instance:

```

action create_bicycle

    (input: int serial_number = required:,

     input: string model_name,

     input: string class_name,

// The following line creates a variable capable of referring
// to a PepperCode instance

     output: instance<Bicycle> new_bike,

     no_context:)

{
// Create an object(instance) with the CREATE_OBJECT function.
// model_name is the name of the object.
// (All named objects must have a unique name.)
// class_name is the name of the class the object belongs to.

    new_bike = CREATE_OBJECT(model_name, class_name);

    new_bike.serial_number = serial_number;

    new_bike.model_name = model_name;

    succeed();

}

```

Understanding PepperCode Performance Considerations

The following operations are resource- and computation-intensive processes:

- Creating and deleting objects. Try not to create unnecessary objects. If a feature is implemented with fewer objects, it will be more efficient.
- List processing.



For more information about processing lists more efficiently, refer to Writing Osets.

- Several enums in a tight loop.



For more information about processing enums more efficiently, refer to Writing Enumerations in Loops.

- String compares. Strings are compared character-by-character.
-



For more information, refer to Using EQ With Strings.

- Recursion. Because of the overhead involved with calling actions, don't use recursion with PepperCode. Instead, use an iterative algorithm or write a C++ function that uses recursion.

Using PepperCode Naming Conventions

A name for a class, action, slot, and parameter is a contiguous set of alphanumeric characters. Underscores separate characters into "words," for example, `action_name`. In addition, a class name uses a capital letter at the beginning of each word, for example, `Spl_Class`. This helps you to easily distinguish between class names and action parameters; for example, `My_Class` versus `my_class`.

The C++ runtime functions should be in all capital letters.

CHAPTER 4

Understanding PepperCode Classes

PepperCode objects are either classes or instances of classes. Often, however, the term "object" is used loosely to mean "instance" and not "class".

A PepperCode class is a collection of members called "slots". Every slot has these attributes:

- name
- data type
- initial value
- an attribute called "class_slot:"
- an attribute called "side_effect:"

The programmer may declare that a class has a particular slot, or the programmer may declare that the class inherits slots from zero or more parent classes. A class also has slots that are predefined by the language itself.

A class implicitly inherits from the predefined class called "Base_Class". (Base_Class is predefined starting in Release 8.0.)

To form the set of slots belonging to a class, we first make a set of all of the slots belonging to the first parent class. Then we add all of the slots from the second parent class whose names do not duplicate those already in the set, and repeat for each additional parent class. Next we add the slots directly declared in the class. If the name of any of those duplicates that of a slot that is already in the set, then the directly declared slot is allowed to override certain characteristics of the parent class. It can change the data type, change the initial value, add the "class_slot" attribute, or add the "side_effect:" attribute (this is called "specialization"). Finally we add any slots predefined by the language whose names do not duplicate those already in the set.

Thus, if a class has the opportunity to inherit a slot "x" from more than one parent, it inherits that slot from the parent that appears first in the list, and the remaining parents have no effect on that slot.

Example:

In the example below, Homer and Marge are the "parents" of Bart and Lisa. Bart inherits hair_color from Homer because Homer is the first parent on his inheritance list. Likewise, Lisa inherits hair_color from Marge.

```
class Homer {
```

```

    string hair_color

    string favorite_beer

};

class Marge {

    string hair_color

    string favorite_book

};

// Bart inherits hair_color from Homer

class Bart: Homer Marge {

};

// Lisa inherits hair_color from Marge

class Lisa: Marge Homer {

};

```

Every slot on a class is capable of storing a value. If the programmer specifies a value with an "default:" clause, the slot is born with that explicit default value; otherwise, the slot is born with an implicit default value. During execution, the program may assign a new value to a slot belonging to the class.

The "class_slot:" attribute has no effect on the class itself, but does affect instances of the class.

The "side_effect:" attribute permits the slot to participate in a side effect function.

The programmer may "specialize" a slot which was inherited from a parent class by changing the default value, or by adding the "class_slot:" attribute, or by adding the "side_effect:" attribute. Such a slot inherits from the parent class those characteristics not specified by the programmer. The programmer may not "specialize" the data type, because that creates an independent slot which does not inherit any characteristics from any parent class.

You should use the side_effect slot only when it is needed. The only reason for not using it is that it wastes memory and disk space. However, severe consequences can occur if you don't use the side_effect slot when it is needed.

A class is a "first-class" object. The program can manipulate data on the slots of a class, and it can manipulate variables of type class. A variable of type class is capable of referring to the class itself or to any subclass of that class.

At runtime, the program can create a new class which inherits from one or more parent classes (every class will inherit from Base_Class whether that is specified explicitly or not). However, it cannot specialize a slot (although it can change the default value of a slot after the new class has been created), nor can it create slots other than those inherited from the parents.

At runtime, the program can refer to the value of a slot on a class, and it can redefine the value of that slot.

Classes can inherit redefined values. If a class inherited the value of a slot from a parent when it was born, and the program has not subsequently redefined the slot on the child, then a change at runtime to the value of the slot on the parent will propagate to the corresponding slot on the child. In other words, the value of an inherited slot on a class is coupled to that of the slot on the parent until it is "disconnected", either by specifying an explicit default value for the child slot at its birth, or by assigning a value to the slot at runtime. The explicit default or the assignment will "disconnect" the slot even if the redefined value is not different from the parent's value.

An instance of a class is an replica of the class which contains exactly the same set of slots as does the class.

If a slot has the "class_slot:" attribute, then the instance does not have an independent copy of the slot; instead, the copy belonging to the class appears as if it also belonged to the instance. Changing the value of that slot on the class will change the slot on the instance too, and vice versa.

If a slot does not have the "class_slot:" attribute, then the instance has an independent copy whose initial value is copied from the corresponding slot on the class. (For this reason, a slot on a class which does not have the "class_slot:" attribute is often called a "default value slot", because it provides the initial value for the corresponding slot on the instance.)

Subsequent changes to the slot belonging to the class have no effect on the corresponding slot belonging to the instance, and vice versa. In particular, once an instance is created, it is not affected by the inheritance of redefined values described in connection with slots on classes.

The PepperCode object model is very different from the C++ object model:

- Classes are first-class objects. A C++ programmer may think of a class as incorporating a special "hidden" instance which contains a copy of all of the slots defined on the class, whereas true instances contain only the slots which do not have the "class_slot:" attribute.
- All inheritance in PepperCode is "virtual" in C++ terms. In the case of multiple inheritance, a class contains only one copy of a particular slot, even if it could have inherited that slot from multiple parents.
- PepperCode classes and instances do not have function members. A slot of type "action" behaves like a C++ slot of type "pointer to function" in the sense that the program can redefine it at runtime; it behaves like a C++ "virtual function" in the sense that even when the object is masquerading as a member of one of its parent classes, the slot will have the action appropriate to the true, runtime type of the object.
- PepperCode classes and instances exist in a global, dynamically-allocated, environment. There is no static or automatic allocation for PepperCode objects, and no scoping. There are automatic, scoped variables capable of pointing to PepperCode objects, however.

Writing New Class Definitions

Following is the syntax for a new class definition:

```

class new_class_name : parent_class_name1 [parent_class_name2 ...] {
    data_type slot_name_n { [class_slot:] [side_effect:] default: value }
    ...
};

```

Any *class_name* or *slot_name* is a contiguous set of alphanumeric characters. A *class_name* should have underscores and capital letters that separate characters into “words,” for example, *Spl_Class*. A *slot_name* can use underscores to create words. This enables you to easily distinguish between class names and slot names.

One or more *parent_class_name(s)* can be placed after a colon (:) and are delimited by a blank space, for example, *: Base_Class Mid_Class High_Class*. In a new class definition, at least one parent class must be specified. There must be a space after the colon (:).

Braces ({}), followed by a semicolon (;) contain the slots. If no slots are specified, the braces are empty; for example:

```

class Their_Class : Base_Class { };

```

In Release 8.0 and later, class definitions can set default values for slots without a separate “slot” statement.

Example:

The default value for *i* is assigned a value of 123 in the following class definition:

```

class c {
    int i { class_slot: side_effects: default: 123 }
};

```

In PepperCode versions earlier than 8.0, default slot values must be placed in slot statements. The value is assigned using the *default:* keyword. The object system provides default values at object creation time, so there are no uninitialized slots.

Old syntax:

```

class new_class_name : parent_class_name1 [parent_class_name2 ...] {
    data_type slot_name_n
    ...
};

slot new_class_name.slot_name_n { default: value [class_slot:] [side_effect:] };
...

```

Note that a class statement and all slot statements for that class must reside in the same file.

Understanding Default Values

If you omit slot statements, the compiler provides the following defaults for you:

- integer – 0 (zero)
- string – “default string”
- oset – empty list
- action – nil (attempting to execute this causes a runtime error)
- float – 0.0
- instance – Null_Instance
- class – Null_Class

You need to decide if these defaults are appropriate in your code. If not, you should provide a default. For more information on Null_Instance and Null_Class, refer to a following section, “Some Predefined Classes.”

Understanding Multiple Inheritance

When multiple parent classes are specified (called multiple inheritance), classes listed first after the colon take priority when the new class inherits values and defaults. In other words, for : Base_Class Mid_Class High_Class, if Base_Class and High_Class have specified a data type and default value for the same slot name, the Base_Class type and value would be used.

Specializing Slots

If a slot of type class A is defined in the parent class, you can specialize it in the subclass with a different type B, provided that B is a subclass of A. For example:

```
class A {};  
  
class B : A {};  
  
class C {  
    instance<A> foo  
};  
  
class D : C {  
    instance<B> foo // specialize slot definition  
};
```

You cannot specialize a slot of a primitive type, such as int or float, in a subclass declaration.

Understanding Dot Notation

In slot statements, the slots are referred to as:

```
class_name.slot_name
```

This format is called dot notation. `class_name` is the name of the class and `slot_name` is the name of the slot as specified in the slot definition.

For example:

```
class c {  
  
    int my_slot  
  
};  
  
slot c.my_slot { default: 155 };
```



For more information about dot notation, refer to Using Dot Notation in Expressions.

Declaring Classes

The following syntax is for class definitions. Only one definition may exist for a particular class identifier. If a list of base classes appears, then this is a derived class, and a definition for each of the base classes must precede this definition.

```
class <class identifier> { <class body> }  
  
class <class identifier> : <base classes> { <class body> }
```

The class body consists of a list of slot declarations.

The list of derived-class slots is formed as described at the start of this chapter.

```
class <class identifier> ;
```

This is a forward (or incomplete) class declaration. It is compatible with any other class declaration having the same identifier.

Forward Class Declarations

Forward class declarations are not recommended in Release 8.0, but they are still allowed to maintain backward compatibility with previous versions.

Forward class declarations can be used only to inform the compiler of the existence of a class that is defined elsewhere and only in the manner specified in the first example below. Of course, the

compiler must have access to the class definition. If they are used for any purpose beyond this limited use, an error will occur.

The following example illustrates how forward class declarations can be used:

```
class Second;

class First {
    instance<Second> firstslot
};
```

The following example illustrates how forward class declarations cannot be used:

```
class Second;

action spl_main (local: string c_name)
{
    c_name = Second.name;
}
```

In this example, execution of the code in this second example causes the following error:

The declaration of "Second" is incomplete.



For more information on this type of error, see The declaration of "%s" is incomplete..

Slot Clause List Statements

The following syntax is for slot clause list statements.

```
slot <classname> . <slotname> { <slot clauses> } ;
```

A slot clause list statement provides a slot clause list to add attributes to the normal declaration of a slot which appeared within an earlier class statement. This quietly overrides the corresponding attributes in the normal declaration. The complete declaration of the class must appear in the same source file as the slot clause list statement. A default value specified in a slot clause list statement must be compatible in data type with the normal declaration.

This statement is a convenience for use in situations where a derived class inherits a slot from a base class but wishes to change only one attribute, such as the default value; it eliminates the need to repeat the entire slot declaration within the derived class.

Example:

```

class base {

    int i { class_slot: default: 17; }

};

class derived: base {

};

slot derived.i { default: 18; }

```

If the default value is a string or an undefined identifier, and the data type of the slot is "action", "class", or "instance", then the identifier is assumed to name an action, class, or instance, which need not be declared in the current compilation. In this circumstance, a class or instance need not exist until runtime, but it must exist before the first reference to the slot or the first instantiation of the class containing the slot; otherwise, a runtime error will occur. In this circumstance, the undeclared action must actually exist in some other compilation which will be linked into the same program.

Slot Declaration Statements

A slot declaration consists of a data type and the slot name, optionally followed by a slot clause list. For example:

```

int slotname
float another_slotname { default: 3.5 class_slot: side_effect: }

```

If a slot declaration does not provide a slot clause list, then a subsequent slot clause list statement may provide one, but there must not be more than one slot clause list for a particular slot in a particular class.

If the program does not specify an explicit default value for a slot via inheritance, or via a slot clause list within the declaration, or via a separate slot clause list statement, then the data type determines the implicit default value according to the following list. (Note that the implicit default values for slots are different than the implicit default values for action parameters. Implicit default values for action parameters are listed in Understanding Parameter Defaults.)

integer	0
string	"default string"
instance	Null_Instance
class	Null_Class
action<schema_name>	nil pointer
action<action_name>	nil pointer

float	0.0
oset	empty list
enum<>	nil pointer

Understanding Instance and Class Slots

The value for a slot is by default stored on its object instance and is called an instance slot. You could also store the value of a slot on the object class; this is called a class slot. For an instance slot, a copy of it is created each time an object instance is created. For a class slot, all object instances use the same slot and value; it can be used for communication between objects. A PepperCode class slot is like a C++ static data member or static member function—all instances of the class read and write the same value for that slot.

To create a class slot, put the `class_slot:` keyword in a slot statement:

```
slot class_name.slot_name { class_slot: };
```

For example:

```
class Spl_Class : Base_Class {
    int          value_1
    float        value_2
    string description
};

slot Spl_Class.value_1 { default: 10 class_slot: };
slot Spl_Class.value_2 { default: 99.345 class_slot: };
slot Spl_Class.description { default: "default" class_slot: };
```

Writing Temporary Objects

Typically, applications written in PepperCode provide the capability of saving objects to a disk file called a snapshot. It's useful to be able to tag certain objects so that the application knows not to save them. Placing the keyword `temporary_instances:` at the end of the slot definition list accomplishes this. For example:

```
class Temp_Element : Spl_Class {
    int    count
    temporary_instances: // This is a temporary object
};
```

Using Predefined Classes

One of the reasons that PepperCode 8.0 is leaner and faster than the PepperCode of previous releases is because more of its constructs are predefined. As such, they are in the predefined scope. The following objects are declared by the compiler to be in the predefined scope:

```
class Base_Class {
    string name;

    string class_name { class_slot: }

    string class_display_name { class_slot: }

};

class Null_Class: Base_Class {};

instance<Null_Class> Null_Instance;

class Base_Enum_Class: Base_Class {
    string enumerator;

    string enum_display_name;

};
```

As mentioned earlier, `Base_Class` is the PepperCode root class that has functionality needed by the object system; for example, it provides utilities for communication to and from the Planning graphical user interface (GUI). All objects must inherit, either directly or indirectly, from `Base_Class`.

Every class and instance has a predefined readonly slot of type string called "name" which gives the name of the class or instance.

The slot "class_name" gives the name of the class or, if used on an instance, the name of the class to which the instance belongs. It is a readonly slot.



For more information in an example of using the `class_name` slot, see `GET_NAME_OF_CLASS`.

The slot "class_display_name" is a readonly slot used for internationalization.

When you refer to this slot, the runtime system looks in the local language message table for the string which corresponds to the name of the class. If found, it returns that string; otherwise, it returns the name of the class. This slot is not visible from any instance of the class.

A slot "enum_display_name" of type string appears on each instance created by an "enum" statement; it translates the value of the "enumerator" slot, which is the name of an enumeration constant, into a local language. Otherwise, it has the same behavior and restrictions as "class_display_name".

C and C++ programmers tend to assume that NULL, the integer constant 0, and the value zero are all interchangeable with the concept of a null pointer. (Actually, however, a C or C++ implementation need not represent a null pointer with the value zero.) In PepperCode, the null instance isn't represented by zero: it is a valid instance whose name is `Null_Instance`. The PepperCode runtime function `GET_NULL_INSTANCE` returns this instance. Because the compiler permits you to set an instance either to the null instance or to zero, it's important to distinguish between the two. For example, the following action will always fail:

```

action a (local: instance<c> ci)
{
    ci = GET_NULL_INSTANCE();

    if (EQ(ci, 0))
        succeed();

    ci = 0;

    if (EQ(ci, GET_NULL_INSTANCE()))
        succeed();

    fail();
}

```



For more information and a description of `GET_NULL_INSTANCE`, refer to [Accessing C/C++ Functions](#).

Similarly, the null class is a specific class whose name is `Null_Class`, which isn't the same as a class whose value is 0.

Starting in Release 8.0, you are allowed to use the name of a class in an expression. However, we have maintained backward compatibility in that you can still use the `GET_CLASS` C++ Functions to return the name of a class. Provided the compiler has seen a definition of a particular class, you can use that class in an expression instead of calling `GET_CLASS_BY_NAME`:

```

class some_class { int some_slot };

action spl_main ()
{
    some_class.some_slot = 15;
}

```



Note: Assignment statements that use the name of a class in an expression can only be made in an action.

Using Instance Names

All instances of a class have an intrinsic single name slot that is automatically initialized to provide the name of the instance. Intrinsic, in this case, means that you don't have to declare a name slot on the class in order to get this slot. This intrinsic name slot is read-only so you cannot change it. Each instance of a class should have a unique name; if you try to create a new class or object with the same name as an existing class or object, the object system generates a unique name for the object. The following example illustrates how objects are named:

```
class Named_Class { };

action named_object_test

    (local: instance<Named_Class> named_object,

    no_context:)

{

    execute create_object(:object_name "animal",

                        :class_name "Named_Class");

    named_object = create_object.new_object;

    PRINTF("\n%s\n", named_object.name);

    execute create_object(:object_name "vegetable",

                        :class_name "Named_Class");

    named_object = create_object.new_object;

    PRINTF("\n%s\n", named_object.name);

    execute create_object(:object_name "mineral",

                        :class_name "Named_Class");

    named_object = create_object.new_object;

    PRINTF("\n%s\n", named_object.name);

    succeed();

}
```

When you run this, it prints:

```
Enter an action call: B()

animal

vegetable

mineral

Result: ( :RESULT 3 )
```

As was mentioned earlier, slot name is read-only. The following code will fail:

```
execute create_object( :object_name "animal",
                      :class_name "Named_Class");

named_object = create_object.new_object;

named_object.name = "cat"; // This line will cause failure.

PRINTF("\n%s\n", named_object.name);
```


CHAPTER 5

Understanding PepperCode Actions

An action is similar to a C function or Pascal procedure, but with very different semantics for memory allocation and the lifetimes of variables and changes to variables, which is described later. Following is an example of an action definition:

```
/*          action      */

action print_simple_string

/*          action parameters      */

(input: string pstring = "Null",

 local: int string_length = 0,

 output: int printed)

/*          action body      */

{

    printed = 0;

    string_length = STRLEN(pstring);

    if (string_length < 2) {

        PRINTF("\nstring < 2");

        fail;

    }

    else {

        PRINTF("\n%s", pstring);

        printed = 1;

    }

}
```

```

        succeed();
    }
}

```

An action has a unique name that can be referenced from classes and other actions. It also has a list of local, input, and output parameters and an action body containing PepperCode statements.

To execute an action from within another action, use an `execute` statement.

Writing Action Definitions

A PepperCode action is analogous to a C function or Fortran subroutine. It provides formal parameters, local variables, and a series of executable statements.

Following is the syntax for a new action definition:

```

action <schema_name> action_name

    (param_type: data_type param_name_1 = value_1,
      ...
      param_type: data_type param_name_n = value_n,
      context: | no_context:)
{
    ...
    succeed(); | fail(); | leave();
}

```

Any *action_name*, *schema_name*, or *param_name* is a contiguous set of alphanumeric characters; the name can have underscores that separate characters into “words,” for example, `spl_action`.

schema_name is optional. If it is specified, put the name in angle brackets (<>).

Any previous declaration must match with respect to the presence or absence of a schema and (if the schema has been defined with an `action_schema` statement) with respect to the name of the schema.



For more information, refer to Writing Schemas.

The action parameter list is a group of parameter definitions enclosed in parentheses (()); if there are no parameter definitions, empty parentheses are required. However, if you have an action declaration, no parentheses appear in the action definition. A space separates the `data_type`

from the *param_name*. *param_type* always ends in a colon (:). Except for the last line in the list, a comma separates each definition from the next. Optional default values are specified with an equal sign (=); for example, `local: string string_length = 0`.



For more information, refer to Declaring Actions: Forward (or Incomplete) Action Declarations and Writing Action Parameters.

If the optional `no_context` keyword is included in the parameter list, no new context is created when the action is executed. If this keyword is omitted, a new context is created before execution, since the default is `context`.



For more information, refer to Understanding Context.

PepperCode statements are placed inside the action body, which is contained by braces (`{}`). One of the PepperCode exit statements, either `succeed()`, `fail()`, or `leave()`, must also appear. These statements exit the action and specify whether changes made to object values are accepted or rejected.



For more information, refer to Understanding Context.

Following are examples of action definitions. The first example is a simple action that prints “Hello There.”

```
action print_hello_there

    ()                // No action parameters

{

    PRINTF("\nHello There\n"); // Call C printf

    succeed();

}
```

Here is a more general implementation of an action that prints text. It accepts a string to be printed.

```
action print_simple_string

    (input: string pstring,      // Pass the string to be printed

     no_context:)               // Prevent context creation

{

    PRINTF("\n%s\n", pstring); // Call C printf

}
```

```

    succeed();
}

```

Following is a more complex example, showing an action with various types of action parameters, defaults, and exit statements.

```

action print_simple_string

    (input: string pstring = "Null",    // Pass in the string to be printed

     local: int string_length = 0,      // Local variable with a default

     output: int printed,               // Show that the string was printed

     no_context:)                      // Prevent context creation
{
    printed = 0;                       // Initialize the output parameter

    string_length = STRLEN(pstring);    // Call C strlen

    if (LT (string_length, 2)) {        // If the length is less than 2,
        PRINTF("\nstring < 2");        // print a warning

        fail();                        // and fail from the action.
    }

    else {                             // If not less than 2,
        PRINTF("\n%s", pstring);        // call C printf

        printed = 1;                   // set the output variable

        succeed();                     // and succeed from the action.
    }
}

```

Incomplete and Forward Declarations

PeopleSoft allows two exceptions with regard to the parameter list. First, for backward compatibility, this definition may omit the parameter list if a previous declaration did not omit the parameter list; we treat this as if this declaration had the same parameter list as the previous one.



For more information, and another example, see [Declaring Actions: Forward \(or Incomplete\) Action Declarations](#).

Example:

```

action<schema1> uses_schema1(input: string s = "It is also the definition.");

// This declaration works because it has a parameter list

// and the definition does not.

action<schema1> uses_schema1

{

    PRINTF("%s%s", s1, s);

};

```

Second, if the previous declaration has no local parameters, this declaration may have local parameters.

```

action <optional schema> <identifier> ;

```

This is an incomplete declaration. It declares that an action exists having the specified name, but we know nothing about its parameter list. Any previous declaration must not have a body or parameter list, and must match with respect to the presence or absence of a schema and (if the schema is present) with respect to the name of the schema.

```

action <optional schema> <identifier> <parameter list> ;

```

This is an incomplete declaration. It declares that an action exists having the specified schema and body. Any previous declaration must match with respect to the presence or absence of a schema and (if the schema is present) with respect to the name of the schema. Any previous declaration must either have an equivalent parameter list or no parameter list. Any previous declaration must not have a body.

It is unnecessary (and undesirable) to put local parameter declarations into an incomplete or "forward" action declaration. The preferred style is to put all the non-local variables in both the declaration and definition of the action, but to put local variables in the definition only. For example:

```

action a(input: int i, inout: int io, output: int o);

action a(input: int i, inout: int io, output: int o, local: l)

{

    succeed();

}

```

Avoiding Incomplete Declarations

Preferred programming practice is to avoid incomplete declarations by including "*.spl" files. In fact, if you include the *.spl file containing an action definition, you do not need to make a forward action declaration for it. Thanks to the "precompiled header" feature of the compiler, including a "*.spl" file uses far fewer system resources than in previous PepperCode versions. Because the compiler makes two passes over the source file, it allows the use of an action to precede the definition if both occur in the same file. This also applies to the use of classes.

As you probably know from C++, a pre-compiled header file contains all of the action and class header information. These inclusions are resolved and the compiler determines what actions and classes have been defined in the first pass of the compiler over the source code before compilation to object starts.

When you do use an incomplete declaration, preferred programming practice is to put all local declarations in the parameter list belonging to the definition and none in the parameter list belonging to the incomplete declaration. This allows programmers to add or delete local parameters without having to edit two different parts of the program; and without changing the modification date of the file containing the incomplete declaration (assuming the incomplete declaration and the definition are in separate files).

An incomplete declaration with local parameters will cause a warning message about obsolete coding practice; so will a definition that omits the parameter list.

Matching Parameters and Parameter Lists

Two parameter lists match if all keyword parameters match, disregarding order.

Two keyword parameters match if their keyword-parameter types, names, data types, and initializations match.

Using context:, no_context:, and readonly:

Within the action parameter list, only one of the "context:", "no_context:", and "readonly:" reserved words may appear. The compiler may choose to implement "readonly:" by creating a context and forcing it to fail at runtime, or by using the parent context but ensuring that no changes are made.

After taking the union of the set of parameters defined by the action schema and the set of parameters defined by the action's own parameter list, no two arguments may have the same name, with one exception: an action may specify either "context:", "no_context:", or "readonly:" even if one of those words appears in the action schema parameter list. In that case, the action overrides the action schema.

For example, mutually recursive actions A and B could be coded thus:

```
action a(input: int i);  
  
action b(input: int i) { ... }
```

```
action a(input: int i, local: float f) { ... }
```

Writing Action Parameters

Here is the syntax for action parameters.

```
<category> <type_decl> <id>

<category> <type_decl> <id> = <expression>

input: <type_decl> <id> = required:

inout: <type_decl> <id> = required:
```

The <category> may be "input:", "output:", "inout:", or "local:", indicating the following behavior:

Scenario	<i>input</i>	<i>output</i>	<i>inout</i>	<i>local</i>
Caller can pass actual argument in invocation?	Y	Y	Y	N
Caller can alter this using dot notation?	N	N	N	N
Caller can read this using dot notation?	N	Y	Y	N
Callee can alter this?	N	Y	Y	Y
Callee can read this?	Y	Y	Y	Y

Passing Arguments: Considering Data Type

Output Category Example:

The comments referring to "row" refer to the table above.

```
action spl_main(local: int lo)

{

    action_output.o = 6;           // ERROR (caller cannot alter this) [2nd row]

    execute action_output(:o lo); // Caller can pass arg in invocation [1st row]

    PRINTF("%d", action_output.o); // Caller can read result [3rd row]
```

```

    PRINTF("%d", lo);
}

action action_output(output: int o)
{
    PRINTF("%d", o);    // Callee can read this (will see the default) [5th row]
    o = 5;              // Callee can alter this [4th row]
}

```

Using required: Keyword as Explicit Default Value

The required keyword, new for Release 8.0, is a way of requiring that a value be assigned to certain input parameters when calling an action. If you use "required:" as the explicit default value of an action input or inout parameter, the compiler and action interpreter will force the caller to pass an explicit actual argument.

Example:

The required: keyword is assigned as the default value for quantity.

```

action counter(input: int quantity = required: )
{
    PRINTF("%d\n", quantity);
}

action spl_main()
{
    execute counter();
}

```

Executing this program causes the following error:

```
You must supply a value for "required:" parameter "quantity"
```

If you assign a value to quantity as in the following example, the error goes away:

```

action counter(input: int quantity = required: )
{
    PRINTF("%d\n", quantity);
}

action spl_main()
{
    execute counter(:quantity 15);
}

```

Understanding non-local action parameters

If a parameter has type "action", and the parameter is not "local:", then the data type must refer to an action schema rather than a specific action.

Example:

You are not allowed to do either of the following in a parameter list declaration:

```
input: action alternate_repair

output: action alternate_repair
```

In the action definition for action `one_pass_inventory_constraint_repair`, the following input parameter is declared (this type of parameter list declaration is allowed):

```
input: action<constraint_repair> alternate_repair,
```

In this declaration, "constraint_repair" is an action schema, and "alternate_repair" is not the name of an actual action but an input variable that can be used to pass an action into action `one_pass_inventory_constraint_repair`. The action passed in must be an action that has action_schema "constraint_repair" in its definition.

The action "rm_repair_explode_planned_orders_only" is passed into `one_pass_inventory_constraint_repair` using the following execution statement (This execution statement was copied from action `default_gs_dispatch`.):

```
execute one_pass_inventory_constraint_repair(

:constraint_class constraint_class,

:deconflict_env deconflict_env,

:use_alternate_repair 1,

:alternate_repair rm_repair_explode_planned_orders_only);
```

Notice that this execute statement sets input variable "alternate_repair" equal to action "rm_repair_explode_planned_orders_only." This means that if you say "execute alternate_repair" in action "one_pass_inventory_constraint_repair," you will execute action "rm_repair_explode_planned_orders_only" (This is, of course, true only for the execution statement above and only while it is in the process of execution.).

Understanding Parameter Defaults

If, when an action is invoked, the caller supplies no actual parameter corresponding to a particular formal parameter, the program attempts to use a default value. This applies to all four categories. The rules for determining the default value are these (in order):

1. If the declaration of the parameter used the optional "`= <expression>`" construct, that expression provides the default value. The `<expression>` must be a literal or identifier, and its data type must be one which could legally be assigned to the formal parameter (see

"Assignment"). In the example below, "Ford" is the default for "model_name". (It is intended that the compiler someday permit general constant expressions here.)

If the data type is "instance", the expression may be an undeclared identifier which names an instance that will be created at runtime (This would be `new_car` in the example below.); otherwise, the identifier must name a declared entity. Notice that these rules are stricter than the corresponding rules for the explicit default value specified by the slot "default:" clause.

Example of <expression> for types string and instance:

```
action create_car(input: string model_name = "Ford",
                 output: instance<Automobile> new_car) {<body>}
```

2. If the default is not specified in the declaration of the parameter, the program uses an implicit default value determined by the data type as follows.

integer	0
string	nil pointer
instance	nil pointer
class	nil pointer
action<schema_name>	nil pointer
float	0.0
oset	empty list
action<action_name>	action_name
enum<>	nil pointer

Note that the implicit default values for action parameters are different than the implicit default values for class slots.



For more information about implicit default values for class slots, see Slot Declaration Statements:

3. If the declaration uses the "= required:" construct, the compiler (or, at runtime, the action interpreter) reports an error.



For more information about the error message, see You must supply a value for "required:" parameter "%s".

It is intended that the caller should read "output:" and "inout:" parameters only after invoking the action, but such a restriction is not in general enforceable by static analysis. If the caller violates this restriction, then either it sees the default value, or a runtime error occurs.

Understanding How Parameters Behave With Execute

When you invoke an action with an "execute" statement, the parameters shall behave as if the following steps occurred in order:

1. Assign to formal input: and inout: parameters from any actual parameters provided by the caller
2. Assign default values to remaining input:, output:, and inout: parameters
3. Invoke the action being called
4. Allocate local: parameters and assign their default values
5. Execute statements in the body of the action being called
6. Deallocate local: parameters
7. Return from the action being called
8. Assign from formal output: parameters to any actual parameters provided by the caller.

Notice that if you specify an actual output parameter in an "execute" statement, and then use the `<action>.<output_parameter>` notation in an assignment statement, the assignment generated by the "execute" statement occurs first. Also notice that the "execute" statement does not assign to an actual "inout" parameter after returning from the action; this irregularity preserves compatibility with original PepperCode compiler.

If an action uses "required:" for any parameter that is not inherited from a schema, then you may not assign that action to a slot on an object. Also, you may not pass that action as an actual parameter when invoking another action, because in those cases it would be impossible to supply the required value when invoking the action via the slot or formal parameter.

In the following example, action "counter" will be evoked by `spl_main`. What happens in each of the eight steps above will be explained in the eight steps listed below the example.

```

action counter(input: int dummy,

input: int quantity = required:,

local: int i,

output: int o)
{
    PRINTF("quantity equals %d (passed in from spl_main).\n", quantity);

    PRINTF("dummy equals %d (by default).\n", dummy);

    PRINTF("local parameter i equals %d (by default).\n", i);

    PRINTF("default value of o equals %d.\n", o);

    // o, which will be passed to spl_main, is assigned a value of 5.

```

```

    o = 5;
}

action spl_main(local: int lo)
{
    PRINTF("default value of lo equals %d.\n", lo);

    // lo = 0

    execute counter(:quantity 15, :o lo);

    // lo = 5

    PRINTF("After returning from counter successfully, lo equals %d.\n", lo);
}

```

When the action "counter" is executed in spl_main (following the numbered list above):

1. Integer "quantity" is assigned a value of 15.
2. Integer "dummy" is assigned a value of zero by default (see default values list above).
3. action "counter" is invoked.
4. Space is allocated for local parameter "i", and it is assigned a default value of zero.
5. PRINTF commands in "counter" are executed, the the output parameter o is assigned a value of 5.
6. Local parameter i is deallocated.
7. action counter returns to spl_main.
8. lo, which was passed to counter (child action) as the value of counter's output parameter o, is now passed the value of 5. lo will be assigned this new value only when counter returns successfully. If counter were to fail, lo would not change to the new value. Possible failure is the reason that lo keeps its original value until counter returns.

Action Parameters are No Longer Static

In the Release 7.5 (and earlier) compiler, the first call to "non_static" in the example below would see "5" as the value of "x", but the third call would see "17" because the value "17" was "left over" from the previous call. In the Release 8.0 compiler, the first and third calls will both see "5", because that is the default value for "x" wherever you don't specify a value for x in the action invocation.

```

action non_static(input: int x = 5,

```

```

        input: string count)

    {

        PRINTF("This is the %s time non_static has been run. x = %d.\n",count,x);

    }

action spl_main()

{

    execute non_static(:count "1st");

    execute non_static(:count "2nd", :x 17);

    execute non_static(:count "3rd");

}

```

Running this program yields the following output:

```

This is the 1st time non_static has been run. x = 5.

This is the 2nd time non_static has been run. x = 17.

This is the 3rd time non_static has been run. x = 5.

```

By the way, this code will compile and run on the Release 8.0 compiler, as written.

Writing Schemas

A schema is a common structure for action parameters. It allows actions to be grouped by function or associated by type, and reduces the duplicate definition of parameters. An action can have action parameters from a schema, from its own parameter list, or both. A schema can also contain keywords that are placed in a parameter list. A schema is similar to a C typedef for a pointer to a function type—for example, `typedef int (*)(int, char);`.

A schema is also used to declare a slot of type action. This type of slot is similar to a C++ function member.

The syntax for a schema definition is the following:

```
action_schema schema_name (param_list);
```

A calling action can use either of the following parameter definitions to refer to actions that have a schema:

```

local: action<schema_name> param_name

local: action<schema_name> param_name = action_name

```

For example, if you were to define a schema for the action `print_simple_string`, shown in a previous section called “Action Definitions,” you could use the following definition:

```
action_schema print_str

    (input:                string pstring = "Null",

    output:                int printed,

    no_context:);
```

The action definition could be revised as follows:

```
action <print_str> print_simple_string

    (local: int string_length = 0)

{

    printed = 0;

    string_length = STRLEN(pstring);

    if ((string_length < 2) {

        PRINTF("\nstring < 2");

        fail();

    }

    else {

        PRINTF("\n%s", pstring);

        printed = 1;

        succeed();

    }

}
```

And the parameter definition in the action `print_simple_strings` could also be revised:



For more information about the original listing of action `print_simple_strings`, refer to Writing Action Parameters.

```
action print_simple_strings

    (local: action<print_str> print_string = print_simple_string,

    no_context:)

{
```

```

    execute print_string();

    execute print_string(:pstring "String Number 1");

    execute print_string();

    execute print_string(:pstring "A");

    execute print_string(:pstring "Null");

    execute print_string();

    succeed();

}

```



For more information about the use of schemas, refer to Writing Methods.

Action Schema Declarations and Definitions

An action schema describes a set of arguments.

```
action_schema <identifier> <parameter list> ;
```

This defines that an action schema exists have the specified parameter list. Any previous definition must match with respect to the parameter list. No two parameters may have the same name, and only one of the reserved words "context:", "no_context:", and "readonly:" may appear.

```
action_schema <identifier> ;
```

This is an incomplete declaration. Using this type of declaration in Release 8.0 PepperCode will cause a parameter list error.



For more information on the error, see Action should have a parameter list..

Declaring Actions: Forward (or Incomplete) Action Declarations

The use of forward action declarations is not recommended. Because of the compiler's new method for processing inclusions, forward action declarations are no longer necessary. The suggested method for gaining access to an action definition (so you can execute the action) is to include the *.spl file containing it. If the action definition is in the same file, just later on in the file, you don't need a forward action declaration at all, thanks to the two passes of the Release 8.0 Compiler.

The forward action declarations used in Release 7.5 are still allowed.



For more information on the Release 7.5-type forward action declarations, see [Declaring Actions: Forward \(or Incomplete\) Action Declarations](#).

Starting in Release 8.0, there are new rules for local parameter declarations and forward action declarations. They are simpler and more like C++. Parameter lists may now be placed in both the action declaration and the action definition. These parameter lists must, however, match with respect to non-local variables. It is unnecessary (and undesirable) to put local parameter declarations into an incomplete or "forward" action declaration. The preferred style is to put all the non-local variables in both the declaration and definition of the action, but to put local variables in the definition only.

For example:

```
action a(input: int i, inout: int io, output: int o);

action a(input: int i, inout: int io, output: int o, local: l)
{
    succeed();
}
```

The Release 8.0 Compiler considers these two declarations to be perfectly compatible, because the definition is allowed to have local variables that do not appear in the declaration. Keeping local variables out of the declaration helps avoid unnecessary recompilation when you change local variables. (Declaring local variables in the forward declaration is allowed but not recommended.)

Both of the following examples use this new type of forward action declaration. They are very much like the forward action declarations of previous PepperCode versions. The only difference between this type and the type used in previous versions is that the action definition and the action declaration both have parameter lists.

The compiler must see a declaration or definition of "print_three_columns" before it can be called. The compiler can "see" action print_three_columns with a forward action declaration in two different ways:

- Example 1: The *.spl file containing the action declaration is included in the file that executes the action. This is a C++ style of programming. In this example, this would be "dcl_columns.spl". Including the action declaration file is discouraged in Release 8.0 for reasons that are explained in the example.
- Example 2: The forward action declaration is placed in the file where the action is being executed, in lieu of including the *.spl file containing the action definition.

Example 1: Using a C++ style of programming, we include the file that provides the declaration of print_three_columns instead of the file that provides the definition. C++ doesn't let you include the file containing the definition here, but PepperCode does, so we could just as well have included "columns.spl", eliminating the need for "dcl_columns.spl" entirely, and eliminating the opportunity to cause an error by letting the definition and declaration get out of synch.

user.spl:

```
#include "dcl_columns.spl"

action first_user

( input: float supply,

  input: float demand,

  local: float difference)

{

  execute print_three_columns(:a supply, :b demand, :c (demand - supply));

}
```

Anybody who wants to call `print_three_columns` can include this file (which provides a declaration) instead of `columns.spl` (which provides a definition). This works only if the declaration and definition match with respect to all variables but local variables.

`dcl_columns.spl`:

```
action print_three_columns(input: float a, input: float b, input: float c);
```

By including `dcl_columns.spl` in `columns.spl`, we allow the compiler to check that the declaration and definition match (a mismatch would cause a malfunction in the code which includes `dcl_columns.spl` and which invokes `print_three_columns`). If you forget this step, there's a danger that someone will modify "`columns.spl`" without modifying "`dcl_columns.spl`", causing an error that the compiler cannot possibly detect.

`columns.spl`

```
#include "dcl_columns.spl"

action print_three_columns(input: float a, input: float b, input: float c)

{

  PRINTF("%15.5e\t%15.5e\t%15.5e", a, b, c);

}
```



Since `columns.spl` includes `dcl_columns.spl` and `dcl_columns.spl` cannot be compiled separately, `dcl_columns.spl` must be compiled with the `--header_only` option before compiling `columns.spl`.

Example 2: We place the forward action declaration in `user.spl` in lieu of including the `*.spl` file containing the action definition.

`user.spl`

```
action print_three_columns(input: float a, input: float b, input: float c);
```

```

action first_user(input: float supply, input: float demand,
    local: float difference)
{
    execute print_three_columns(:a supply, :b demand, :c (demand - supply));
}

```

columns.spl:

```

action print_three_columns(input: float a, input: float b, input: float c)
{
    PRINTF("%15.5e\t%15.5e\t%15.5e", a, b, c);
}

```

Note that in Example 2, we do not have to include `dcl_columns.spl` in `columns.spl`. Also, the forward action declaration in `user.spl` takes the place of including `columns.spl`. We are, however, limited by both of these methods in that the parameter lists for the action declaration and the action definition must match with respect to non-local variables.

The rules for forward class declarations have not changed. Forward class declarations are very different than forward action declarations.



For more information on forward class declarations, see [Forward Class Declarations](#).



For more information on the mismatched parameter error that can be caused by improper forward action declarations, see the error message `Mismatch in parameter "%s" (see %s:%d)`.

Executing Actions

To execute an action within the action body of another action—referred to as the *calling action*—use the following syntax:

```
execute action_name (parameter(s));
```

In the calling action, `action_name` must be defined as a local parameter of type action:

```
local: action<action_name> param_name
```

Input parameters are listed by keyword, not position. In addition, you can omit parameters. The syntax for each input parameter is:

:keyword value

keyword is the name of the parameter as defined in the action that is executed.

The following example illustrates these concepts. Here is an action that will be executed:

```
action add_three_ints

    (input:          int arg1 = 0,          // Three input parameters
      input:          int arg2 = 0,
      input:          int arg3 = 0,
      output:         int result,          // One output parameter
      no_context:)

{
    result = arg1 + arg2 + arg3;
    succeed();
}
```

The following action executes the action add_three_ints:

```
action add_some_ints

    (local:          action<add_three_ints> add_three_ints,
      no_context:)

{
    execute add_three_ints(:arg1 1, :arg2 1, :arg3 2);
    PRINTF("\nResult is: %d\n", add_three_ints.result);
    execute add_three_ints(:arg2 1, :arg3 2, :arg1 1);
        // Notice that arguments don't need to be in positional order
    PRINTF("\nResult is: %d\n", add_three_ints.result);
    succeed();
}
```

New Rule for Invoking Action

In certain cases, there is no longer a need to declare a local action. However, PeopleSoft has maintained backward compatibility in that you can still declare a local action in these cases. As long as you define an action in the current scope and are not using it as a variable, you can invoke

it without declaring it as a local parameter. Please note that this new rule applies only to actions declared as local. If you want to declare an input, output, or inout variable of type action, you will have to use an action schema for its data type.



For more information and an example of how to do this, see [Writing Action Parameters](#).

Example:

```
action b();

action a() {
    execute b();
}
```

instead of:

```
action b();

action a(local: action<b> b) {
    execute b();
}
```

Passing Output in execute Statement

This section explains the new Release 8.0 rule for returning output values in execute statements. However, we have maintained backward compatibility in that you can still pass output using the method of previous versions. In an "execute" statement, you can pass an actual argument to an "output:" parameter instead of writing an explicit assignment after the "execute" statement (for backward compatibility, you must still write an explicit assignment for an "inout:" parameter.)

Example:

For the action

```
action some_action(output: int o) {<body>;}
```

Starting in Release 8.0, you can pass output in the following way:

```
action spl_main ()
{
    execute some_action(:o some_variable);
}
```

In previous releases, you could only pass output in the following way:

```

action spl_main ()
{
    execute some_action();

    some_variable = some_action.o;
}

```

Writing Methods

Methods on objects are implemented as a slot that is the name of the method. This slot is usually a class slot, so it is shared by all instances of the class. The value of the method slot is the name of a PepperCode action to call that implements this method. Note that the actions that implement a particular method must share the same action schema.

When PepperCode code calls a method, the underlying code looks up the ActionSchema for the named action, and creates a C++ method call to the action's Action_Execute() method, passing a data structure that holds the parsed parameters to the actual C++ code.

In a class definition, a slot of type action is an implementation of a PepperCode method, like a C++ member function or method:

```
action<schema_name> name
```

The action method stored in an action slot can be referenced and executed. Any action of that schema type can be assigned to that slot.

The dispatch of a method—the process of calling the correct method associated with a class—is not performed automatically. Instead, the value of a local action parameter is defined and the action is called through the local parameter.

Following are three examples illustrating these concepts.

Implementing A Method: Example 1

Following is an example of how to implement a PepperCode method. It includes storing actions on objects and dispatching those actions. The action that will be executed uses the schema print_str:

```

action_schema print_str

    (input:                string pstring = "Null",

     output:               int printed,

     no_context:);

```

Here are the actions that use this schema:

```

action<print_str> print_string
    ()
{
    PRINTF("\n%s", pstring);

    printed = 1;

    succeed();
}

action<print_str> print_indented_string
    ()
{
    PRINTF("\n      %s", pstring);          // Indent the string

    printed = 1;

    succeed();
}

```

Next, a class has a slot that stores an action of schema type `print_str`. That slot has a default value of action `print_string`:

```

class Printable_Object : Spl_Class {
    string          description
    action<print_str> print_action
};

slot Printable_Object.description { default: "default description" };
slot Printable_Object.print_action { default: print_string class_slot: };

```



Note: In the common case, each instance of a particular class will have its action slots set to the values as every other instance of that class. In that case, it is important to use the `class_slot` keyword on each of those slots to reduce the amount of memory consumed. Only if different instances will store different actions on the same slot should you omit that keyword.

Finally, the actions are executed from another action:

```

action dispatch_print_string
    (input:          instance<Printable_Object> printable_object,

```

```

    local:          action<print_str> print_string_action,

    output:        int printed,

    no_context:)

{

    // Retrieve print string from object

    print_string_action = printable_object.print_action;

    // Call the action

    execute print_string_action

        (:pstring printable_object.description);

    // Retrieve and set the output variable

    printed = print_string_action.printed;

    succeed();

}

```

Implementing A Method: Example 2

Here is an example that illustrates both the use of action schemas and polymorphism. A classic example of polymorphism is a program that draws shapes such as circles, squares, and triangles. Good object-oriented design suggests that the code that draws a list of objects should not need to know the shape of each object or how to draw a particular shape. Instead, it should ask each object to draw itself, and the code packaged within each object should know the shape of that object and how to draw it.

This example uses an action schema `draw` to provide the “signature” for a family of actions that draw shapes. Then it creates three actions belonging to this family: `draw_circle`, `draw_square`, and `draw_triangle`. In a realistic example, each action would contain code specific to that shape, but for simplicity this example just prints a message.

The example also creates a parent class `shape` and three subclasses for the specific shapes. Default statements associate the appropriate action with each subclass: for example, `draw_triangle` is associated with `triangle`. Notice the use of `no_context`: in each schema statement to avoid wasting memory.

The action creator creates an instance of each of the possible shapes and invokes the action drawer to draw each shape. Notice that drawer merely asks each object to draw itself: it does not need to know what shape the object has or how to draw that shape. You could add a new kind of shape such as an ellipse without altering drawer at all, which is evidence that this mechanism achieves the goals of polymorphism and encapsulation.

```

// class declaration

```

```
class shape;

// See Understanding Operators And Functions for information on the
// PepperCode runtime functions listed here.

cpp_function instance<shape> CREATE_OBJECT(string, string) "cpp_create_object";
cpp_function void PRINTF(string) "printf";
cpp_function void INIT_CLASSES(void) "initialize_spl_objects";

// Generic action to draw a shape
action_schema draw

    (input: instance<shape> the_shape,
     no_context:);

// Specific action to draw a circle
action<draw> draw_circle()
{
    PRINTF("Draw circle\n");
    succeed();
}

// Specific action to draw a square
action<draw> draw_square()
{
    PRINTF("Draw square\n");
    succeed();
}

// Specific action to draw a triangle
action<draw> draw_triangle()
{
    PRINTF("Draw triangle\n");
    succeed();
}

// Generic class for shape; specific classes for circle, square,
```

```

// triangle, each having the appropriate action or "method"

class shape {

    action<draw> draw_myself

};

class circle: shape {};

slot circle.draw_myself { default: draw_circle class_slot: };

class square: shape {};

slot square.draw_myself { default: draw_square class_slot: };

class triangle: shape {};

slot triangle.draw_myself { default: draw_triangle class_slot: };

// Tell the shape to draw itself; this executes draw_circle, draw_square,
// or draw_triangle—whichever "method" is associated with this instance
// of the polymorphic shape

action drawer

    (input: instance<shape> the_shape,

     local: action<draw> draw,

     no_context:)

{

    draw = the_shape.draw_myself;

    execute draw(:the_shape the_shape);

    succeed();

}

// Creates three objects which can draw themselves using the appropriate
// "method"

action creator

    (local: instance<shape> chalk,

     local: instance<shape> trafilgar,

     local: instance<shape> bermuda)

{

```

```

    chalk = CREATE_OBJECT("chalk", "circle");

    trafalgar = CREATE_OBJECT("trafalgar", "square");

    bermuda = CREATE_OBJECT("bermuda", "triangle");

    execute drawer(:the_shape chalk);

    execute drawer(:the_shape trafalgar);

    execute drawer(:the_shape bermuda);

    succeed();

}

```

To run this code, follow the compilation and linking process.



For more information, refer to Compiling And Linking PepperCode.

First, link it into an existing Production Planning software product that generates an executable called standalone. Running the program with the command line option -I causes it to prompt on the keyboard for a transaction to be invoked by the Action Interpreter. Type creator() to invoke the test case and then :exit to leave the interpreter and end the program:

```

shell> ./standalone -I

Checking ResponseAgent configuration.....Done Initializing Runtime Object
System...

...Done

Setting app name to 'standalone'

Initializing standalone...

Initializing communication buffer and hash table...MJD...Done

Creating slot classes.....Done

Creating slot specifier classes.....Done Initializing Schedule.....Done

Creating the Base Class.....Done

Creating form classes.....Done

...Done

Entering interpreter mode...

Enter an action call: creator()

Draw circle

```



```

Draw square

Draw triangle

Result: ( :RESULT 3 )

Enter an action call: :exit

...Done.

shell>

```

Implementing A Method: Example 3

This example of PepperCode code is followed by a section-by-section analysis of that code.

```

// Forward declaration of the class graphic_object

class Graphic_Object;

// Declaration of the type of the method.

action_schema Display (input: int file_handle, instance<graphic_object> this);

// Definition of the base object class

class Graphic_Object {

    int x;

    int y;

    action<Display> print_method;

};

// Define the action that executes for this method for the Graphic_Object class

slot Graphic_Object.print_method {:default_value default_print};

// Create a subclass of the Graphic_Object class. This subclass has a new
member variable.

class Circle : Graphic_Object {

    int radius;

};

// Override the default action with a new one for the print_method.

slot Circle.print_method {:default_value default_print};

Create a second subclass of the Graphic_Object class. This as 2 new member
variables.

```

```

class Square : Graphic_Object {

    int width;

    int height;

};

// Note that you DO NOT override the print_method.

// Implement the actual action for the default print method.
action<Display> default_print ()

{

FPRINTF (file-handle, "x = %d, y = %d\n", this.x, this.y);

};

// Implement the action for the circle print method.
action<Display> circle_print (

local: instance<Circle> this_circle)

{

this_circle = this; // Unsafe cast from graphic_object to a circle;

FPRINTF (file-handle, "x = %d, y = %d radius = %d\n", this.x, this.y,
this_circle.radius);

};

// Implement a transaction to test this code out.
action<Transaction> transaction_test (

local: instance<Graphic_Object> my_instance,

local: action<Display> print_method)

{

// Assume an action exists to create a graphic object
execute create_graphic_object (x: 10 y: 15);

// Grab the new object, and cast it to the base class.
my_instance = create_graphic_object.new_instance

// Call the method.

//

// Note that a future version of the compiler should support

```

```

// execute my_instance.print_method (file_handle: file_handle);

//

print_method = my_instance.print_method();

execute print_method (this: my_instance file_handle: file_handle);

// Assume a action exists for creating circles.

Execute create_circle (x: 1 y: 2 radius: 3);

// Grab the new object and cast it to the base class.

My_instance = create_circle.new_instance;

// Call the print_method

print_method = my_instance.print_method;

execute print_method (this: my_instance file_handle: file_handle);

// Assume a action exists for creating squares.

Execute create_square (x: 4 y: 5 width: 6 height: 7);

// Grab the new object and cast it to the base class.

My_instance = create_square.new_instance;

// Call the print_method

print_method = my_instance.print_method;

execute print_method (file_handle: file_handle);

};

```

Following is an explanation of the previous code, one section at a time.

```

// Forward declaration of the class graphic_object

class graphic_object;

```

This line tells the PepperCode compiler that a class named Graphic_Object exists, and that it can be referenced. This is a declaration only for the compiler to do type checking.

```

// Declaration of the type of the method.

action_schema Display (input: int file_handle, instance<graphic_object> this);

```

Declares an action schema of type Display. This declaration tells the compiler that there is a family of actions that will all have the same calling structure. Actions that are members of this family will be interchangeable because the parameter structure is known. This creates a new type in the PepperCode file that can be used as action<Display>.

```

// Definition of the base object class

```

```

class Graphic_Object {

    int x;

    int y;

    action<Display> print_method;

};

// Define the action that executes for this method for the Graphic_Object class

slot Graphic_Object.print_method {:default_value default_print

:class_slot};

```

This code declares a PepperCode class of objects. This class has three member variables. One of those member variables has shared storage that is shared between all instances of the class. This is designated with the `:class_slot` keyword in the slot statement. The two member variables `x`, and `y` are simple integer members that hold a coordinate point for any graphic object.

The `action<Display> print_method;` line declares a slot of type `action`. This slot will hold the name of an action to be called as a method. The action that is the value of this slot must be of type `Display`—declared as `action<Display>`. This creates a method on a object, and the name of the method is the member variable name.

The slot line sets the default value of the method to be the action named `default_print`. Note that this method has not yet been defined. That is OK since the method is not accessed until the method is executed at run time. At compile time a member is created to hold the action name and declare its type.

```

// Create a subclass of the Graphic_Object class. This subclass has a new
member

//   variable.

class Circle : Graphic_Object {

    int radius;

};

// Override the default action with a new one for the print_method.

slot Circle.print_method {:default_value default_print};

```

Creates a subclass of the `Graphic_Object`. Note this class inherits all structure and behavior from the class `Graphic_Object`. This example adds a new slot named `radius`. Since this class has a new slot, you are going to need a new way of printing the data for the object, so you should override the `print_method` with a new action. This is done in the slot statement. Note that the slot statement changes the default value of this slot for the class `Circle`. Since this is a class slot—only one piece of memory shared by all instances—the method is now set for all instances of the `Circle` class.

Create a second subclass of the `Graphic_Object` class. This has two new member variables.

```

class Square : graphic_object {

    int width;

    int height;

};

// Note that you DO NOT override the print_method.

```

Creates a second subclass of the Graphic_Object class. This new subclass also has new structure. Since the print_method is not overridden, instances of this class will use the default print_method, and the new structure will not be printed out.

```

// Implement the actual action for the default print method.

action<Display> default_print ()

{

    FPRINTF (file-handle, "x = %d, y = %d\n", this.x, this.y);

};

// Implement the action for the circle print method.

action<Display> circle_print (

    local: instance<Circle> this_circle)

{

    this_circle = this; // Unsafe cast from graphic_object to a circle;

    FPRINTF (file-handle, "x = %d, y = %d radius = %d\n", this.x, this.y,
    this_circle.radius);

};

```

Implements actions for the two version of the method. Note that these actions do very simple things. They print the slots of the object to a file handle. Later on you can write new methods that take a file or screen handle and have different behavior.

Note the base method prints out the x and y coordinate, and the circle method prints out the x, y, and radius slots.

```

// Implement a transaction to test this code out.

action<Transaction> transaction_test (

    local: instance<Graphic_Object> my_instance,

    local: action<Display> print_method)

{

```

This routine tests out the code.

It starts by declaring two local variables to the action that will hold instances of `Graphic_Object`, and a variable to hold the method. The need for the variable to hold the method is because the compiler doesn't yet support the desired functionality and will be made obsolete in the future.

```
// Assume an action exists to create a graphic object

execute create_graphic_object (x: 10 y: 15);

// Grab the new object, and cast it to the base class.

my_instance = create_graphic_object.new_instance
```

Calls an action named `create_graphic_object` that you have not explicitly written in this example, but assume it creates an object of the class `Graphic_Object` and assigns the `x` and `y` values passed in the member variables. An output parameter of this object is the `new_instance`, which you retrieve, and assign to the local variable. Note the local variable is of the base class, and so this is a cast.

You now have a `Graphic_Object` in the local variable with `x = 10`, and `y = 15`.

```
// Call the method.

//

// Note that in a future version of the compiler it should support

// execute my_instance.print_method (file_handle: file_handle);

//

print_method = my_instance.print_method();

execute print_method (this: my_instance file_handle: file_handle);
```

The local variable `print_method` gets assigned as its value the value of the method slot `print_method` from the object held in `my_instance`. You then execute the action that is the value of the local variable, passing it the object, and the `file_handle` parameter. The underlying C++ code that is generated by the PepperCode compiler takes the string value that is the name of the action which is actually stored in the slot, and looks up a definition of the named action. It then parses the parameters passed into C++ member slots on that object, and calls the `action_execute` method on the underlying C++ object, which implements the body of the action. The action lookup happens in-line with the `execute` statement in the generated C++ code. This means that the value of the action slot can be changed at any time up to and including the line before the `execute` statement, and this would affect the action actually executed.

The file pointed to by `file_handle` would get the following text put into it:

`x = 10, y = 15`

```
// Assume a action exists for creating circles.

Execute create_circle (x: 1 y: 2 radius: 3);

// Grab the new object and cast it to the base class.
```

```

My_instance = create_circle.new_instance;

// Call the print_method

print_method = my_instance.print_method;

execute print_method (this: my_instance file_handle: file_handle);

```

Follow the same procedure for a circle object. Note that a different method is executed, because the value of the `print_method` slot for Circle objects is `circle_method`, and this results in the C++ code looking up a different actionschema, and consequently a different function is executed.

```

// Assume a action exists for creating squares.

Execute create_square (x: 4 y: 5 width: 6 height: 7);

// Grab the new object and cast it to the base class.

My_instance = create_square.new_instance;

// Call the print_method

print_method = my_instance.print_method;

execute print_method (file_handle: file_handle);

};

```

Do the same thing for squares, but because you did not override the `print_method` slot for the square class the default `_print` action is called to implement this method.

A very important consequence of implementing methods as slots that hold actions is that behavior of a object can be customized very simply at run time. Suppose you have an object that is deeply imbedded inside the Planning code. This object has a method that performs a desired operation. For example, a `planning_period` object has a `consume` method that takes a SO, and consumes forecasts from the planning period. Further you have offered a single action that implements this method. This action consumes forecast for the sales order from the planning period the sales order was taken in—at the order date. If the forecast for that period is already consumed to 0, then no further consumption takes place.

Suppose a service partner wants to customize this code to consume any amount over the periods forecast from the nearest prior, or later period with unconsumed forecast. This customization is easy. Write the new `Consume` method in a customer module, and change the default method on the planning period to be the new method by using the transaction `transaction_set_slot_default` (class, slot, value);

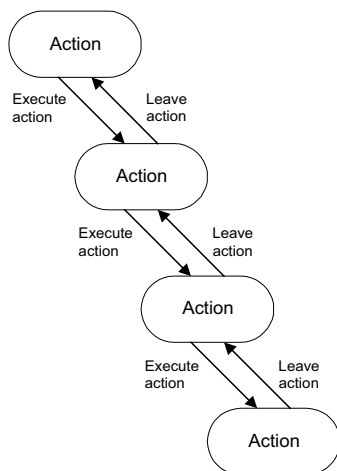
Understanding Context

When an action is executed within the body of another action (the calling action), a new *context* is created. A context is a mechanism for maintaining objects and their values. You can think of a context as a collection of all the objects that it creates or deletes, and all the slot values it changes. A context is valid as long as a calling action has not exited.

The PepperCode exit statements determine whether changes made to class and object slots are accepted or rejected:

- `succeed()`;—Accept changes.
- `fail()`;—Reject changes. The world outside the action is restored to the state it held before the action started executing.
- `leave()`;—Remember changes so they can be restored later, but don't change the data values. If the action is executed again with the same context, the remembered values are restored and can be modified; any one of the exit statements can then be used on that data.

You must execute one of these statements before a context is accepted or rejected. If you have a hierarchy of calling actions, you can leave several actions; as long as a calling action higher in the hierarchy has not exited, you can go back to the contexts lower in the hierarchy, as illustrated here.



Leaving and restoring multiple contexts in a hierarchy of calling actions

For example, you can re-execute Action2 after leaving it and returning to Action1. On that second execution, the values of parameters and slots of classes revert to the values they had before Action2 was left.

Multiple contexts is a powerful feature of PepperCode that enables you to explore possible solutions. Binding an action to multiple contexts lets you try independent experiments and postpone selecting the optimum outcome until the experiment is complete.

In addition to an action using the `succeed`, `fail`, or `leave` statements on itself, a parent action can also pass the `succeed` or `fail` functions a context—an action parameter name—to discard or accept the changes of a child action. For example:

```

execute var0();

execute var1();

if (GT(var0.score, var1.score))

```



```

        succeed(var0);

else

        succeed(var1);

```

In general, actions that do not change data values for an object should use the `no_context:` keyword, while actions that do change values probably need a context. The `no_context:` keyword saves memory and computing resources by not copying object data values.

Although `context:` is the default, this keyword is useful if, for example, a schema uses `no_context:` and you want to override that behavior in an action.



When changing the value of an object in an action that could fail, always give the action a context; specifically, don't use the `no_context:` keyword.

To illustrate how the context mechanism works, consider this simple example that uses an instance of a class called `gradebook` to keep track of the scores on a school examination. A child action called `enter_one_score` makes the changes in the `gradebook`; a parent action called `enter_scores` calls the child to enter three different scores.

To show one method of using context, you can arrange for the child to return with a `leave` statement. The parent then executes either a `succeed` statement to accept the child's changes, or a `fail` statement to reject them:

```

#include "cpp_utility.spl"

class gradebook {

    oset[int] grades

    int average

    int high_score

    int low_score

};

slot gradebook.high_score { default: 0 };

slot gradebook.low_score { default: 100 };

//

// This action enters a new score into the gradebook and updates the
// average, the high score, and the low score. Then it returns without
// either committing or retracting the changes that it has made. The
// parent action can then choose whether to accept or reject the changes.

//

```

```

action enter_one_score

    (input: instance<gradebook> g,

    input: int score,

    local: int sum)

{

    g.high_score = MAX(g.high_score, score);

    g.low_score = MIN(g.low_score, score);

    g.grades.push(score);

    sum = 0;

    foreach grade in g.grades

        sum = ADD(sum, grade);

    g.average = DIV(sum, g.grades.length());

    PRINTF("average/high/low inside enter_one_score:\t%d %d %d\n",

        g.average, g.high_score, g.low_score);

    leave;          // Let the parent action decide to accept or reject changes
}

//

// For demonstration purposes, this parent action calls enter_one_score to
// enter three scores into the gradebook. The first two times, it accepts
// the changes made by the child action. The third time, it rejects the
// changes.

//

action enter_scores

    (local: instance<gradebook> g)

{

    // Create a gradebook for chemistry class

    g = CREATE_OBJECT("chemistry_class", "gradebook");

    execute enter_one_score(:g g, :score 99); // Enter a score

    PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n",

```

```

        g.average, g.high_score, g.low_score);
succeed(enter_one_score);    // Accept the changes

PRINTF("average/high/low after accepting changes:\t%d %d %d\n\n",
        g.average, g.high_score, g.low_score);

execute enter_one_score(:g g, :score 75); // Enter a score

PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n",
        g.average, g.high_score, g.low_score);

succeed(enter_one_score);    // Accept the changes

PRINTF("average/high/low after accepting changes:\t%d %d %d\n\n",
        g.average, g.high_score, g.low_score);

execute enter_one_score(:g g, :score 50); // Enter a score

PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n",
        g.average, g.high_score, g.low_score);

fail(enter_one_score);      // Reject the changes

PRINTF("average/high/low after rejecting changes:\t%d %d %d\n\n",
        g.average, g.high_score, g.low_score);

succeed();

}

```

When you run the example, it prints these results:

average/high/low inside enter_one_score:	99 99 99
average/high/low outside enter_one_score:	0 0 100
average/high/low after accepting changes:	99 99 99
average/high/low inside enter_one_score:	87 99 75
average/high/low outside enter_one_score:	99 99 99
average/high/low after accepting changes:	87 99 75
average/high/low inside enter_one_score:	74 99 50
average/high/low outside enter_one_score:	87 99 75
average/high/low after rejecting changes:	87 99 75

The PRINTF statement for inside enter_one_score demonstrates that within the child the changes always take place immediately, as they would in any programming language. But the PRINTF

statement for outside `enter_one_score` demonstrates that the changes disappear after the child executes the `leave` statement.

Only after the parent executes the `succeed` statement—in the first two cases—do the child's changes appear in the parent's environment. And when the parent executes the `fail` statement—in the third case—the child's changes disappear forever.

The previous example gives the parent control over the child's changes. Alternatively, the child can control whether its own changes ever appear in the parent's environment. Now change the example slightly to show how this works. This time, the child normally accepts its own changes with `succeed`, but if the grade average falls too low, it rejects them with `fail`:

```
#include "cpp_utility.spl"

cpp_function void INIT_CLASSES() "initialize_spl_objects";

class gradebook {
    oset[int] grades

    int average

    int high_score

    int low_score
};

slot gradebook.high_score { default: 0 };
slot gradebook.low_score { default: 100 };

//

// This action enters a new score into the gradebook and updates the
// average, the high score, and the low score. Then it returns without
// either committing or retracting the changes that it has made. The
// parent action can then choose whether to accept or reject the changes.
//

action enter_one_score
    (input: instance<gradebook> g,
     input: int score,
     local: int sum)
{
    g.high_score = MAX(g.high_score, score);
    g.low_score = MIN(g.low_score, score);
```

```

    g.grades.push(score);

    sum = 0;

    foreach grade in g.grades
        sum = sum + grade;

    g.average = sum / g.grades.length();

    PRINTF("average/high/low inside enter_one_score:\t%d %d %d\n",
        g.average, g.high_score, g.low_score);

    // If the score brought the average below 80, discard the changes.
    if (g.average < 80)
        fail();

    succeed();
}

//

// For demonstration purposes, this parent action calls enter_one_score
// to enter four scores into the gradebook. Whether the child's changes
// propagate to the parent depends on how the child returns to the parent.
//

action enter_scores
    (local: instance<gradebook> g)
{
    // Create a gradebook for chemistry class
    g = CREATE_OBJECT("chemistry_class", "gradebook");

    execute enter_one_score(:g g, :score 99);

    PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n\n",
        g.average, g.high_score, g.low_score);

    execute enter_one_score(:g g, :score 75);

    PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n\n",
        g.average, g.high_score, g.low_score);

    execute enter_one_score(:g g, :score 50);

```

```

    PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n\n",
           g.average, g.high_score, g.low_score);

    execute enter_one_score(:g g, :score 82);

    PRINTF("average/high/low outside enter_one_score:\t%d %d %d\n\n",
           g.average, g.high_score, g.low_score);

    succeed();
}

```

When you execute this second example, it prints:

```

average/high/low inside enter_one_score:          99 99 99
average/high/low outside enter_one_score:          99 99 99
average/high/low inside enter_one_score:           87 99 75
average/high/low outside enter_one_score:          87 99 75
average/high/low inside enter_one_score:           74 99 50
average/high/low outside enter_one_score:          87 99 75
average/high/low inside enter_one_score:           85 99 75
average/high/low outside enter_one_score:          85 99 75

```

Notice that for the first, second, and fourth invocations of `enter_one_score`, the parent sees the changes made by the child. But on the third invocation, the child executes the fail statement—rejecting its own changes—and from the viewpoint of the parent, it seems as if the changes had never been made.

Accessing Action Status

The PepperCode compiler predeclares for you an enumeration called `Action_Status`:

```
enum Action_Status { LEAVE, FAIL, SUCCEED };
```

For any action that doesn't already have a parameter named `status`, the compiler adds an output parameter of type `Action_Status`:

```
output: enum<Action_Status> status
```

You can query this parameter to determine whether the action executed a leave, fail, or succeed statement. For example:

```

action child(input: int i)
{

```

```

        if (i == 5)
            succeed();

        fail();
    }

    action parent()
    {
        execute child(:i 5);

        if (child.status == SUCCEEDED)
            PRINTF("success\n");

        PRINTF("failure\n");

        succeed();
    }

```

Understanding How Actions Are Executed

Many computer languages provide a runtime system that invokes a specific function to start the program. For example, a C program starts with the main function; a Pascal or Fortran program starts at a program statement.

The PepperCode runtime system, however, provides an Action Interpreter, which is code that reads a string containing a human-readable action invocation that is similar to the syntax you use in a PepperCode execute statement. The Action Interpreter parses the string, invokes the action, and returns a string containing a human-readable list of output values.

The Action Interpreter is used in several ways:

- When you use the GUI interface to the Planning software, it passes strings to the Action Interpreter to perform commands.
- When you create a command file for use as an input script to the Planning software, the Action Interpreter reads the action invocations specified in the file.
- When a client invokes a remote procedure on the server, the invocation may involve the Action Interpreter.
- The Planning software source code defines a cpp function called `SERVER_EXECUTE_ACTION` that invokes the Action Interpreter.

An execute statement is usually not executed by the Action Interpreter, except in certain client-server situations when `#remote_include` is used.

In addition, you can define an Interpreter menu item for debugging.



For more information, refer to Using The Action Interpreter.

Action Execution & Definitions

If a PepperCode program contains an action called "spl_main", that action will be called automatically when the program starts, simulating the statement:

```
execute spl_main();
```

If a PepperCode program lacks such an action, it will not execute any PepperCode statements until it reads an action invocation from a networked client, a command file, or the keyboard.

You should design spl_main to take three parameters:

```
action spl_main
    (input: int argc,
     input: oset[string] argv,
     input: string identity)
```

The argc and argv parameters correspond to their namesakes in the C or C++ main function. The identity parameter is set to client, server, or standalone, depending on whether the program has been linked to serve as a client, a server, or a monolithic program.

This plays a role similar to that of the main function in C or C++; because it's invoked before any other action, it's a good place to perform initializations, such as starting the graphical user interface (GUI).

Every PepperCode-generated program accepts these options (these are options to the generated program, not to the PepperCode compiler):

-file filename Read action invocations from "filename"

-I Prompt for action invocations on stdout and read them from stdin

An action invocation inside a file or on stdin looks like an "execute" statement without the "execute" keyword:

```
transaction_do_something(:name "xyz", :value 123)
```

Using Transaction Logs

A transaction is an action that is part of the external interface of Planning software. It can be invoked by the Action Interpreter when a menu item is selected within the Planning software or when in a command file is run.

The PepperCode Action Interpreter can log each transaction it executes that changes data on the server. If you took a snapshot of the data at the beginning of a session with Planning software, then performed all of the transactions listed in the log file generated by that session, the resulting changes to the data would be the same as those that occurred during the actual session. So, the log file can help you to restore the system to the state before a session was executed.

Clients cannot create a transaction log file. The Action Interpreter never logs an action executed by the client on the client, but it can log an action executed by a client on the server.

The Action Interpreter writes to a log file—as specified in the .rps file—a timestamp and a human-readable string describing the transaction invocation. Here is an example of a log entry:

```
// 01/19/95 20:58:15 PST  
  
transaction_factorial (:factor 4 :error 0)
```

Notice that the log entry doesn't have commas between arguments or a terminating semicolon.

By default, every transaction invoked by the Action Interpreter is written to the log file.

CHAPTER 6

Writing Control Statements

PepperCode has the following types of control statements:

- assignment (=)
- if-else
- while
- foreach
- succeed, fail, or leave
- subaction execution
- expressions for executing cpp functions
- execute statement



For more information about action exits, succeed, fail, and leave, refer to Understanding Context.

Writing Assignment Statements

```
<lvalue_expression> = <rvalue_expression>;
```

An assignment statement evaluates the two expressions and assigns the rvalue expression to the lvalue.

For strings, instances, lists, and classes, the assignment operator behaves like the default assignment operator for a C pointer: assignment copies the pointer but does not duplicate the target. If the destination list is a slot on an object, however, it receives a duplicate copy of the list. If the destination list is an action parameter, both source and destination point to the same list (no copy is made).

Only an lvalue expression can be the target of an assignment, or the actual parameter corresponding to a formal parameter which is an "inout:" or "output:". An expression is an lvalue if it satisfies all of the following:

- It is a "local:", "output:", or "inout:" parameter of the current action.

- Its data type is not "action" (the data type may be "action schema").

Writing if-else Statements

The PepperCode if-else statement is similar to the C++ if-else statement:

```
if ( <expression> ) <statement> ;

if ( <expression> ) <statement> else <else_stmt> ;
```

The expression must evaluate to type "int", "date", "time", "instance", "class", or "string".

<statement> is either one statement or multiple statements within braces ({ }). At least one statement or an empty block ({ }) is required after the if and, when used, the else.

Following is an example of an if-else statement that could be placed in an action body:

```
if ((string_length < 2) {           // If the length is less than 2,
    PRINTF("\nstring < 2");         // print a warning
    fail();                         // and fail from the action.
}

else {                             // If not less than 2,
    PRINTF("\n%s", pstring);        // call C printf
    succeed();                     // and succeed from the action.
}
```

Writing while Statements

The PepperCode while statement is similar to the C++ while statement:

```
while ( <expression> ) <statement> ;
```

The expression must evaluate to type "int", "date", "time", "instance", "class", or "string".

<statement> is either one statement or multiple statements within braces ({ }).

The expression is evaluated first and then before each iteration. The statement repeats until the expression evaluates to false.

Here is an example of using the while statement:

```
cpp_function int PRINTF(string) "printf";

action print_n_times
```

```

        (input: string message,
         input: int counter)
    {
        while (counter > 0)
        {
            PRINTF("message");

            counter = SUB(counter, 1);
        }
    }

```

Here's another example:

```

action a(local: int i)
{
    i = 0;

    while (i < 10)
    {
        PRINTF("%ld\n", i);

        i = i + 1;
    }

    succeed();
}

```

Writing foreach Statements

The foreach statement iterates over an osset or array:

```

foreach id in <expression> <statement> ;

foreach id in reverse <expression> <statement> ;

```

The <expression> must have type osset or array; "id" declares a temporary variable whose type matches the element type of the osset or array and whose scope is <statement>. For each element of the osset or array, we assign that element to "id" and then execute <statement>.

For an osset, we visit the elements in order from head to tail (or, for the "in reverse" form, from tail to head).

For an array, we visit the elements in a repeatable order which is the same for every target machine, but which may change if one adds or deletes elements. The "in reverse" form is not allowed.

Adding or deleting elements within <statement> is an error, except that it is safe to delete any element which has already been assigned to "id".



For more information, refer to Writing Osets and Writing Arrays.

<statement> is either one statement terminated with a semicolon (;) or multiple statements within braces ({ }).

If the oset is empty, <statement> does not execute.

To move through the list from the last element to first element, use the reverse keyword:

```
foreach item in reverse oset statement
```

Here is a straightforward example of the use of foreach to traverse a list of integers, summing them. Notice that the loop index variable element is declared for you by the compiler:

```
action sum_int_list
  (input: oset[int] int_list,
   output: int sum,
   no_context:)
{
  sum = 0;
  foreach element in int_list
    sum = sum + element;
  succeed();
}
```

Here is another example of a foreach statement:

```
action a
  (inout: int i,
   input: array[int] l,
   output: int o)
{
```

```

o=0;          // Output variable o

foreach o in l      // Index variable o
{
    i = i + o;      // Index variable o
}

o = i;          // Output variable o

succeed();

}

```

Writing execute Statements

The execute statement invokes an action.

```
execute <identifier> ( <actual arglist> ) ;
```

Invokes the action specified by <identifier> and pass to it the <actual arglist>. If <identifier> names a local variable of type action, invoke the corresponding action and save its state in that variable; if it names an action, create an implicit local variable to save its state.



For more information about execute statement, refer to Executing Actions.

Writing succeed, fail, or leave Statements

```

leave ;

fail ;

succeed ;

```

Returns from the current action, telling the context mechanism to leave its changes uncommitted; or to roll back its changes; or to commit them.

```
succeed ( <identifier> ) ;
```

If <identifier> is the name of a local variable that represents an action with uncommitted change, this statement commits those changes. If there are no uncommitted changes, a runtime error occurs. The identifier must name an explicit local variable, not an implicit one created by using the "execute" statement with an action name.



For more information about action exits, succeed, fail, and leave, refer to Understanding Context.

Writing break and continue in Loops

The break statement breaks out of a loop. Use it in a foreach or while loop to break out of a loop if a certain condition occurs. For example, the following code prints a list of integers from 1 to either 100 or to the integer j that is input into this action, whichever is lesser.

```
action a_break(  
    input: int j,  
    local: int i)  
{  
    i = 1;  
    while (i < 101)  
    {  
        PRINTF("%ld\n", i);  
        i = i + 1;  
        if (i > j) break;  
    }  
    succeed();  
}
```

The continue statement continues to the end of a loop without breaking out of the loop. Use it in a foreach or while loop to skip code in a loop if a certain condition occurs. For example, the following code prints a list of 100 integers. The list will be either the numbers 1 to 100, or if the integer j is less than 100, the list will be the numbers 1 to j and then j, which is printed 100 minus j times.

```
action a_continue(  
    input: int j,  
    local: int i)  
{  
    i = 0;  
    while (LT(i, 100))
```



```

    {

        i = i + 1;

        if (i > j) continue;

        PRINTF ("%ld\n", i);

    }

    succeed();

}

```

Starting in Release 8.0, you cannot use a C++ function statement to declare BREAK or CONTINUE within PepperCode; these keywords are automatically recognized by the compiler.

The BREAK and CONTINUE statements were originally implemented by users as C++ functions, then added to the compiler as pseudo C++ functions, so in previous releases of PepperCode the users did not need to define them. The Release 8.0 Compiler provides "break" and "continue" statements and also BREAK and CONTINUE statements as intrinsic functions (built-in functions), so these cpp_function declarations are no longer needed.

In version 7.5.2 or earlier, if you use break or continue, you had to include the file cpp_utility.sp1 or use the following statements:

```

cpp_function void BREAK() "break";

cpp_function void CONTINUE() "continue";

```

Writing Enumerations in Loops

Enumerated types—data type enum—are implemented as named object instances. As a result, the PepperCode object system looks up the enumeration each time it's referenced, which is why many references to an enumeration can be slow. If many lookups are required to execute a piece of code, that piece of code may perform poorly. For example, the following code looks up the enumerator instance 100 times:

```

action slow_enumerator

    (local: enum<Boolean_Flag> the_flag,

     local: int index)

{

    index = 0;

    while (index < 100) {

        // The next line creates C++ code that looks up the enumeration

```

```

        // TRUE by name each time through this loop. This is a classic
        // invariant value inside a loop.

        the_flag = TRUE;

        index = index + 1;

    }

}

```

Every time the loop is traversed the enumeration TRUE is looked up. This lookup is slow if executed many times.

An alternate implementation that looks up the enumerator once is as follows:

```

action slow_enumerator (

    (local: enumerator<Boolean_Flag> the_flag,

    local: enumerator<Boolean_Flag> true_flag,

    local: int index)

{

    index = 0;

    // The C++ code looks up the enumeration instance and stores its
    // value in the local true_flag.

    true_flag = TRUE;

    while (index < 100) {

        the_flag = true_flag;

        index = index + 1;

    }

}

```

The implementation of enumerations will change in future releases, so this coding technique won't be required. In addition, this technique is required only if the enumeration is going to be looked up many times: either inside a loop or inside an action called in a loop. For enumerations that are accessed only a few times, this technique is not required, since the performance penalty is not as large for each lookup.

Following is another example. This code references a Boolean enumeration inside of a loop:

```

foreach item in items {

    if (item.value == TRUE)          // slow use of an enum

```

```
...
}
```

The above code could be written much more efficiently by assigning TRUE to a local enum parameter and then using the parameter in the if statement; for example:

```
...

local: enum<Boolean_Flag> true_enum,

...

true_enum = TRUE;

foreach item in items {

    if (item.value == true_enum)

        ...

}
```

Using Dot Notation in Expressions

Within expressions, you use dot notation to access the value stored on a slot of an instance or class variable:

```
variable_name.slot_name. ... slot_name
```

The following example shows that slots accessed with dot notation can be used in the same way that you would use parameters or literal constants:

```
class C{

    int first_slot

    int second_slot

    int third_slot

};

class D{

    instance <C> nested

};

action a

    (input: int first_parm,
```

```
    input: int second_parm,  
    input: instance<C> ic,  
    input: instance<D> id,  
    output: int third_parm)  
{  
    third_parm = first_parm + second_parm;  
    ic.third_slot = ic.first_slot + ic.second_slot;  
    id.nested.third_slot = id.nested.first_slot +  
        id.nested.second_slot;  
    succeed();  
}
```

CHAPTER 7

Writing Osets

As described earlier, an oset behaves like a list.



For more information, refer to Understanding PepperCode Data Types.

You can copy an oset with the assignment operator (=):

```
oset1 = oset2
```

The result of the copy is an independent list—changes to the new list do not effect the old list, and vice-versa. When an oset is “copied,” only the link list elements of the oset are copied, not the values of the oset itself.

You can also traverse a list with the foreach or while statement, described later in this section.



For more information and a description of foreach and while, refer to Writing Control Statements.

A list recognizes the operators and functions in this table. index must be an expression that evaluates to an integer. item evaluates to the type that the list contains.

Oset Operators and Functions

Message	Description	Return type
list.enqueue(item)	Places item at the end of the list.	
list.push(item)	Places item at the beginning of the list.	
list.push_ordered(item)	Add item to the list so that the list remains sorted in ascending order, assuming it was sorted to begin with.	
list.pop()	Delete the item at the beginning of the list.	
list.delete(item)	Deletes the first member of the list that matches item.	
list.delete_first()	Delete the head item.	

Message	Description	Return type
<code>list.delete_last()</code>	Delete the tail item.	
<code>list.nth(index)</code>	Returns the value of the nth element of the list. index is an integer; the list is numbered starting with zero.	
<code>list.set_nth()</code>	Sets the value of the nth element of the list.	
<code>list.first()</code>	Returns the value of the first element of the list.	
<code>list.last()</code>	Returns the value of the last element of the list.	
<code>list.empty()</code>	If the list is empty, returns 1; otherwise, returns 0.	integer
<code>list.length()</code>	Returns the number of elements in the list.	integer
<code>list.flush()</code>	Causes the list to become empty.	

The following example illustrates the manipulation of lists. Notice that the local parameter `float_list` is created by default as a completely legal, but empty list; no initialization is required.

```

cpp_function void PRINTF(string) "printf";

// Print the list

action print_float_list

    (input: oset[float] float_list,

     no_context:)

{

    if (float_list.empty())

        PRINTF("List is empty\n");

    else

    {

        PRINTF("List contains: ");

        foreach item in float_list

            PRINTF(" %g", item);

        PRINTF("\n");

    }

    succeed();

}

// Test various operations on a list

```

```

action play_with_list

    (local: oset[float] float_list)

{
    execute print_float_list(:float_list float_list);

    float_list.push(5.0);

    float_list.push(6.0);

    float_list.push(7.0);

    execute print_float_list(:float_list float_list);

    float_list.enqueue(8.0);

    execute print_float_list(:float_list float_list);

    float_list.delete(5.0);

    execute print_float_list(:float_list float_list);

    float_list.delete(143.0);

    execute print_float_list(:float_list float_list);

    PRINTF("Value of element number 2 is %g\n", float_list.nth(2));

    PRINTF("Value of first element is %g\n", float_list.first());

    PRINTF("Value of last element is %g\n", float_list.last());

    PRINTF("Length of list is %ld\n", float_list.length());

    float_list.flush();

    PRINTF("Length of list is %ld\n", float_list.length());

    succeed();
}

```



For more information about the process you need to run this code, refer to [Compiling And Linking PepperCode](#).

Link the code to an existing Planning product, which creates an executable program called standalone. If you run this particular product with the command-line option -I, it prompts on the keyboard for input to the Action Interpreter. Invoke `play_with_list()` and then type `:exit` to leave the Action Interpreter:

```
shell> ./standalone -I

Checking ResponseAgent configuration.....Done Initializing Runtime Object
System...

...Done

Setting app name to 'standalone'

Initializing standalone...

Initializing communication buffer and hash table...MJD...Done

Creating slot classes.....Done

Creating slot specifier classes.....Done Initializing Schedule.....Done

Creating the Base Class.....Done

Creating form classes.....Done

...Done

Entering interpreter mode...

Enter an action call: play_with_list()

List is empty

List contains: 7 6 5

List contains: 7 6 5 8

List contains: 7 6 8

List contains: 7 6 8

Value of element number 2 is 8

Value of first element is 7

Value of last element is 8

Length of list is 3

Length of list is 0

Result: ( :RESULT 3 )

Enter an action call: :exit

...Done.

shell>
```


Writing Osets with Action Parameters

When using osets, remember that an oset is copied every time it is assigned to an object slot or action parameter. For this reason, it may be more efficient to store large osets on an object, and then pass the object from action to action. Here are some examples of when osets are copied:

```

action transform_objects

    (input: oset[instance<Spl_Class>] old_objects,

     output: oset[instance<Spl_Class>] new_objects)

...

    // my_objects is copied

    execute transform_objects(:old_objects my_objects);

    // new_objects is copied

    local_parameter = transform_objects.new_objects;

...

    local: oset[int] temp_scores,

    local: oset[int] real_scores,

...

    temp_scores = real_scores;           // real_scores is copied

    temp_scores.enqueue(99);

    real_scores = temp_scores;           // temp_scores is copied

    object.scores = real_scores;         // real_scores is copied

...

```

Writing Osets in Loops

Because osets are implemented as linked lists, accessing the *n*th element of an oset requires a linear traversal of the list until the *n*th element is accessed. If this occurs inside a loop, the linear traversal of the list may become expensive. The following code is $O(n^2)$ in computational complexity and will perform badly if *n* is large:

```

action poor_oset_use

    (input: oset[instance<foo>] the_list,

     local: instance<foo> item,

     local int index,

```

```

        local: int len)
    {
        len = the_list.length();
        index = 0;
        while (index < len) {
            // Get the nth item in the list.
            // This causes a linear traversal of the list.
            item = the_list.nth(index);
            // Do what ever you need to do to the list.
            my_function (item);
            index = index + 1;
        }
    }

```

The foreach construct is designed to iterate over every element in an oset—without incurring the overhead of a linear lookup to find each element of the set. The following code will execute with $O(n)$ complexity on the list:

```

action better_list_iteration
    (input: oset[instance<foo>] the_list)
    {
        foreach item in the_list {
            my_function (item);
        }
    }

```

This code is also much simpler, and clearer in the behavior that it exhibits.

Writing Osets with the foreach Statement

The foreach statement will loop over every item in an oset. The only way to stop the foreach is to use BREAK. For example, the following code will exit the foreach statement when the value 100 is found in an oset of integers:

```

...
local: oset[int] scores,      // (67 23 5 1 7 55 100 99 33 25 30 2 1)

```

```
...  
  
foreach score in scores {  
    if (score == 100) {  
        perfect_score_found = 1;  
        BREAK;          // stop looping, you found a perfect score  
    }  
}  
  
...
```


CHAPTER 8

Writing Arrays

This section describes arrays for PeopleCode.

Writing Associative Arrays

Here is the syntax for an associative array.

```
<element_type> [ <index_type> ] <id>
```

This declares an associative array (that is, a hash table) whose elements have the data type `<element_type>`, and whose index or key is any expression having the data type `<index_type>`. The index type can be any of these types:

- `int`
- `string`
- `float`
- `instance` (implicitly `instance<Base_Class>`)
- `class` (implicitly `class<Base_Class>`)
- `date`
- `time`

The element type can be any of these:

- `int`
- `string`
- `float`
- `instance<some_particular_class>`
- `class<some_particular_class>`
- `date`
- `time`

- <another array>

For example, "local: int[string] x" declares a local variable named "x" which is an associative array of integers, indexed by keys which are strings.

An associative array is "sparse": indexes need not be integers, and even if they are integers, they need not be consecutive. For example, here is an associative array with two elements whose indices (keys) are "a" and "z":

```
local: float[string] indexed_by_string...

indexed_by_string["a"] = 1.5;

indexed_by_string["z"] = 25.6;
```

As another example, here is an associative array with two elements whose keys are 0 and 5 (elements having keys 1 through 4 simply do not exist):

```
local: float[int] indexed_by_int...

indexed_by_int[0] = 1.5;

indexed_by_int[5] = 25.6;
```

If you attempt to read an element using a nonexistent key, you get a default value with no error message. The default value for a particular element type is the same as the implicit default value for an action formal parameter of that type (an integer element is 0, a string element is nil, etc.) Using the arrays shown in the preceding examples, the following statement will print zeros (because the default value for a formal parameter of type float is zero):

```
PRINTF("%f %f", indexed_by_int[1], indexed_by_string["b"]);
```

In most cases, by generating a default value instead of a runtime error, the PepperCode language causes programs to be more reliable. But this does place on the programmer the burden of detecting the situation where, due to some error, a particular key doesn't exist. The straightforward method is to use a function called "exists":

```
if (indexed_by_string.exists("b"))

    execute some_action(:some_parameter indexed_by_string["b"]);

else {

    exit_msg = NLSPRINT("Element at index 'b' doesn't exist");

    fail();

}
```

However, the preceding example looks up the key in the array twice (once for the "exists" function and once for the array indexing itself), and that's a relatively expensive operation. If you know that the value 0.0 can never legally appear in the array, then it's cheaper to test for that:

```
temp = indexed_by_string["b"];
```

```

if (0.0 != temp)

    execute some_action(:some_parameter temp);

else {

    exit_msg = NLSPRINT("Bad/nonexistent element at index 'b'");

    fail();

}

```

Even if the value 0.0 might legally appear in the array, it's cheaper to test for that first, and to call "exists" only in the case where you need to distinguish a zero value from a nonexistent entry:

```

temp = indexed_by_string["b"];

// Short-circuit || skips the "exists" unless temp is zero

if (0.0 != temp || indexed_by_string.exists("b"))

    execute some_action(:some_parameter temp);

else {

    exit_msg = NLSPRINT("Bad/nonexistent element at index 'b'");

    fail();

}

```

Writing Nonassociative Arrays

Here is the syntax for a nonassociative array.

```
<element_type> [ ] <id>
```

This declares a nonassociative array (vector) whose elements have the data type <element_type>, and whose index is a nonnegative integer. The array contains a series of contiguous elements whose indices range from zero to some upper bound. Initially the array is empty, but the upper bound automatically grows as needed when you store into it, or if you copy another array to it.

The legal element types are the same as the ones allowed for associative arrays.

The initial length of a nonassociative array is zero. The first assignment to the array changes the upper bound to be the index of that element. Any elements below the upper bound which haven't yet been assigned to will have a default value which is the same as the implicit default value for an action formal parameter of that type (zero for int, nil for instance, etc.)

After that, any assignment to an element using an index greater than the upper bound raises the upper bound and increases the length of the array.

For example, "local: float[] y" declares a local variable named "y" which is a nonassociative array of floating-point numbers. If you assign the following values to it:

```

y[3] = 30;

y[5] = 50;

```

then elements `y[3]` and `y[5]` come into existence with the values 30 and 50; to keep the array elements contiguous, `y[0]`, `y[1]`, `y[2]`, and `y[4]` also come into existence and have the default value zero.

All elements between zero and the current upper bound are considered to exist, even if no value has been assigned to them yet. In fact, they do occupy space in memory. This is a significant difference between nonassociative arrays and associative arrays. In the following example, the `PRINTF` statement will say "1 26001":

```

local: string[int] associative,

local: string[] vector

...

associative[26000] = "abc";

vector[26000] = "abc";

PRINTF("%d %d", associative.length(), vector.length());

```

If you use an index outside the bounds of the vector to read an element, that doesn't cause an error or enlarge the array; instead, it gives you the default value. You may use either the "exists" function or the "length" function to decide whether an index is out of bounds.

Using the vector "y" shown a few paragraphs ago, reading either `y[4]` or `y[6]` will yield the value 0.0, because array elements which haven't been set always return the default value. However, `y[4]` exists but `y[6]` doesn't, because `y[4]` is below the upper bound but `y[6]` is above it:

```

PRINTF("%f %f\n%d %d", y[4], y[6], y.exists(4), y.exists(6));

0.0      0.0

1        0

```

Assigning to `y[6]` raises the upper bound of the array and causes that element to come into existence (even if the value you assign to it is zero):

```

y[6] = 0.0;

PRINTF("%f %f\n%d %d", y[4], y[6], y.exists(4), y.exists(6));

0.0 0.0

1      1

```

There is an expense associated with enlarging an existing array. Thus, if you know at the outset what size the array will eventually need to be, it's fastest to assign to the highest-index element at the outset, so that the array immediately grows to the desired size. If you assign to the elements in order beginning with index 0 and continuing up to `n`, the time cost of memory allocation will be greater by roughly a factor of $\log_2(n)$ than if you had assigned to index `n` initially. Therefore,

even if you don't know at the outset what value to put into element *n*, it is well worth assigning some arbitrary value using that index, and then assigning the correct value later on. After you have established the size of the array, you can access elements within bounds in any order without incurring any software-imposed time penalty. (Of course, hardware mechanisms like the processor cache or the demand-paging system may favor one access pattern over another.)

Understanding Array operations

Arrays are similar to osets in their ability to store an arbitrary collection of elements, and certain operations apply to either data type. For example, "flush()" removes all elements of an array just as it removes all elements of an oset. Because osets existed first, and used "member function" notation like "variable.flush()", arrays use the same style of function notation.

Array Operators and Functions

Function	Description
delete(i)	Remove from an associative array the element having index (key) <i>i</i> . If the key did not exist, returns 0; otherwise returns 1. For a nonassociative array, this causes a compilation error.
exists(i)	Return 1 if the element having index <i>i</i> exists, else 0. For a nonassociative array, this tests whether the index is in bounds.
flush()	Make the array be empty.
length()	Return the number of elements in the array. For a multidimensional array, this counts only the number of subarrays, and does not walk the subarrays counting their elements recursively.
rlength()	Like "length()", but walks subarrays recursively and counts the number of leaf elements.
empty()	Returns nonzero if length() is zero.

Writing Arrays of Arrays

PepperCode arrays are one-dimensional, but because the element type of an array can be another array, they provide approximately the effect of multi-dimensional arrays.

For example, the following declaration creates approximately the effect of a two-dimensional array whose first dimension is indexed by int, whose second dimension is indexed by string, and whose element type is float.

```
local: float[int][string] a
```

More precisely, that declaration creates an associative array whose index is int and whose element type is array; each of those elements is an associative array whose index is string and whose element type is float.

A multidimensional array can mix associative and nonassociative arrays. For example, this is an associative array indexed by an integer, each of whose elements is a nonassociative array whose element type is string:

```
local: string [int] [] b
```

There are some subtle differences between a PepperCode array of arrays and a true multidimensional array in a language like Fortran or Pascal. A multidimensional array is a perfect rectangle: each row has the same number of columns. But a PepperCode array of arrays is sparse, and can have an irregular shape. Suppose that you make the following assignments to associative array "a":

```
a[0] ["a"] = 27.1;
```

```
a[0] ["b"] = 28.2;
```

```
a[1] ["w"] = 29.3;
```

Now a[0] (let's call this the first "row") has two elements ("columns"), while a[1] (the second "row") has only one. And the "column" keys for the second row are different than either of the column keys for the first row.

Even if the arrays are nonassociative rather than associative, the result may not be rectangular; in the following example, the first row once again has two elements while the second row has one:

```
local: int [] [] d2
```

```
d2[0][0] = 1;
```

```
d2[0][1] = 2;
```

```
d2[1][0] = 3;
```

The non-rectangular property makes PepperCode arrays useful for storing irregular data. For example, if you want to use three indices "nation", "state", and "city" to access an integer representing population, you can declare an associative array of arrays of arrays where each index has type string. Because a large state like California has many more cities than a small state like Delaware, a rectangular array dimensioned to suit California would waste considerable unused space for Delaware. Because PepperCode arrays are not constrained to be rectangular, they don't waste space in that fashion:

```
local: int[string][string][string] population
```

```
...
```

```
population["usa"]["california"]["pleasanton"] = x;
```

```
...
```

```
population["usa"]["delaware"]["dover"] = y;
```

```
...
```

Another implication of the "array of arrays" model is that operator functions like "exists" and "delete" apply to a particular, one-dimensional array. Here is a function which deletes an element of the outer array (in other words, it deletes an entire inner array):

```
population["usa"].delete("kansas")
```

And here is a function which deletes a single element of an inner array:

```
population["usa"]["california"].delete("san mateo")
```

Here is a function which checks the existence of one element of the outer array:

```
population["usa"].exists("nevada")
```

And here is a function which checks the existence of one element of an inner array:

```
population["usa"]["nevada"].exists("reno")
```

Of course, you can also apply a function to the outermost level of the array of arrays. The following example tells how many states appear within the "usa" array:

```
population["usa"].length()
```

The "rlength" function travels the array recursively and counts leaf elements. Here we count the number of cities in California, in the entire US, and in the entire world:

```
population["usa"]["california"].rlength() // length() would work too
```

```
population["usa"].rlength()    population.rlength()
```

It is not necessary to check whether a particular array exists before applying a function to it: if an array doesn't exist, then it behaves like an empty array. Thus, the second statement in the following example quietly returns zero even though the array corresponding to "oregon" no longer exists:

```
population["usa"].delete("oregon")
```

```
population["usa"]["oregon"].length()
```

If you want to have a nonassociative rectangular array, you can do so by using a technique often employed in C programs. (Although C provides multidimensional array declarations, their usefulness is limited because the dimensions are not maintained as variables at runtime and are not included with the array when you pass the array as an argument to a function.) This technique declares a one dimensional array and treats it as a two-dimensional array by performing the index computation explicitly. Here is an example:

```
input: int[] x, input: int rows, input: int columns
```

```
...
```

```
x[i*columns+j] = 10;    // Assign to x[i][j]
```

Writing Statements Involving Arrays

You can use a "foreach" statement to traverse an array, and you can use an assignment statement to copy an array. See the section on the "foreach" statement for an example.

A "foreach" statement guarantees to traverse arrays in an order which, although unspecified, will be the same on every target machine. (In this implementation, when you use "instance" as the index type for an array, we actually use the UID of the instance rather than the address of the instance, to avoid machine dependence.) You may not use the "in reverse" form for arrays.

(In this implementation, if you traverse two identically typed associative arrays with "foreach" loops, the sequence of indices will be the same for both arrays if you satisfy two requirements: [1] the set of indices now present must be the same and [2] the maximum size the array has ever had must be the same. Requirement [1] probably doesn't surprise you, but maybe requirement [2] does. The reason for [2] is that we traverse the hash buckets in order instead of using additional memory to remember the order of insertion of the elements. We also automatically increase the number of hash buckets to maintain good performance as the array grows, which forces us to alter the function which maps indices onto hash buckets, and to redistribute the elements among the buckets. But we never reduce the number of buckets when you delete. Maybe that's a bug; if we did so, then we could eliminate requirement [2].)

An assignment statement makes a complete copy of an array. The right side and left side arrays must have exactly the same index and element types.

Writing Array Accesses

To access an array as an lvalue or rvalue, write its name followed by a list of keys or indices, each enclosed in square brackets:

```
a[5] ["blue"] [37.2] = a[5] ["red"] [37.2];
```

Writing Arrays Indexed by Float

Because floating-point numbers are subject to roundoff error, it would be easy to construct a situation where a program would fail to find an element within an associative array, or would enter a second element with a slightly different key instead of replacing an existing element.

To avoid this, if the index type of an array is "float", PepperCode discards the low order bits of any value used to access such an array. This is similar to the "epsilon" feature used in floating point comparisons, but is implemented differently and is not governed by the "SET_EPSILON" function.

CHAPTER 9

Understanding Histories And Side Effects

A history is a data structure that represents a value that varies over schedule time. *Schedule time* is the time over which the scheduling system is making decisions (not wall clock time). Since schedules are projecting actions into the future, and these actions can have effects on the value of many variables in the schedule, a data structure is required to record these changes.

The history data structure has a rich programmatic interface for manipulating values and finding relevant information about the values of a history over time. This data structure is used to implement projected on-hand inventory balances, capacity availability, machine state, and any other time-varying variable.



For more information and a listing of the functions used for the history interface, refer to Using History Functions.



Note: The history sections in this section contain several code examples. These example are written in a pseudo-code similar to C++. They are not written in PepperCode.

Understanding the History Abstract Data Structure

A history has the following properties:

- It has a value at all points in time.
- It has a single value at any point in time.
- The value at any point in time is a function of the effects that persist at that point in time.

A simple interface to a history object can be the following two functions:

- `GetValue (History, time)` , which returns a history value at the given time.
- `SetValue (History, time, value)` , which sets the history value at the given time.

These two functions could be used to implement the entire history facility, although that would be inefficient.



Note: The functions `GetValue` and `SetValue`, as was stated earlier, are written in pseudo-code. They are used here for example only. The real history functions are listed in `Using History Functions`.

The following line gets the value of a history at a given point in time.

```
GetValue(Part_A_History, "1/2/96 3:48:00 PM")
```

The following line initializes a history to have all zero values.

```
foreach time from -infinity to infinity{
    SetValue (Part_A_History, time, 0)
}
```

This could be composed into a function:

```
History CreateHistory (int initial_value)
{
    History the_history = new History;
    foreach time from -infinity to infinity {
        SetValue (the_history, time, initial_value);
    }
    return the_history;
}
```

Note that this code would run forever.

Representing Availability of a Capacity Resource

Histories can be used to represent concepts such as the availability of a capacity resource.

Suppose you have a manufacturing work center that has two drills in it. You want to know how many drills are UNASSIGNED to production tasks at any point in time. If you ever assign more than two drills, you want to know about that so you can signal a violation. To do this you create a history to represent the availability of drills in the drill work center (DrillWC).

Histories are written as a list of history elements. Each history element has a time interval and a value. A history interval is shown as ((start . end) value). A history is a list of history elements such as (((start_time1 . end_time1) value1) ((end_time1 . end_time2) value2)). Adjacent history elements must meet; that is, the end of the prior history is equal to the start of the following element. This ensures that all points in time have a value.

To assign an initial value to our history, you could use the function discussed earlier, `CreateHistory`.

```
DrillWC = CreateHistory (2);
```

This sets DrillWC to the following history:

```
(((-infinity . infinity) 2))
```

Now suppose a task is scheduled to use a single drill from DrillWC from time 10 to time 20. This will result in only a single drill being available to be assigned from 10 to 20. The history that represents this state is as follows:

```
(((-infinity . 10) 2) ((10 . 20) 1) ((20 . infinity) 2))
```

There are two drills available up to time 10, then from time 10 to time 20 there is one drill available, and then two again from 20 to the end of time.

Now suppose a second task from time 15 to time 25 uses two drills. The history would look like the following:

```
(((-infinity . 10) 2) ((10 . 15) 1) ((15 . 20) -1) ((20 . 25) 0) ((25 . infinity) 2))
```

Task	Start	End	Drills used
T1	10	20	1
T2	15	25	2

From 0 to 10, no one uses the drill, so two are available.

From time 10 to 15, only task T1 uses a drill, so one drill is available.

From time 15 to 20 both T1 and T2 use the drill, for three total, leaving -1 drills available.

From time 20 to time 25, only T2 uses the DrillWC, so zero drills are available, and after 25 both drills are available.

Given this data structure and the accessors, you can write functions that get useful data, such as getting every time where this history is over allocated, or getting the next over-allocation after a given time. For example:

```
boolean Overallocated (History the_history)
{
    foreach time from -infinity to infinity {
        if (GetValue (the_history, time) < 0) return TRUE;
    }
    return FALSE;
}

time GetNextOverallocation (History the_history, time start_time)
```

```

{
    foreach time from start_time to infinity {
        if (GetValue (the_History, time) < 0) return time;
    }

    return NEVER;    // Special key that means no time found.
};

```

Understanding The History Data Structure

The current implementation of histories is a linked list of temporally ordered history elements. As done before in this section, pseudo code is used to discuss the data structure, but the actual implementation is in C++ using Lists, and a number of other complications.

Understanding History Data Structure Elements

Each history element has four slots:

- the start time of the interval,
- the end time of the interval,
- the value for that interval,
- and a list of the side effect objects responsible for the value.

The list is known as the changers list. The end time of the preceding element must be equal to the start time of the following element. This means that the intervals meet. Every history must cover each point in time between and including the `beginning_of_time` fence to the `end_of_time` fence.

Understanding A History Elements List

The list of history elements is a doubly linked list, so it can be traversed in order of either increasing time or decreasing time.

Because histories are implemented as a doubly linked list, you have some known performance characteristics for the base accessor functions. Both `GetValue()`, and `SetValue()` have linear— $O(n)$ —speed, where n is the number of distinct history elements generated by the entire set of changers on the history. Clearly as n becomes large, you expect the performance of histories to decrease.

When history lookup becomes a bottleneck in the performance of the system, the history functions will be reimplemented to use a different data structure to index times to values. One alternative is to use an index similar to ISAM that quickly points a time into roughly the correct area of the history, and then perform a linear search for the correct time point. This leaves the

performance at $O(m)$, where $m = n/\text{constant}$. Another possible implementation is to make the history elements form a balanced tree, and thus have a better performance.

Understanding An Example of History Object

The History object in C is defined as:

```
class History {
    List<Interval> rep;

    HE_Type type;

};
```

This example says that a history is implemented as a class with two member variables. The type member variable determines the data type of the value of each element. The `List<Interval>` rep member is the list of history elements. This doubly-linked list operates with a series of member functions for controlling the current item in the list, and for moving the current pointer forward and backward through the list, as well as for inserting and deleting elements. The list data type is used to implement Osets and histories, as well as many other items in the substrate.

Understanding An Example of Interval Implementation

The implementation of Interval is complicated by the fact that different histories can have different data types. The following example show the Interval implementation for a double data type. The actual implementation involves a union of multiple records with multiple data types. An Interval structure for a double data type looks like the following example:

```
struct Interval {
    Time start;

    Time end;

    ListVoid changers;

    double value;

};
```

Understanding GetValue Implementation

The following example is the implementation of GetValue for the history. Note that some of the additional functions and code are due to the union of the multiple data types.

```
RPS_FLOAT History_get_value_double(void* h, RPS_DATE dt)
{
    History* history = (History*)h;
```

```

    Interval *he;

    assert(history->get_type() == HE_Double);

    history->find(dt);

    he = history->current();

    if (!he) return 0.0;

    return HistoryElement(he).d->value;

}

```

To examine this example line by line:

The first line creates a local History variable, and casts the history passed in.

```
History* history = (History*)h;
```

The second line declares a local interval pointer.

```
Interval *he;
```

The next line is an error check to make sure the history passed in has a data type assigned.

```
assert(history->get_type() == HE_Double);
```

The next line uses the find method on the history to set the current pointer in the history to point to the HistoryElement that contains the time point dt.

```
history->find(dt);
```

The next line retrieves the interval for the current HistoryElement.

```
he = history->current();
```

If no element is found then return 0.0 as the value. This value could also represent an error.

```
if (!he) return 0.0;
```

The next line uses the historyElement to look up the value through a union that composes all the different types of History elements into a single structure. This value is the one returned to the caller.

```
return HistoryElement(he).d->value;
```

Finding Maximum

GetValue was a simple history accessor. Following is a more complicated example that finds the quantity of the maximum over allocation that occurs on a history between two time points.

```
// What is the maximum value that the history is over allocated by from
```

```

// start_time to end_time

//

CPP_FLOAT cpp_max_quantity_overallocated (void *history, CPP_DATE start_time,
CPP_DATE end_time)

{
    History *the_history = (History*)history;

    List<Interval> lhe;

    Interval *he;

    CPP_FLOAT max_overallocated = 0.0;

    assert(the_history->get_type() == HE_Double);

    if (start_time > end_time)

        // This should probably print an error message

        return 0;

    the_history->get_list(lhe);

    lhe.reset(BEGIN);

    he = lhe.current();

    while (he) {

        // begin checking the quantity in a "legal time period"

        if (he->end > start_time) {

            // staring right at special case

            if ((start_time == end_time) && (he->start == end_time)) {

                max_overallocated = HistoryElement(he).d->value;

                return rps_abs_d(max_overallocated);

            }

            if (he->start > end_time) // terminate

                return rps_abs_d(max_overallocated);

            if (rps_lt_d(HistoryElement(he).d->value, max_overallocated))

                max_overallocated = HistoryElement(he).d->value;

        }

        he = lhe.next();
    }
}

```

```
    }  
  
    return rps_abs_d(max_overallocated);  
}
```

Understanding Side Effects and Persistence

This section discusses how histories are updated from values that change inside PepperCode. What happens is that a mechanism called a “side effect” watches slot values, and when the values change, the side effect updates other things, such as histories.

Histories are data structures that contain computed information, based on the values assigned to schedule tasks.

What exactly is the processing that these supplies and constraints have on histories, and what are the differences caused by inventory/capacity and consume/supply dimensions? How are the history values maintained? This section addresses these questions.

Understanding The Effect of Supply/Constraint and Capacity/Inventory on Side Effects

A side effect computes a dependent variable from a set of independent variables any time one of the independent variables change. This structure is implemented in the substrate of the PepperCode system. The performance ramifications are very significant, and can be costly. Every resource supply and constraint has a side effect with independent variables of the start_time, end_time, quantity, and selected_object, and a dependent variable of the history. The following processing takes place when an independent variable changes:

- every side effect attached to that variable has its retract function fired.
- the slot value is changed.
- all the side effects are asserted.

Remember the performance ramifications. For a reasonable-sized BOM, moving a task and setting its start time and end time slots are relatively expensive operations.

There are two different dimensions of the differences which affect the behavior of side effects: resource supply vs. resource constraint, and capacity vs. inventory.

The resource supply vs. resource constraint dimension has to do with how to take a value—the quantity—and combine it with the existing value of the interval. For resource constraints, subtract the value from the history value. For resource supply, add the value to the interval value. This combination function can get more complex than addition/subtraction. For histories that represent time-varying states of an object, the combination function sets the state to the state of the latest effect in time.

The capacity vs. inventory dimension has to do with persistence of the effect, or how long the effect lasts. For a capacity resource, the effect of the usage of a resource persists from the start_time to the end_time of the task; then the resource is given back to the pool, and is available for other uses. For inventory parts, the usage persists forever. That is, once a part is used, it is never given back to the pool for some one else to use. This difference is encoded by how the side effect determines the start time and end time for the persistence of the side effect. For a capacity side effect, the side effect code looks up the start and end of the task, and uses these as the start and end of the side effect. For an inventory constraint, the side effect looks up the start time of the task for the start of the side effect, and uses infinity for the end time. A resource supply inventory side effect uses the end time of the task as the start time of the side effect, and infinity as the end time of the side effect.

Understanding the Scheduling Classes: Resource and Task

The two PepperCode classes involved with scheduling are resource and task.

Understanding the Resource Class

In PepperCode, there is an object called a Resource. This object is the base class for Equipment_Resource, and for Inventory_Resource. This means all resources share the same basic structure for manipulating the time varying portion of the behavior. Notice there are two history slots on this object. The initial_history slot is a history that can be used to determine how much was available for each time period when the resource was created. For a capacity resource this represents the capacity available for the resource. In any PepperCode function, the resource_history slot is typically passed as the history value.

```
class Resource : Named_Object Spl_Class {

    oset[instance<Resource_Constraint>] resource_constraints

    oset[instance<Resource_Supply>] resource_supplies

    float initial_amount

    enum<Relevant_Status> relevant_status

    history<float> initial_history

    history<float> resource_history

    action<substitute> substitute_action

    action<resource_batch> resource_batch_action

    action<calculate_duration> calculate_duration_action

    action<calculate_quantity> calculate_quantity_action

    action<consumable> consumable_action

};
```

Understanding Tasks

In PepperCode, tasks look roughly like the following:

```
class Base_Task : Named_Object Spl_Class {

    instance<Base_Task> parent

    date start_time

    date end_time

    enum<Task_Status> status

};

class Duration_Task : Base_Task {

    oset[instance<Resource_Constraint>] resource_constraints

    oset[instance<Resource_Supply>] resource_supplies

};
```

Duration_task is the basic scheduled task. It is scheduled to run from start_time to end_time, and has a list of the resource_constraints associated with the task, and a list of resource supplies associated with the task.

Understanding Resource Supplies and Constraints

A resource constraint represents a usage of either a inventory resource or a capacity resource by the task. A resource supply represent a supply of either a inventory or capacity resource. Both resource_constraints and resource_supplies have effects on the history of the object they effect, but only resource_constraints are constraints in the search engine.

The definition of the resource_constraints (usages of a resource by the task) and resource_supply (supply of a resource) are shown following. These are fairly straightforward.

```
class Constraint : Spl_Class {

    float weight // instance weight

    float class_weight // class weight

    action<penalty> penalty_action

    instance<Spl_Class> object

};

class Repairable_Constraint : Constraint {

    enum<Relevant_Status> relevant_status

    action<start_and_end> start_and_end_action
```

```

    action<constraint_repair> repair_action

    action<repair_earliest> repair_earliest_action

    action<earliest_violated> earliest_violated_action

    int start_fall_earlier
};

class Resource_Constraint : Repairable_Constraint {

    int created_by_user

    instance<Duration_Task> object

    class<Resource> selected_object_class

    instance<Resource> selected_object

    float quantity

    action<duration_constraint> duration_constraint_action

    action<inventory_constraint> inventory_constraint_action

};

class Resource_Supply : Spl_Class {

    instance<Duration_Task> object

    instance<Resource> resource

    float quantity

};

```

Understanding the Effect of Resource Supplies and Constraints on Histories

Refer to the Resource_Constraint class in the previous section. The resource_constraint object has the following slots:

- a object slot which holds the task that this resource constraint is associated with. This allows the resource_constraint to get to the start_time, and end_time of the scheduled duration_task.
- a quantity slot, which is the amount of the resource that this task needs to use.
- a selected_object slot that holds the Resource object that will be affected.

These slots allow access to all the required information needed to call these history functions:

- a function that takes (start_time, end_time, quantity, and a history) and updates the history to reflect a usage of quantity from start time to end time.
- a function that updates the history to reflect the *removal* of a usage of quantity from start_time

to end_time.

The two functions in the last two bullets are opposites; they serve to assert and retract the side effect of a task from a history.

Programming for Side Effects: The side_effect Keyword

You should add the “side_effect” keyword to the declaration to a slot that is:

- An input or output to a side effect function.
- Any slot which lies along the path leading from the root object of a side effect subtree to an input or output slot.

Adding the “side_effect” keyword consumes memory. Aside from that, there is no harm done if you add on a slot which is not associated with side effects, but if you do not use it on a slot which is associated with side effects, a runtime error message will be printed on the server console.

Here is an example of using the side_effect keyword. It is taken from resource.spl.

```
class Resource_Supply : Spl_Class {
    instance<Duration_Task>  object
    instance<Resource>       resource
    float                    quantity
    float                    increment_amount // if duration, in seconds,
                                           // otherwise units
    int                      increment_type   // 0 - none, 1 - duration,
                                           // 2 - quantity
};

slot Resource_Supply.object { side_effect: };
slot Resource_Supply.quantity { side_effect: };
slot Resource_Supply.resource { side_effect: };
slot Resource_Supply.increment_amount {default: 0.0 class_slot: };
slot Resource_Supply.increment_type {default: 0 class_slot: };
slot Resource_Supply.init_action { default: resource_supply_init };
slot Resource_Supply.delete_action { default: delete_resource_supply };
```


Understanding Schedules

PeopleSoft Planning uses an in-memory representation to allow fast computation. There is no option of running a consolidation job to compute projected on-hand balances, next requirements, or any of the other batch processes in MRP systems. PeopleSoft Planning always computes the ramifications of scheduling an event.

A schedule is a set of inventory and capacity histories that projects the availability of material and capacity into the future, and a set of tasks that have been scheduled. The tasks change the projected values for on-hand material and capacity availability.

Any real world event represented in PeopleSoft Planning is a scheduled event in the in-memory model. The receipt of material based on a scheduled purchase order delivery is a real world event represented as a PO Delivery task in PeopleSoft Planning. This task is a part of the schedule. This task has an effect on the projected on hand balance of material for the part that is delivered. The change in the projected on hand quantity of the delivered material is calculated immediately when the PO delivery is added to the schedule. No consolidation or netting process is required.



For more information on how this technically happens in the software, refer to Understanding Side Effects and Persistence.

A production operation is a scheduled task. Its materials usage, capacity allocations, and materials supplied are all effects of scheduling this task. As soon as the production operation is added to the schedule, its effects are computed on available balances of material and capacity availability. A Sales Order is another real world event that represents the shipment of material from a location under PeopleSoft Planning control to a customer. This is represented as a shipment of material, and changes the projected on hand balance in the future.

A change in a BOM's effectivity is not a real world event, and so does not directly affect the schedule. It does indirectly affect the schedule by determining the appropriate BOM's to add to a work order, but the resulting mapped materials requirements with the work order operations are what affect the schedule, not the actual BOM, or its effectivity. Effectivity is a set of dates on the data, such as the start and end date, that specify when this data is true. For example, if the effectivity for a part number is today until next Friday, the part number is good until next Friday.

The ramifications of this model of a schedule are not immediately apparent, but they effect every aspect of thinking about the schedule, or about materials availability, or about the processes that are used to find information.

For example, there is no transaction to allocate material to an order. Since an order—be it a production order, or a sales order—is a scheduled task, its effects on materials availability and capacity are always asserted into the projected on-hand balance for inventory and availability for capacity. Allocation of material to priority orders is handled by scheduling orders such that they are predicted to have materials and capacity available when they are scheduled to execute. If insufficient material is available, and an allocation decision needs to be made, then the conflicting orders are rescheduled to a time point where the material is anticipated to be available.

Here is a scheduling example.

Part A has five on hand now. Next Monday you project an order for five units to be delivered: you have a PO Delivery scheduled with a due date of next Monday, and a quantity of five. The history for Part A will project an on-hand balance of five up through next Monday, and then a projected balance of ten for the rest of time. At this point, our schedule consists of a single history for Part A, and one scheduled task for the delivery of the material.

Add two sales orders both due this Friday, both for five part A's. As soon as the Sales Orders are entered into the schedule, the effects of these orders are computed on projected on hand balances and capacity availability. Your schedule now consists of a history for Part A which shows a projected balance of five units from now till Friday, then -5 units on hand from Friday to Monday, and 0 units on hand from Monday till the end of time; and three tasks, and one PO Delivery on next Monday, and two sales order shipments on Friday. Notice that all that happened was that two sales orders were added. No allocation of either material process or inventory netting process was executed.

The schedule now has an Inventory violation. You project that there will be a shortage on Friday of five units. At this point you do not make an inventory allocation of material to an order. You decide which order to schedule on Friday, and which order to reschedule to Monday, Monday being when the material is projected to be available. This decision can be made in a number of ways, but order priority might be a good criteria.

Now our schedule consists of a projected on hand inventory for part A of five till Friday, then 0 from Friday till the end of time; and three tasks, a sales order shipment on Friday, a PO Delivery Monday morning, and a sales order shipment Monday morning, right after the material delivery. The inventory violation has been resolved by moving a task.

CHAPTER 10

Understanding Operators And Functions

This sections lists and describes how to use the PepperCode intrinsic operators and functions. It also describes how to access and use C/C++ functions.

Understanding Infix and Intrinsic Operators and Functions

There is a new inline notation type for mathematical operations in Release 8.0. It applies to the operators in the following table. However, we have maintained backward compatibility in that you can still use prefix notation.

Example:

You can now use the following expression (using inline notation):

```
i = i + 1;
```

In previous releases, you would have had to code this expression in prefix notation as follows:

```
i = ADD(i,1);
```

The following table lists the new Infix notation operators and the data types with which they can be used.

Each operator allows a limited set of data types for its operands.

PepperCode Infix Notation

Operator	Left operand	Right operand	Result	Coerces?
 &&	int	int	int	N
<= >= < >	int	int	int	Y
	float	float		
	string	string		
	date	date		
	time	time		
== !=	int	int	int	Y
	float	float		
	string	string		
	instance	instance		
	class	class		
	date	date		
	time	time		
	action	action		
	enum	enum		
~	string	string	int	N
*	int	int	int	Y
	float	float	float	
	int	time	time	
	time	int	time	
/ %	int	int	int	Y
	float	float	float	
	time	int	time	
	time	time	int	
+	int	int	int	Y
	float	float	float	
	date	time	date	

Operator	Left operand	Right operand	Result	Coerces?
	time	date	date	
	time	time	time	
-	int	int	int	Y
	float	float	float	
	date	time	date	
	time	time	time	
!	none--unary	int	int	N
		date	int	
		time	int	
		string	int	
		instance	int	
		class	int	
+ -	none--unary	int	int	N
		float	float	
		time	time	
.	instance	slotname	any	N
	class	slotname		
	action	out-parmname		
	enum	slotname		
	oset	fcnname		
	array	fcnname		
()	cpp_function	arglist	any	N
	action		none	
	intrinsic fcn		any	
[]	array	int	any	N
		string		
		float		
		instance		

Operators `&&` and `||` perform short-circuit evaluation.

PepperCode has several intrinsic operators and functions derived from C++ operators and functions, as listed in this table. Most of these operators and functions take two arguments—`arg1` and `arg2`, in that order; they all return one value. True is 1 and false is zero. An argument can be an expression. For logical operations, inputs are TRUE if nonzero, and the output is 1 for TRUE and zero for FALSE.



Note: Most of the prefix functions have an equivalent in the Infix operators. The prefix operators are still included to maintain compatibility with any release prior to Release 8.0. When possible, use the infix operators.

PepperCode Prefix Operators and Functions

Operator or function	Description	Number of arguments	Argument data types	Return type
GT	Is <code>arg1</code> greater than <code>arg2</code> ?	2	int, float, time, date, string	int
GT_OR_EQ	Is <code>arg1</code> greater than or equal to <code>arg2</code> ?	2	int, float, time, date, string	int
LT	Is <code>arg1</code> less than <code>arg2</code> ?	2	int, float, time, date, string	int
LT_OR_EQ	Is <code>arg1</code> less than or equal to <code>arg2</code> ?	2	int, float, time, date, string	int
EQ	Is <code>arg1</code> equal to <code>arg2</code> ?	2	int, float, time, date, string, instance, class, enum	int
NE	Is <code>arg1</code> not equal to <code>arg2</code> ?	2	int, float, time, date, string, instance, enum	int
ADD	Add <code>arg1</code> to <code>arg2</code> .	2	int, float, time, date	Same as argument
SUB	Subtract <code>arg2</code> from <code>arg1</code> .	2	int, float, time, date	Same as argument
MUL	Multiply <code>arg1</code> by <code>arg2</code> .	2	int, float, time, date	Same as argument
DIV	Divide <code>arg1</code> by <code>arg2</code> .	2	int, float, time, date	Same as argument

Operator or function	Description	Number of arguments	Argument data types	Return type
AND	Logical AND arg1 and arg2.	2	int, instance, class	int
OR	Logical OR arg1 and arg2.	2	int, instance, class	int
NOT	Logical NOT arg.	1	int, time, date, instance, class	int
NIL	Return 1 if the argument is NIL; otherwise, return zero.	1	string, oset, instance, class, action	int
MIN	Return the minimum of two numbers.	2	int, float, time, date	Same as argument
MAX	Return the maximum of two numbers.	2	int, float, time, date	Same as argument
ABS	Return the absolute value—always positive—of arg.	1	int, float, time	Same as argument
MOD	Return the remainder of arg1 divided by arg2.	2	int, float, time, date	Same as argument
POW	Return arg1 raised to the power of arg2.	2	int, float	Same as argument
ROUND	Round to a float value.	1	float	float
ROUND_UP	Round a float value up. The ceiling is an increase to the next highest integer value.)	1	float	float
ROUND_DOWN	Round a float value down. The floor is a decrease to the next lowest integer value.)	1	float	float

Operator or function	Description	Number of arguments	Argument data types	Return type
INT_TO_FLOAT	Convert an integer to a float.	1	int, date, time	float
INT_TO_TIME	Convert an integer to a time value.	1	int	time
INT_TO_STRING	Convert an integer to a string.	1	int, time	string
FLOAT_TO_INT	Convert a float to an integer.	1	float	int
FLOAT_TO_STRING	Convert a float to a string.	1	float	string
STRING_TO_INT	Convert a string to an integer. cpp_ascii_to_int scheduler/utills/cpp_spl_misc.h	1	string	int
STRING_TO_FLOAT	Convert a string to a float.	1	string	float
RETRACT	Fires the retract method for each side effect associated with the slot passed in as an argument.	1	slot	int
ASSERT	Fires the assert method for each side effect associated with the slot passed in as an argument.	1	slot	int

If you invoke a function with arguments of two different types, the “lower priority” type gets converted to the “higher priority” type. The following list shows the priority of a data type, from highest to lowest:

- float
- date

- time
- int



Note: The RETRACT and ASSERT functions are meant for use in exceptional circumstances by programmers who understand fully the side effects mechanism. Normally the side effect methods are fired automatically as part of the act of assigning a value to a slot, so it is not necessary to invoke these functions. The compiler will issue an error if the argument which you pass to the function is not a slot.

Understanding SET_EPSILON and SET_FLOAT_FORMAT

Although most of the operators and functions listed in the Intrinsic Operators and Functions table are self-explanatory to an experienced C/C++ programmer, SET_EPSILON and SET_FLOAT_FORMAT require further explanation:

SET_EPSILON and SET_FLOAT_FORMAT

Function	# arguments, data types	Description
SET_EPSILON	1, int	<p>When you compare floating point values by using functions such as EQ, NE, LT_OR_EQ, or LT, PepperCode considers values to be equal if they differ by less than a tiny amount, called the epsilon. As a result, small round-off errors do not prevent numbers from being considered equal.</p> <p>This function sets the value of the epsilon to be approximately $1 \times 10^{(-n)}$, where n is the argument you pass to the function. If the numbers being compared are greater than 1.0, the system scales the epsilon upward by multiplying it by the sum of the numbers being compared.</p>
SET_FLOAT_FORMAT	1, string	<p>This function sets the format that the FLOAT_TO_STRING function uses when it converts a floating point number to human-readable form. The argument to SET_FLOAT_FORMAT is a string that must follow the rules for the C language function printf. For example, the format could be one of the following:</p>

Function	# arguments, data types	Description
		% 15.2f Create a 15-character string containing a number that has two digits to the right of the decimal point, but no exponent.
		% .5f Create a string—just large enough to represent the number—with five digits to the right of the decimal point, by no exponent.
		% .5e Create a 15-character string containing a number that has two digits to the right of the decimal point and that has an exponent.
		% 15g Use 15 digits for the string and use an exponent only if the number is too big or small to be represented without one.

Using EQ With Strings

Using EQ on strings is not a pointer comparison, it is a string compare (strcmp). The following statements do almost exactly the same thing in PepperCode:

```

...
input: string string_1,
input: string string_2,
...
if (EQ (string_1, string_1))    // really a strcmp
...
if (EQ (STRING_COMPARE (string_1, string_1), 0))
...

```

strcmp must check every character of one string against every corresponding character of another string until there is not a match. This can be expensive for strings that begin with the same set of characters.

Accessing C/C++ Functions

In addition to the PepperCode intrinsic operators and functions, you can declare and then use C/C++ operators and functions from within PepperCode code.

To declare a C or C++ function, use the following syntax:

```
cpp_function return_type SPL_FUNCTION_NAME (argument(s)) "cpp_function_name";
```

- `return_type` can be any PepperCode data type or void for C/C++ functions that don't return anything.
- `SPL_FUNCTION_NAME` is the name that PepperCode actions will use to reference the C/C++ function. This name should always be in all capital letters so it is easy to identify. Multiple `cpp_function` declarations may map different PepperCode identifiers to the same C++ function. However, multiple `cpp_function` declarations may not map the same PepperCode identifier to multiple C++ functions
- `argument(s)` is a list of PepperCode data types, separated by commas, for all arguments. The data types of the arguments supplied in this arglist must match with those specified in the C++ function. Also, the data types of function calls must match.
- `cpp_function_name` is the actual C/C++ function name, placed inside of double quotes (" ").

PepperCode Data Types in `cpp_function` Statements

As these scenarios illustrate, you must provide meaningful PepperCode types in the C++ function declaration.

The types provided in the C++ function declaration must correspond to the types in the actual C++ function. Starting in Release 8.0, the PepperCode compiler generates code that allows the linker to check the argument list data types.

Many of the Release 7.5 `cpp_function` declarations have already been modified to work in Release 8.0. For example, in Release 7.5, the `SET_SLOT_ACTION_ON_INSTANCE` function took `class<Spl_Class>` as the first argument. Since the name of the function specifies an instance, it obviously requires an instance as an argument instead of a class. So, the `cpp_function` declaration was changed to specify the data type `"instance<Spl_Class>"` as the first argument instead of the data type `"class<Spl_Class>."`

Release 8.0 Function Declaration:

```
cpp_function void SET_SLOT_ACTION_ON_INSTANCE (instance<Spl_Class>, string,
string) "cpp_set_slot_action_on_instance";
```

Release 7.5 Function Declaration:

```
cpp_function void SET_SLOT_ACTION_ON_INSTANCE (class<Spl_Class>, string,
string) "cpp_set_slot_action_on_instance";
```

Rules for Passing Arguments

The rules for passing arguments in Release 8.0 have not really changed, but argument types are checked in Release 8.0, and this may require some changes to the way that you declare and use C++ functions.

The arglist declares the formal arguments of the C++ function using PepperCode data types. Data types can be preceded by "const:".

Example:

```
cpp_function int STRING_COMPARE (const: string, const: string) "nlstrcmp";
```

Since argument types were not checked in previous releases, you could have probably gotten by with:

```
cpp_function int STRING_COMPARE (string, string) "nlstrcmp";
```

However, this will not work in Release 8.0.

You can also use the symbol "&".

Example:

```
cpp_function string SYSTEM_CALL(string, int &) "cpp_system_call";
```

which indicates pass-by-reference...

and the symbol "*".

Example:

```
cpp_function int LIST_FILES_IN_DIRECTORY(const:
string, oset [string], oset [string], string *) "ls_to_lists";
```

which indicates pass-by-pointer.

The default is pass-by-value. Only one level of "&" or "*" is currently allowed.

When you pass an actual argument to a formal argument which was declared with "*", the compiler implicitly takes the address of the actual argument.

The arglist may contain the argument "void" alone, or the arglist may contain no arguments. These have the same meaning.

Example:

```
cpp_function date GET_EARLY_FENCE() "LpExportToSPL_getEarlyFence";
```

If the arglist contains at least one argument which is not "void", it may end with "..." or ", ...". This denotes that the C++ function has a variable argument list.

Example:

```
cpp_function void LP_LOG(int, int, string, ...) "LpExportToSPL_lpLog";
```

Typedefs Used With C++ Functions

When writing C++ functions to be invoked from PepperCode, the programmer should use the typedefs which begin with "RPS_" in the table below. These "barrier" typedefs provide a small degree of insulation from changes in the underlying implementation. However, to do anything meaningful with the data inside the C++ function, you typically need to know what the underlying C++ declaration is, since mere typedefs (as opposed to classes with methods that operate on the data) don't allow that; so the table shows them as well.

<i>PepperCode</i>	<i>C++ typedefs</i>	<i>Underlying type</i>	<i>Comments</i>
Int	RPS_INT	32-bit int	[1]
Float	RPS_FLOAT	double	IEEE 64 bits
String	RPS_STRING	(char *)	[4]
Date	RPS_DATE	32-bit int	[2]
Time	RPS_TIME	32-bit int	[2]
void	RPS_VOID	(void)	[3]
instance<>	RPS_INSTANCE	(rtoc_instance_object *)	
class<>	RPS_CLASS	(rtoc_class_object *)	
action<>	RPS_ACTION	(struct spl_action_info *)	
oset[]	RPS_OSET	(ListVoid *)	[5]
enum<>	RPS_ENUM	(rtoc_instance_object *)	
history[]	RPS_HISTORY	(History *)	[5]
array	RPS_ARRAY	(spl_array *)	[5]

PepperCode and C++ typedefs

Notes:

1. For identical behavior on all machines, we intend that all PepperCode implementations perform 32-bit integer arithmetic on PepperCode type "int", no matter whether the word size of the underlying machine is 32 or 64 bits. The previous compiler used 64-bit arithmetic on 64-bit machines.
2. Due to loose programming practices in the past, we force PepperCode date and time to occupy the same amount of space as PepperCode int. C++ programmers should not assume this is equivalent to C++ "time_t". On some 64-bit machines (DEC Alpha OSF/1 Unix, SGI Irix) "time_t" uses 32 bits, but on others (HP-UX) it uses 64.
3. This type is used only to indicate that a function has no return value or no arguments. The "void *" type is not allowed.
4. The declaration "string *" maps to "char **". The declaration "string &" maps to "char *&". The declaration "const string" maps to "const char **", not "char *const". It is expected that the implementation will change to use some "class rps_string" type instead of "char *". At that point, the restrictions in [5] will apply.
5. The declarations "oset &" and "oset *" are not yet implemented. The C++ programmer should understand that assigning to "ListVoid *" does not invoke the copy constructor for ListVoid, and thus causes a memory leak. The same comments apply to "history" and to "array".

Using PepperCode Runtime Functions

This table lists most of the PepperCode runtime functions. There are also sections following this one that list PepperCode runtime that are grouped together under specific utilities, such as history functions.

In the 8.0 and later versions of PepperCode, it is no longer necessary to write a "#include" statement for a .h file when you declare a `cpp_function`. Instead, the SPL compiler generates the C++ declaration based on the `cpp_function` statement, and the linker will check that the declaration is consistent with the actual C++ definition. However, if any of the arguments to the function are const or pointers or references, you must use "const:", "*", and "&" in the declaration to indicate this. If the function takes a variable-length argument list, you must use "..." to indicate this.

An example of a declaration using the new syntax:

```
cpp_function void MYFUNC(int, const: float *, date & ...) "myfunc";
```

To call a class member function, you must use a non-member wrapper function.

When you pass an actual argument to a formal argument declared with "*", the compiler automatically takes the address of the argument for you (there is no "address of" operator in SPL comparable to the "&" operator in C++). In the table below, you do not have to #include the .h file in the C++ function Include file column.

For PepperCode before version 8.0, if you use the `cpp_function` statement to add a new function to the system, rather than to simply obtain access to an existing one, you must also create a .h file that contains the C or C++ function prototype declaration. You must use #include to include the .h file in the PepperCode source file that invokes the function. The PepperCode compiler simply passes the #include statement through to the C++ compiler when the included filename ends in .h. This file satisfies the requirement that the C++ compiler must see a prototype for each function it is asked to invoke. For these versions of PepperCode, in the table below, you have to #include the .h file in the C++ function Include file column.

PepperCode Runtime Functions

Function signature	Description	Arguments and returns	C++ function Include file
void CREATE_MULTIPLE_INHERITED_SUBCLASS (string, oset[string])	Creates a subclass of an existing set of parent classes.	Argument1: the name of the new subclass Argument2: an oset of parent class names	cpp_create_multiply_inherited_subclass scheduler/utls/cpp_spl_misc.h
String CREATE_NAME_FROM_OSET(oset[string], int)	Creates a name from an oset of strings.	Argument1: an oset of strings Argument2: an flag for specifying a unique name (1 = unique)	cpp_create_name_from_o_set scheduler/utls/cpp_spl_misc.h
instance<Spl_Class> CREATE_OBJECT(string, string)	Create an instance of an already existing class.	Argument1: the name of the instance you want to create Argument2: the name of the class it belongs to Returns: an instance of the class	cpp_create_object scheduler/utls/cpp_spl_misc.h
Void CREATE_SUBCLASS (string, string)	Creates a subclass of an existing parent class.	Argument1: the name of the new subclass Argument2: the name of the parent class	cpp_create_subclass scheduler/utls/cpp_spl_misc.h

Function signature	Description	Arguments and returns	C++ function Include file
Date CURRENT_TIME ()		Returns: the current system date	rps_get_current_time substrate/utillsCC/util.h
String DATE_TO_STRING (date)	Converts a date into a string with format "mm/dd/yy hh:mm:ss"	Argument1: a date	cpp_get_date_string scheduler/utills/cpp_spl_misc.h
Void DELETE_OBJECT (instance<Spl_Class>)	Deletes an instance. This function should only be used within a delete method.	Argument1: an instance	cpp_delete_object scheduler/utills/cpp_spl_misc.h
void DESCRIBE (instance<Spl_Class> , int)	Prints the value of every slot on an instance.	Argument1: an instance Argument2: a verbose flag (1 = verbose, 0 = not verbose)	rps_describe substrate/objectcore/rps.h
float EXP (float)	Calls the C exp function.	Argument1: a float Returns: the result of exp	C function exp PepperCode compiler includes .h file for you
int FLOAT_TO_INT (float)	Converts a float into an integer (the float is truncated).	Argument1: a float Returns: an integer	rps_float_to_int substrate/utillsSPL/cpp_math.h
String FLOAT_TO_STRING (float)	Converts a float into a string.		rps_float_to_string substrate/utillsSPL/cpp_math.h
Class<Spl_Class> GET_CLASS_BY_NAME (string)	Finds a class when given its name. Note: This is now obsolete. Use <i>name.class_name</i> instead.	Argument1: the name of a class Returns: a class (if found), or 0	

Function signature	Description	Arguments and returns	C++ function Include file
class<Spl_Class> GET_CLASS_OF_INSTANCE (instance<Spl_Class>)	Gets the class of an instance.	Argument1: an instance Returns: a class	cpp_get_class_of_instance scheduler/utils/cpp_spl_misc.h
Void GET_DESCENDANTS (oset[instance<Spl_Class>], class<Spl_Class>, int)	Finds all descendants of a class (including descendants of descendants). The descendants can be instances or classes. Note that a class is considered to be its own descendant. Normally GET_DESCENDANTS returns classes if the first argument is an oset of classes, or instances if the first argument is an oset of instances. For backward compatibility, it accepts a third argument of type integer which is 0 for classes and 1 for instances.	Argument1: an oset of instances or classes that will be “filled in” during the function execution. Argument2: a class Argument3: a flag for specifying instances or classes (1 = instances, 0 = classes)	cpp_get_descendants scheduler/utils/cpp_get_descendants.h
void GET_DIRECT_DESCENDANTS	Finds only the direct descendants of	Argument1: an oset of instances or classes that will be	cpp_get_direct_descendants scheduler/utils/cpp_spl_

Function signature	Description	Arguments and returns	C++ function Include file
(oset[instance<Spl_Class>], class<Spl_Class>, int)	a class. The descendants can be instances or classes.	“filled in” during the function execution. Argument2: a class Argument3: a flag for specifying instances or classes (1 = instances, 0 = classes)	misc.h
instance<Spl_Class> GET_INSTANCE_BY_NAME (string)	Finds a named instance when given its name.	Argument1: the name of an instance Returns: an instance (if found), or 0	cpp_get_instance_by_name scheduler/utls/cpp_spl_misc.h
int GET_MSG_LEVEL ()		Returns: the current message level	get_debug_level substrate/utlsCC/Error.h
string GET_NAME_OF_CLASS (class<Base_Class>)	Gets the name of a class. Not used in Release 8.0 or later. See GET_NAME_OF_CLASS below.	Argument1: a class Returns: a class name	rps_get_name_of_class substrate/objectcore/rps.h
instance<Base_Class> GET_NULL_INSTANCE ()	This is a shortcut for GET_INSTANCE_BY_NAME ("Null_Instance").	Returns: The Null_Instance object.	get_null_instance scheduler/utls/cpp_spl_globals.h
void GET_RANDOM_SEED (oset[int])	Copies the random seed into an existing oset of integers. Any existing integers in the oset will be flushed at execution	Argument1: an oset of integers	cpp_get_random_seed scheduler/utls/random.h

Function signature	Description	Arguments and returns	C++ function Include file
	time.		
instance<Spl_Class> GET_TYPED_INSTANCE_BY_NAME(string instance_name, string class_name)	Gets the named instance of that type.	Argument1: the name of the instance you are looking for. Argument2: the name of the class of that instance.	If the named instance of that type exists, returns the instance handle; otherwise, returns 0. cpp_get_typed_instance_by_name scheduler/spl/cpp_utility.spl
int INSTANCE_EXISTS_IN_LIST(instance<Spl_Class> , oset[instance<Spl_Class>])	Checks an oset to see if an instance (or class) is a member of the oset.	Argument1: an instance (or class) Argument2: an oset of instances (or classes) Returns: 1 if the instance (or class) is a member of the oset, 0 otherwise	cpp_instance_exists_in_list scheduler/utis/cpp_spl_misc.h
int LIST_FILES_IN_DIRECTORY (string, oset[string] subdirectories, oset[string] files)	Returns a list of subdirectories and files that are the immediate children of a given directory.	Argument1: the name of a directory. Argument2: an output giving a list of subdirectories which are immediate children of the Argument1 directory. Argument3: an output giving a list of subdirectories which are immediate children of the Argument1 directory. Returns: 0 if fails; nonzero otherwise. To read error messages on a non-zero return, use the following code: error =	list_directory cpp_io.h

Function signature	Description	Arguments and returns	C++ function Include file
		subdirectories.pop(); where error is a string, and subdirectories is Argument2.	
int MSG (int, string)	Prints a string (formatted for printf) when the current message level is greater than or equal to the first argument.	Argument1: message level needed to print Argument2: string (formatted for printf) Returns: 1 when the print was performed, 0 otherwise.	
int OBJECT_IS_ALIVE (instance<Spl_Class>)	Checks to see if an instance has been deleted. GET_DESCENDANTS and GET_DIRECT_DESCENDANTS automatically make this check.	Argument1: an instance Returns: 1 if the instance is not deleted, 0 otherwise	cpp_object_is_alive scheduler/utls/cpp_spl_misc.h
void PRINF (string)	Prints without appending a newline to the format string.	Argument1: a string	
void PRINTF (string)	Calls the C printf function. PRINTF appends a newline to the format string; PRINF does not.	Argument1: a string	printf
int RANDOM (int)	Returns a random number	Argument1: the upper bound for the random number. The random	rps_random substrate/utlsCC/RpsMat

Function signature	Description	Arguments and returns	C++ function Include file
	between 0 and one less than Argument1.	number will always be less than this argument. Returns: a random integer	h.h
void RANDOMIZE_SEED ()	Randomly generate a random number seed.		cpp_randomize_seed scheduler/utils/random.h
integer REGMATCH (string, string)	Performs pattern matching on a string.	Argument1: a string upon which to perform pattern matching. Argument2: the string containing the pattern to match. <hr/> For rules about building this string, refer to Using Expression Comparisons. <hr/> Returns: 1 if a match is found; 0 otherwise.	cpp_regex_match scheduler/utils/cpp_spl_utility.h
int RENAME_FILE (string, string)	Renames a file.	Argument1: The name of an existing file. Argument: The new name for the file. Returns: 0 if successful; nonzero otherwise. To read error messages on a non-zero return, use the STRERROR function.	rename_file cpp_io.h
int	Sets the	Argument1: the new	set_debug_level

Function signature	Description	Arguments and returns	C++ function Include file
SET_MSG_LEVEL (int)	current message level.	message level Returns: the previous message level	substrate/utlsCC/Error.h
int SET_RANDOM_SEED(ose[t][int])	Sets the random seed.	Argument1: an oset of integers Returns: 1 when the random seed was set, 0 otherwise.	cpp_set_random_seed scheduler/utls/random.h
void SORT_BY_NAME (ose[t][instance<Spl_Class>])	Sorts an oset of named instances by name.	Argument1: An oset of named instances. This operation is destructive. The list passed in is destructively changed.	cpp_sort_by_name scheduler/utls/cpp_spl_sort.h
int STRERROR (int)	Returns an error message for RENAME_FILE.	Argument1: A nonzero integer value returned upon failure of RENAME_FILE. Returns: The official OS error message.	strerror cpp_io.h
int STRING_COMPARE (string, string)	Calls the C strcmp function.	Argument1: a string Argument2: a string Returns: the result of strcmp	C function strcmp PepperCode compiler includes .h file for you
string STRING_CONCAT (ose[t][string])	Concatenates an oset of strings.	Argument1: an oset of strings Returns: a concatenated string	cpp_string_concat substrate/utlsSPL/CppString.h
date STRING_TO_DATE (string)	Converts a string of format “mm/dd/yy hh:mm:ss” into a date	Argument1: a string of format “mm/dd/yy hh:mm:ss” Returns: a date	cpp_string_to_date scheduler/utls/cpp_spl_misc.h
int STRLEN (string)	Calls the C strlen	Argument1: a string	C function strlen

Function signature	Description	Arguments and returns	C++ function Include file
	function.	Returns: the length of a string	PepperCode compiler includes .h file for you
string STRPRINT (string, values, ...)	Prints to a string.	<p>Argument1: A formatted string similar to what is used in C++ printf. The formatting directive must be %n\$x, where n is the position of the variable argument and x is the data type of the variable. The type of the argument is anything you can use for printf() (except for the * precision argument, such as %.*s). In addition, you can use %D, which prints a date/time value.</p> <p>Argument2: The values, if any, for the formatted string in argument1. There must be a value for each formatting directive in the string.</p> <p>Returns: the string.</p>	<p>strprint</p> <p>substrate/utillsCC/xOpenPrint.h</p>
string STRRPL (string, string, string)	Takes a source string and within that string replaces every occurrence of one string with another string.	<p>Argument1: source string</p> <p>Argument2: string that will be replaced</p> <p>Argument3: replacement string</p> <p>Returns: string resulting from replacing Argument3 with Argument2 in Argument1</p>	<p>strrpl</p> <p>substrate/utillsSPL/CppString.h</p>

Function signature	Description	Arguments and returns	C++ function Include file
string STRSTR (string, string)	Calls the C strstr function.	Argument1: a string Argument2: a string Returns: the result of strstr	C function strstr PepperCode compiler includes .h file for you
int STRING_TO_INT (string)	Converts a string into an integer.	Argument1: a string Returns: an integer	cpp_ascii_to_int scheduler/utills/cpp_spl_misc.h
int TYPEP (class1, class2)	Determines if a class is a descendant of another class	Returns a nonzero value if the second class is a descendant of, or equal to, the first class.	

GET_NAME_OF_CLASS

The function GET_NAME_OF_CLASS will not be included as an intrinsic or built-in function in Release 8.0 or later as it is no longer necessary to use this function. You can now use the "class_name" slot in place of GET_NAME_OF_CLASS.

Every class and instance of a class has a predefined readonly slot of type string called "class_name" which gives the name of the class for the class or instance to which it is applied. The following example demonstrates the use of the "class_name" slot as a replacement for the GET_NAME_OF_CLASS function.

Example:

This example uses actual Release 7.5 code taken from mfg_change_over_repair.spl. In the code, an instance of the CO_Candidate class called co_candidate has just been created, and the name of its class is to be printed to output. In Release 7.5, we did so with the following line of code:

```
PRINTF("\n%s:", GET_NAME_OF_CLASS(GET_CLASS_OF_INSTANCE(co_candidate)));
```

In Release 8.0, you could use the following line of code for this purpose:

```
PRINTF("\n%s:", co_candidate.class_name);
```

This line of code will print the string "CO_Candidate" to output. The "class_name" slot will yield the name of the class for

- An instance of a class (as in the example above)
- A class itself as in the following example.

Example:

The following line of code (if applied to this example) will print the string "CO_Candidate":

```
PRINTF("\n%s::", CO_Candidate.class_name);
```

You may not wish to replace all of your GET_NAME_OF_CLASS calls with the class_name slot. If this is so, there is an option. You could use the cpp_function statement to declare GET_NAME_OF_CLASS as a C++ function. The following line added to your code will do this and will save you a lot of work if you use GET_NAME_OF_CLASS a lot:

```
cpp_function string GET_NAME_OF_CLASS (class<Base_Class>)
"rps_get_name_of_class";
```

TYPEP Example

The following example of TYPEP prints the error message only if the class of the "part" variable was not of type Inventory_Part.

```
if (NOT(TYPEP(Inventory_Part, GET_CLASS_OF_INSTANCE(part)))) {

exit_msg =

NLSPRINT("Unit/Item '%1$s/%2$s' is not derived from a subclass of '%3$s'.",

site_name,part_name,Inventory_Part.class_display_name);

fail();
```

Using Expression Comparisons

Regular expression comparisons use these rules:

.	Matches any single character.
^	Matches the beginning of the string.
\$	Matches the end of the string.
\x	Matches the character x.
[abcd]	Matches any single character from the set abcd.
[^abcd]	Matches any single character not in the set abcd.
[a-d]	Matches any single character between a and d inclusively.
[^a-d]	Matches any single character not between a and d inclusively.
(regexp)	Matches anything that matches regexp.

*	Matches a sequence of 0 or more of the preceding atom.
.	Matches a sequence of 1 or more of the preceding atom.
?	Matches 0 or 1 occurrence of the preceding atom.
e1 e2	Matches either expression e1 or expression e2.

Using Upstairs Objects Functions

When an PepperCode action executes the “fail” statement, it can still return to its caller via output parameters any data which is scalar—values of type float, int, or string—or which is an osset of scalars. However, it cannot return any data structure which requires the creation of an object or a change to any slot on any object, regardless of the data type of the slot, because the context mechanism rolls back—or “undoes”—such changes.

An object is in jeopardy if it would disappear when a “fail” causes the current context to end. This is equivalent to saying that the object appears to have been created in the current context. An object which is in jeopardy may have been created in a child context which ended with a “succeed”, but it will disappear if the current context ends with a “fail”, because the context rollback mechanism operates hierarchically. Similarly, the slot changes are in jeopardy, or the slots appear to have changed in the current context.

The context mechanism records “object-creation” information whenever you invoke `CREATE_OBJECT` or `CREATE_SUBCLASS`. The context mechanism records a “slot change” whenever you use the “=” assignment operator to change a slot on an object.

The following family of C++ functions cause the context mechanism to behave as if an object, which appears to have been created in the current context, had actually been created in the parent context. The functions also cause any slot changes, which appear to have occurred on that object in the current context, to behave as if they had occurred in the parent context instead. These functions move the object-creation and slot-change information “upstairs” to the parent context, so that they are no longer in jeopardy and will be unaffected by a “fail” in the current action, though they will still be rolled back by a “fail” in the parent action.

Upstairs Objects Functions

<i>Function signature</i>	<i>Description C++ function</i>	<i>Arguments and returns</i>
<code>UPSTAIRS_INSTANCE</code> (<code>instance<Base_Class, int></code>) <code>cpp_upstairs_instance</code>	Should be called for each instance before calling fail.	<code>Instance<Base_Class></code> : The instance for which this function is called.

Function signature	Description C++ function	Arguments and returns
		int: When set to a nonzero value, UPSTAIRS_INSTANCE examines each slot on the object and call itself recursively if appropriate.
UPSTAIRS_CLASS (class<Base_Class, int) cpp_upstairs_class	Should be called for each class before calling fail.	Class<Base_Class>: The class for which this function is called. int: When set to a nonzero value, UPSTAIRS_CLASS examines each slot on the object and call itself recursively if appropriate.
UPSTAIRS_OSET_INSTANCE (oset[instance<Base_Class>], int) cpp_upstairs_oset_instance	Should be called for each oset of instances before calling fail.	Oset[instance<Base_Class>]: The oset of instances for which this function is called. int: When set to a nonzero value, UPSTAIRS_INSTANCE examines each slot on the object and call itself recursively if appropriate.
UPSTAIRS_OSET_CLASSES (oset[class<Base_Class>], int) cpp_upstairs_oset_class	Should be called for each oset of classes before calling fail.	Oset[class<Base_Class>]: The oset of classes for which this function is called. int: When set to a nonzero value, UPSTAIRS_CLASS examines each slot on the object and call itself recursively if appropriate.

To use one of these functions, perform the following steps:

1. Create classes and instances in the normal fashion with CREATE_SUBCLASS or CREATE_OBJECT.
2. Change slot values in the normal fashion with the “=” assignment operator.

3. Just before executing “fail”, invoke the appropriate “upstairs” function on each class or instance.

The “upstairs” function move to a higher context the information about the creation of the objects and the changes to their slots, and thereby protects the object-creation and the slot changes from being undone by the “fail”. Moving the context information protects it from jeopardy. Do not change any slots on an object after invoking the “upstairs” function and before executing “fail”, because those changes will not be protected from jeopardy.

When you set the “int” argument to a nonzero value, the function examines each slot on the object and call itself recursively if appropriate. In other words, if a slot contains an instance or class which is in jeopardy, then the function will invoke itself recursively on that instance or class; if a slot contains an oset of instances or an oset of classes, then the function will examine each element of the oset and invoke itself recursively if that particular instance or class is in jeopardy. As a safety feature, the function will not invoke itself recursively on an object which is not in jeopardy, even though some slot changes on that object may themselves be in jeopardy. If you want to move upstairs some in-jeopardy slot changes on a not-in-jeopardy object, you must invoke the “upstairs” function explicitly on the not-in-jeopardy object: recursion will not do this.

Notice that recursion is only an issue for slots of type instance, class, oset of instance, or oset of class. When a function sends a newly created instance upstairs, it also sends upstairs all slot-changes related to that instance, regardless of the slot data type, and no matter whether a slot was changed to point to an object created in the current context or to an object which was not created in the current context. The “recursion” argument determines only whether the function applies itself recursively to instances and classes which are pointed to by slots on the instance (or class) originally passed to the function.

It is legal to execute these functions inside any action, provided that after the current context comes to an end, there will be at least one context remaining below the “workspace” context. Another way to state this restriction is that an “upstairs” function will refuse to move changes into the “workspace” context. Thus, it is not legal to execute these functions in a transaction invoked directly by the action interpreter, whether the action interpreter is operating in the original workspace context or in a bookmark workspace context. If you attempt to execute an “upstairs” function in a context that is not far enough below a workspace context, the function will print a runtime error message and do nothing further.

It is legal to execute these functions inside a “:no_context” action provided there is at least one context between the workspace context and the current context. In the case of a “:no_context” action, the behavior of an “upstairs” function is consistent—it moves the context information upward to the parent context—but it is not necessary what the programmer expects, because in this case the current context is shared with the parent action, and the parent context is associated not with the parent action, but with a more remote ancestor action in the call chain. For example, if a “:no_context” action shares a context with its parent, the “upstairs” function moves the changes to the context belonging to its grandparent; if a “:no_context” action shares a context with its grandparent (because the parent is also a “:no_context” action), the “upstairs” function moves the changes to the context belonging to its great-grandparent; and so on.

It is legal to execute one of these functions explicitly on a class or instance which is not itself in jeopardy; the function will move upstairs any slot changes on that object which are themselves in

jeopardy, even though the object is not. Thus an action can change a slot on an object created by its parent, then call an “upstairs” function on that object to protect the slot change which would otherwise be in jeopardy, and then “fail” without losing the effect of that slot change.

Following is a short example of using the UPSTAIRS_INSTANCE function.

```

action test_upstairs

(local: instance<Base_Class> obj)

{
    obj = create_object(:object_name "animal",
                       :class_name "Named_Class");

    UPSTAIRS_INSTANCE (obj, 1);

    fail();
}

```

Using String Functions for National Language Support

When you wish to create a version of your program targeted at a particular nation, many—but not all—string constants in a program need to be translated into the local language of that nation. Also, string comparisons need to use the algorithms which are appropriate to that language: for example, comparing strings which contain characters with accents and umlauts may require a special string-comparison function.

You can use two functions to support languages targeted at a particular nation: NLSPRINT and NLSTRCMP. Use the following rules to decide when to use these functions:

- Use NLSPRINT to print to a string anything which needs translation to the local language: for example, an error message to the user. Use STRPRINT, which behaves the same as NLSPRINT except that it does not do string translation, to print anything which do not need translation: for example, the name of an action or class.
- Use NLSTR to look up a translation for a string.
- Use NLSTRCMP to compare strings using local-language rules: for example, to sort a list of strings for presentation to the user. Use STRCMP to compare strings without using local-language rules: for example, to compare strings for precise equality regardless of language.

The include file to use with the NLS functions is `substrate/utlsCC/NLString.h`.

International Language Functions

<i>Function signature</i>	<i>Description C++ function</i>	<i>Arguments and returns</i>
<p>string NLSPRINT (string, values, ...)</p> <p>where arg_names is an optional list of argument names.</p>	<p>Prints to a string. Allows for international characters.</p> <p>nlsp rint</p>	<p>Argument1: A formatted string similar to what is used in C++ printf. The formatting directive must be %n\$x, where n is the position of the variable argument and x is the data type of the variable. The type of the argument is anything you can use for printf() (except for the * precision argument, such as %.*s). In addition, you can use %D, which prints a date/time value.</p> <p>Argument2: The values, if any, for the formatted string in argument1. There must be a value for each formatting directive in the string.</p> <p>Returns: the translated string.</p>
<p>string NLSTR (string)</p>	<p>Look up the translated string.</p> <p>nlstr</p>	<p>Argument1: a string. The string that this function looks up a translation for.</p> <p>Returns: the translated string. If no translation table is loaded or no translated string is found, returns the original string.</p>
<p>int NLSTRCMP (string, string)</p>	<p>Compares two strings. Allows for international characters.</p> <p>nlstr cmp</p>	<p>Argument1: a string</p> <p>Argument2: a string</p> <p>Returns: the result of NLSTRCMP, which has the same results as the strcmp function.</p>

Following is an example NLSPRINT statement.

```
exit_msg = NLSPRINT("transaction_create_sales_order: Unit '%1$s' does not
exist.", site_name);
```

Run nlscollect to collect the strings for translation tables. These tables allow your code to print using international languages.

To collect the strings into the translation table, perform the following steps:

1. If you are creating a new translation table, use the following command to copy the given translation table, `rps_nls_collect`, to the translation table file `nls_translation_table`:

```
cp $RPSHOME/resources/rps_nls_table translation_file_name
```

where `translation_file_name` is the name of the file containing the new translation table.

2. Use the following command to run `nlscollect` on your `spl` files. This command will append the translation table for your code onto the translation table file.

```
nlscollect *.spl >> translation_file_name
```

where `translation_file_name` is the name of the file containing the translation table.

3. Have the translator fill in the translations in the translation table.
4. Have the system administrator add the following lines to the `.rps` resource file:

```
TRANSLATION TABLE = translation_file_name
```

```
COLLATION TABLE = collation_file_name
```

```
CHARSET = character_set_name
```

where `translation_file_name` is the name of the file containing the translation table, `collation_file_name` is the name of the file containing the collation table, and `character_set_name` is the name of the character set. The `translation_file_name` and the `collation_file_name` should have the full pathname where these files are stored. The collation table should be in `$RPSHOME/resources`; the default filename is `US-ASCII.collation`. The `character_set_name` should match the `character_set_name` that is listed in the translation table.

The translation table will contain the collected strings, followed by a line to be filled in by a translator. A comment in front of each string shows which file contains the string, along with the line number in the file. The `CHARSET` line tells what `character_set_name` to use (in this case, `US-ASCII`). The following is the beginning of a sample translation table.

```
//!MSGTRANSLATION-RPS-PRA-2.1
```

```
CHARSET=US-ASCII
```

```

{
//mfg_attribute_transactions.spl:988
"transaction_add_mfg_attribute:  Mfg Attribute Name cannot be blank.",
""
}
{
//mfg_attribute_transactions.spl:993
"transaction_add_mfg_attribute:  Mfg Attribute  '%1$s'  already exists.",
""
}
{

```

The following is the beginning of the same translation table., filled out by a translator. In this example, the translated strings just have the letter “X” added.

```

//!MSGTRANSLATION-RPS-PRA-2.0
CHARSET=US-ASCII
{
//mfg_attribute_transactions.spl:988
"transaction_add_mfg_attribute:  Mfg Attribute Name cannot be blank.",
"Xtransaction_add_mfg_attribute:  Mfg Attribute Name cannot be blank."
}
{
//mfg_attribute_transactions.spl:993
"transaction_add_mfg_attribute:  Mfg Attribute  '%1$s'  already exists.",
"Xtransaction_add_mfg_attribute:  Mfg Attribute  '%1$s'  already exists."
}

```


Using Postpone Side Effects Functions

You can postpone side effects. When side effects are postponed, subsequent assignments to input slots do not affect the output slot, and the output slot can be resynchronized later to reflect the eventual values of the input slots. For example, this could be used the "cancel tasks" feature.

Postpone Side Effect Functions

Function signature	Description C++ function	Arguments and returns
RETRACT_AND_POSTPONE_SE (slot)	Invokes the "Retract" method using the current values of the input slots, and then postpones further evaluation of all of the side effects with which that slot is associated. Any subsequent assignment to that slot will not trigger the side effects functions associated with the slot. If a subsequent assignment does change the value of the slot, the histories associated with these functions will remain unchanged and therefore become inconsistent with the value of the input slot.	slot: A reference to a slot that has side effects. Returns: void It is an error to invoke this function on a slot which has no side effects associated with it, but it is harmless to invoke it on a slot for which some or all of the side effects are already postponed. On error, this function prints a message on the server console but allows the server to continue running.
RESYNCH_SE(slot)	Tells the slot to assert the side effects functions associated with the slot, so that all histories associated with those side effects become consistent with the values of all of their input slots.	slot: A reference to a slot that has side effects. Returns: void It is an error to invoke this function on a slot which is not an input to any side effects, but it is harmless to invoke it on a slot whose side effects are not in the postponed state. On error, this function prints a message on the server console but allows the server to continue running.

<i>Function signature</i>	<i>Description C++ function</i>	<i>Arguments and returns</i>
int IS_ASSERTED_SE(slot)	Checks to see if all the side effects associated with a slot are postponed.	<p>slot: A reference to a slot that has side effects.</p> <p>Returns: False if all the side effects associated with the slot are postponed, true otherwise.</p> <p>It is an error to invoke this function on a slot that is not associated with any side effects. On errors, this function prints a message on the server console but allows the server to continue running.</p>

The PepperCode "succeed", "fail", and "leave" operations will behave normally. If an action succeeds, slots retain the states they held at the end of execution of the action; if the action fails or leaves, slots revert to the state they held at the start of execution of the action.

Snapshots will preserve the postponement state of each side effect.

As a side benefit, these functions also eliminate redundant side effect evaluation. The problem is that if two slots "start_time" and "end_time" are associated with the same side effect, the "side_effect_t::Retract" method gets invoked twice and the "side_effect_t::Assert" method gets invoked twice:

```
x.start_time = 5; // Retract, then assert unnecessarily
x.end_time = 10; // Retract unnecessarily, then assert
```

Because nobody references the history between the two assignments, it would be sufficient to invoke the side_effect_t::Retract method once—using the values of the two slots prior to the first assignment—and the side_effect_t::Assert method once—using the values of the two slots after the second assignment.

```
RETRACT_AND_POSTPONE_SE(x.start_time);

    // Save old value of start_time,
    // end_time, and quantity in
    // side_effect_t and retract.

RETRACT_AND_POSTPONE_SE(x.end_time); // Already postponed, so do nothing
```

```

RETRACT_AND_POSTPONE_SE(x.quantity); // Already postponed, so do nothing

x.start_time = 5;           // No side effect processing
x.end_time = 10;           // No side effect processing
x.quantity = 2.0;          // No side effect processing

RESYNCH_SE(x.start_time);    // Invoke "Assert" method
                             // using current values of slots;
                             // change state to "Valid".

RESYNCH_SE(x.end_time); // Not postponed, so do nothing.

RESYNCH_SE(x.quantity);    // Not postponed, so do nothing.

```

Actually, you only need to invoke `RETRACT_AND_POSTPONE_SE` and `RESYNCH_SE` on one of the input slots:

```

RETRACT_AND_POSTPONE_SE(x.start_time);

x.start_time = 5;

x.end_time = 10;

x.quantity = 2.0;

RESYNCH_SE(x.start_time);

```

Using Functions That Query From PepperCode

You can make queries from PepperCode using the same syntax which is available to the client through the action interpreter. You can also issue a special query which operates on an oset of candidate instances, instead of starting with a list of classes.



For more information about the query language, see Writing Queries within the Using PepperTools 8.0 Applications PeopleBook.

The PepperCode interfaces are the `QUERY` and `QUERY_OSET` functions.

The include file to use with these functions is `substrate/utlsSPL/NLString.h`.

Query Functions

Function signature	Description C++ function	Arguments and returns
<pre>int QUERY (string, oset[instance<Base_Class>])</pre>	<p>Makes a query. Returns the result set at time of the query—a list of instances that satisfy the query. It does not create an instance of type “Query,” subscribe to that instance, and periodically reevaluate the query.</p> <p>ai_spl_query</p>	<p>Argument1: a string containing a select statement.</p> <p>Argument2: an oset of instances or classes that will be “filled in” during the function execution. The instances are the ones that satisfy the query.</p> <p>Returns: 0 if there is an error. If there is an error, the oset in argument2 is not changed.</p>
<pre>int QUERY_OSET (oset[instance<Base_Class>], string, oset[instance<Base_Class>])</pre>	<p>Issues a query against an existing oset of instances instead of a list of classes, as is done in QUERY.</p> <p>ai_spl_query_oset</p>	<p>Argument1: The candidate instances.</p> <p>Argument2: the where and order-by clauses.</p> <p>Argument3: the result instances.</p>

An example of a simple PepperCode function which uses the QUERY function is:

```
action test_query(input: string select_statement,
output: oset[instance<Base_Class>] results)
{
    if (QUERY(select_statement, results))
        succeed();
    fail();
}
```

In QUERY_OSET, the string in argument 2 must contain a “where” clause or an “order-by” clause with an optional semicolon at the end, as in the following example:

```
local: oset[instance<Base_class>] candidates,
```

```

output: oset[instance<Base_Class>] results

...

QUERY_OSET(candidates, "where quantity > 50 order by display_name;", results);

```

It applies the "where" predicate to the instances in the candidate list, selects the ones which pass the test, and then sorts the remaining ones using the "order by" keys. The QUERY_OSET function returns zero on error, one on success.

Here is an example of a simple PepperCode function which uses the QUERY_OSET function to achieve the same effect as a normal query:

```

action test_oset_query(input: string class_name,

    input: string where_and_order_by_clauses,

    output: oset[instance<Base_Class>] results,

    local: oset[instance<Base_Class>] candidates)

{

    // Get a list of all instances of the specified class

    GET_DESCENDANTS(candidates, GET_CLASS_BY_NAME(class_name), 1);

    // Filter them using the specified clauses

    if (QUERY_OSET(candidates, where_and_order_by_clauses, results))

        succeed();

    fail();

}

```

Using History Functions

The following table lists the history functions. The include file to use with these functions is scheduler/utls/cpp_spl_history.h. The C++ function is cpp_*NAME*, where *NAME* is the function name.

History Runtime Functions

Function signature	Description	Arguments and returns
float GET_END_OF_HISTORY_VALUE	Gets the last history element in the history. This should be the same as GET_VALUE(history,	Argument1: a history. Return: the value of the last history element in the

Function signature	Description	Arguments and returns
(history<float>)	end_of_time).	history.
int QUANTITY_OF_HISTORY_EXISTS (history<float>, float, date, date)	Tests to see if a quantity in a history always exists over a given length of time.	Argument1: a history. Argument2: a quantity. Argument3: start date. Argument4: end date. Returns: TRUE if no point in time between the start data and the end date has a value < quantity.
int QUANTITY_OF_HISTORY_EXCEEDS (history<float>, int, date, date)	Tests to see if a quantity in a history exists at all over a given period of time.	Argument1: a history. Argument2: a quantity. Argument3: start date. Argument4: end date. Returns: TRUE if any point in time between the start data and the end date has a value > quantity.
float MAX_QUANTITY_OVERALLOCATED (history<float>, date, date)	Finds the maximum quantity that is overallocated. Overallocated means value < 0.	Argument1: a history Argument2: start date Argument3: end date Returns: The maximum amount that any point in time is overallocated between the start time and the end time. The absolute value of the value is returned. Returns 0 if no point in time is overallocated.
void GET_OVERALLOCATED_CHANGERS (oset[instance<Spl_Class>], history<float>, date, date)	Fills the changers list with a list of all resource constraints that have a side effect in the changers list of a history element that is overallocated. Overallocated means value < 0.	Argument1: the changers list. An oset of instances that will be filled during the function execution. Argument2: a history. Argument3: start time. Argument4: end time.

Function signature	Description	Arguments and returns
date NEXT_TIME_TO_TRY (history<float>, instance<Spl_Class>, float, date, int, int, time, date)	Searches for a set of intervals of a given duration, where the calendar is legal, and the quantity exists in the history.	Argument1: a history. Argument2: a calendar. This function looks for intervals where this calendar is legal. Argument3: quantity. This function searches for intervals where this quantity exists in this history. Argument4: the start date. Argument5: search direction. If 1, search later than the start date. If not 1, search earlier than the start date. Argument6: the splittable flag. If 1, the returned intervals do not have to be contiguous. Argument7: the duration of the intervals to search for. Argument8: an invalid time. Returns: the next time to try. If no intervals are found, returns the invalid time.
date STATE_NEXT_TIME_TO_TRY (history<string>, history<string>, string, int, date, int, time, date)	Searches for a set of intervals of a given duration, where the calendar is legal, and the quantity exists in the state history.	Argument1: a state history. Argument2: a calendar. This function looks for intervals where this calendar is legal. Argument3: the name of the state. Argument4: flag to use either the achieve state match test or the string match test.

Function signature	Description	Arguments and returns
		<p>Argument5: the start date.</p> <p>Argument6: search direction. If 1, search later than the start date. If not 1, search earlier than the start date.</p> <p>Argument7: the duration of the intervals to search for.</p> <p>Argument8: an invalid time.</p> <p>Returns: the next time to try. If no intervals are found, returns the invalid time.</p>
float GET_HISTORY_VALUE (history<float>, date)	Returns the value for this time point in the history	<p>Argument1: a history.</p> <p>Argument2: a time point in this history.</p> <p>Returns: value at this time point in the history.</p>
float AREA_UNDER_CURVE (history<float>, date, date)	<p>Computes the sum of positive area under the curve of the history passed in from a start time to an end time.</p> <p>For all intervals or partial intervals:</p> <p>— if (interval value > 0) then (interval end - interval start) * interval value else 0</p>	<p>Argument1: a history.</p> <p>Argument2: start time.</p> <p>Argument3: end time.</p> <p>Returns: the sum of the positive area under the curve of the history from the start time to the end time.</p>
void MOST_OVERALLOCATED_CHANGERS (oset[instance<Spl_Class>], history<float>, date, date)	Return the changers for the most overallocated interval in the history between st and et.	<p>Argument1: the changers list. An oset of instances that will be filled during the function execution.</p> <p>Argument2: a history.</p> <p>Argument3: the start time.</p> <p>Argument4: the end time.</p>

Function signature	Description	Arguments and returns
float GET_INITIAL_AMOUNT (history<float>)	Return the value at the beginning of time.	Argument1: a history. Returns: the value at the beginning of this history.
float MIN_HISTORY_VALUE (history<float>, date, date)	Return the minimum history value.	Argument1: a history. Argument2: start time. Argument3: end time. Returns: The minimum value for all the time points between the start time and the end time.
void ANALYZE_HISTORY (history<float>, history<float>, oset[int])	Return an analysis of a history.	Argument1: a history. Argument2: the history that represents the original values, before anything is changed. Argument3: an oset of four integers that will be filled during the function execution. These are: the number of supply tasks effecting history, the number of constraints effecting history, the number of history elements in initial history, and the number of history elements in history.
int STATE_EXISTS (history<string>, string, date, date)	Tests to see if a state exists.	Argument1: a history. Argument2: a state. Argument3: start time. Argument4: end time. Returns: TRUE if the state is the value of at least one point in time from the start time to the end time.

Function signature	Description	Arguments and returns
date NEXT_LEGAL_CALENDAR_TIME (history<string>, date, date)	Gives the date of the next time when the calendar is legal.	Argument1: a calendar. Argument2: a start time. Argument3: an invalid date. Returns: the date of the next place after the start time when the calendar is legal. If no such time exists, returns the invalid date.
int IS_LEGAL_CALENDAR_TIME_FOR_SPLITTING (history<string>, date, int) (Note: In the code, this is shown as a date. However, int is the proper type to use.)	Tests a calendar to see if a time is a legal time to begin a split child task.	Argument1: a calendar. Argument2: start time. Argument3: search direction. If 1, search later than the start time. If not 1, search earlier than the start time. Returns: TRUE only if the start time is a legal time to begin a split child task in the direction indicated. For example, 8:00 AM is not legal for a backward split on a 5 day 2 shift calendar, but it is legal on a 5 day 3 shift calendar. Returns FALSE (0) if it can not find time.
date PREVIOUS_LEGAL_CALENDAR_TIME (history<string>, date, date)	Returns the previous time that the calendar is legal.	date PREVIOUS_LEGAL_CALENDAR_TIME (history<string> calendar, date st, date invalid_date) Argument1: a calendar. Argument2: start time. Argument3: an invalid date. Returns: the date of the next time before the start

Function signature	Description	Arguments and returns
		time that the calendar is legal. If none exists, returns the invalid date.
int TIME_BETWEEN_TWO_POINTS_FOR_CALENDAR (history<string>, date, date)	Computes the duration of legal time on the calendar.	Argument1: a calendar. Argument2: start time. Argument3: end time. Returns: the duration of legal time on this calendar between the start time and the end time.
date NEXT_CALENDAR_BREAK (history<string>, date)	Finds the next break in the calendar.	Argument1: a calendar. Argument2: a time. Returns: the end time of the interval that contains the given time.
date PREVIOUS_CALENDAR_BREAK (history<string>, date)	Find the previous break in the calendar.	Argument1: a calendar. Argument2: a time. Returns: the start time of the interval that contains the given time.
void ADD_TO_HISTORY_VALUE_ON_CALENDAR (history<float>, history<string>, float)	Add a quantity to a history at all points in time where a given calendar is legal.	Argument1: a history. Argument2: a calendar. Argument3: a quantity that is added to the history whenever the calendar is legal.
void GET_INVENTORY_AREAS (oset[date], oset[date], oset[float], history<float>, date, date)	Computes areas of positive inventory. An area is a continuous set of time points where the value is always positive. The history is passed in and calculated from start time to end time, the results are passed out such that the first element in the oset of start times is the start time of the first area, the first element of the oset of end	Argument1: an oset of start times for computed areas. This will be filled during the function execution. Argument2: an oset of end times for computed areas. This will be filled during the function execution. Argument3: an oset of areas that have a positive inventory. This will be filled during the function

Function signature	Description	Arguments and returns
	times is the end time of the first area, and the first element in the osset of areas is the area—time * quantity—containing positive inventory.	execution. Argument4: a history. Argument5: start time at which to compute the areas. Argument6: end time at which to compute the areas.
int NUMBER_OF_AREAS_SHORT (history<float>, date, date, float)	Count the number of areas where the value does not drop below threshold.	Argument1: a history. Argument2: start time. Argument3: end time. Argument4: a threshold. Returns: the number of areas from start time to end time in this history where the value does not drop below the threshold.
date GET_DATE_OF_NEXT_NEGATIVE_VALUE (history<float>, date, date)	Find the next point in time where the value < 0.	Argument1: a history. Argument2: start time. Argument3: an invalid date. Returns: the next time after the start time where the history value is < 0. If no such time is found, returns the invalid date.
date GET_DATE_OF_PREVIOUS_NOT_ENOUGH (history<float> history, float enough, date st, date et)	Find the previous point in time where the value is not enough.	Argument1: a history. Argument2: a history value that represents enough value. Argument3: a start time. Argument4: an invalid time. Returns: the previous time before the start time where the history value is < enough. If no such time is found, returns the invalid

Function signature	Description	Arguments and returns
		date.
void ADD_TO_HISTORY_VALUE (history<float>, float)	Adds a quantity to the value for all time points on history	Argument1: a history. During function execution, a quantity will be added to the value on all time points on this history. Argument2: a quantity to add to all time points on the history.
void GET_ALLOCATED_CHANGERS (oset[instance<Spl_Class>], history<float>, date, date)	Return all resource constraints that have an effect on history during intervals where the value > 0.	Argument1: changers. An osset of resource constraints that will be filled during function execution. These are all the resource constraints that have an effect on the history where the value > 0. Argument2: a history. Argument3: start time at which to return resource constraints. Argument4: end time at which to return resource constraints.

Using Dump Functions

The following functions allow you to dump information to a file. The include file to use with these functions is scheduler/utis/cpp_spl_dump.h. The C++ function is `cpp_NAME`, where *NAME* is the function name.

Dump Functions

Function signature	Description	Arguments and returns
int OPEN_DUMP_FILE(string, string)	Opens a file that can be modified by the dump functions. Only one file at a time can be opened by OPEN_DUMP_FILE. If	Argument1: The complete filename, including the path, to be opened. Argument2: The mode of

Function signature	Description	Arguments and returns
	the open fails, the dump_fail flag is set to 1.	the file open. "w" for overwrite the existing file, and "a" for append to the existing file. Returns: 1 if successful, 0 if dump failed.
int CLOSE_DUMP_FILE()	Closes the file that was opened by OPEN_DUMP_FILE.	Returns: 1 if successful, 0 if dump failed.
void DUMP_DATE(date, int, int)	Prints a date to the dump file.	Argument1: The date to be printed. Argument2: The number of characters used in the print format. Argument3: A flag to determine if the output to the dump file is right-justified. 0 = left-justified, 1 = right-justified.
void DUMP_FLOAT(float, int, int)	Prints a float to the dump file.	Argument1: The float to be printed. Argument2: The number of characters used in the print format. Argument3: A flag to determine if the output to the dump file is right-justified. 0 = left-justified, 1 = right-justified.
void DUMP_INT(int, int, int)	Prints an integer to the dump file.	Argument1: The integer to be printed. Argument2: The number of characters used in the print format. Argument3: A flag to determine the justification for the output to the dump file. 0 = left-justified, 1 = right-justified.

Function signature	Description	Arguments and returns
void DUMP_NEWLINES(int, int, int)	Prints newlines to the dump file.	Argument1: The number of newlines to print.
void DUMP_SPACES(int, int, Int)	Prints spaces to the dump file.	Argument1: The number of spaces to print.
void DUMP_STRING(string, int, int)	Prints a string to the dump file.	Argument1: The string to be printed. Argument2: The number of characters used in the print format. Argument3: A flag to determine if the output to the dump file is right-justified. 0 = left-justified, 1 = right-justified.
void DUMP_TIME(time, int, int)	Prints a time to the dump file.	Argument1: The time to be printed. Argument2: The number of characters used in the print format. Argument3: A flag to determine if the output to the dump file is right-justified. 0 = left-justified, 1 = right-justified.
int DUMP_RESET_STATUS ()	Resets the dump_failed flag to 0 and returns 0.	Returns: 0
int DUMP_TEST_RESET_ST ATUS()	Resets the dump_failed flag to 0 and returns the previous value of the dump_failed flag. This is useful if you want to check the value of the dump_failed flag.	Returns: The value of the dump_failed flag (before it is reset to 0 by this function).

CHAPTER 11

Writing PepperCode Applications

This section provides guidelines for creating PepperCode applications.

Writing a PepperCode Class

Here is an example of a class that shows its main elements:

Following are general guidelines for writing a PepperCode class. Although you can use these guidelines in any order, you could follow them sequentially as you design a class.

Naming A Class

When writing the name of the class, separate and capitalize each “word” of the class name.

This: `class Spl_Class`

Not this: `class spl_class`

This will allow action parameters to be easily named by using the lowercase version of the class. For example:

```
input: instance<Spl_Class> spl_class,
```

Naming Class Slots

Use lowercase when naming the slots of a class.

This: `action<delete> delete_action`

Not this: `action<delete> Delete_Action` or `action<delete> DELETE_ACTION`

Adding An Action To A Class

When adding a new action to a class (commonly referred to as a method), make the action slot a “class slot.”

Very little memory is required for a class slot, because the slot is stored on the class and not on each instance. Here is an example of a class slot:

```
class Spl_Class : Base_Class {

    action<delete> delete_action

};

slot Spl_Class.delete_action { default: delete_spl_class class_slot: };
```

Notice that a slot default value is needed for specifying a class slot.

However, when assigning an action to an existing action slot (commonly referred to as “specializing the method”), using the `class_slot`: keyword is unnecessary and should not be used to avoid confusion. For example:

```
class My_Spl_Object : Spl_Class {

};

slot My_Spl_Object.delete_action { default: delete_my_spl_object };
```

`delete_action` isn’t defined at this level, and therefore isn’t called a class slot.

When adding an action to a class, always provide a default method.



If a default method is not provided, any PepperCode code that accesses and attempts to execute the action will break.

Adding Default Values To A Class

When defining default values, place the slot statements directly below the class definition, as in the previous example of `Spl_Class`.

This will make the class definition readable and easy to modify.

Specializing Class Slots

Specialize slots when possible.

In PepperCode, the data type of a slot can be “specialized” when the slot is inherited from another class. Specializing slots will prevent unnecessary casting to the needed type. Specializing a slot does not add an extra slot (or any extra memory) to an object. Only the type of slot is changed.

Here are a few classes and PepperCode statements that demonstrate why specializing slots is a good idea:

```
class Basic_Task : Spl_Class { // A basic task object
```

```
};

class Child_Task : Basic_Task {           // A child task
    int child_task_information // with some stuff stored on it
};

class Parent_Task : Basic_Task { // A parent task
    oset[instance<Child_Task>] children
    // with child tasks stored on it
};
```

If you wanted to write new parent and child task classes that inherit from the previous classes, you could do the following. The new child task class will have additional slot information that does not exist in the Child_Task task. The new parent class will have only instances of the new child class stored in its children slot. Here are the new classes without slot specialization:

```
class My_Child_Task : Child_Task { // New child task
    string my_information // with some new stuff stored on it
};

class My_Parent_Task : Parent_Task { // New parent task
};
```



Note: In this case, “parent” and “child” refer to part-whole relationships, not to class-subclass relationships. A Child_Task is not a subclass or instance of a Parent_Task.

Because only tasks of class My_Child_Task are stored in the children slot of My_Parent_Task, and there is no slot specialization on the children slot of class Parent_Task, the following code will have to “cast” to the appropriate class before the slot my_information can be referenced from the My_Child_Task class:

```
action find_child_by_using_my_information
(
    input: instance<My_Parent_Task> my_parent_task,
    input: string my_information,
    local: instance<My_Child_Task> temp_child_task,
    // Must use a cast variable
    output: instance<My_Child_Task> my_child_task,
```

```

        no_context:)
    {
        my_child_task = 0;           // Specify a default for the output variable.
        foreach child_task in my_parent_task.children {
            temp_child_task = child_task;    // Must cast so that the
            // slot my_information can be accessed.
            if (temp_child_task.my_information == my_information) {
                my_child_task = temp_child_task;
                succeed();
            }
        }
        succeed();
    }
}

```

Without the above cast from `child_task` to `temp_child_task`, the compiler would not have allowed the slot `my_information` to be accessed.



For more information about the term “cast,” refer to Using Casting.

If the slot `children` on the class `My_Parent_Task` was specialized to the correct type, as in the following code, the action `find_child_by_using_my_information` could be written without the cast. To specialize a slot, redefine the existing slot name. In this example, the slot `children` does this.

```

class My_Parent_Task : Parent_Task {           // New parent task
    oset[instance<My_Child_Task>] children    // Specialize slot.
};

action find_child_by_using_my_information

    (input: instance<My_Parent_Task> my_parent_task,
     input: string my_information,
     output: instance<My_Child_Task> my_child_task,
     no_context:)

```

```

{
    my_child_task = 0;           // default the output variable

    foreach child_task in my_parent_task.children {
        if (child_task.my_information == my_information) {
            // No cast necessary

            my_child_task = child_task;

            succeed();
        }
    }

    succeed();
}

```

More examples of specializations can be found in the file `scheduler/spl/constraints/constraints.spl`. Look at the slot object and how it is specialized on each resource constraint class.

Using Casting

Casting (or downcasting) means altering the type of an inherited or system-defined parameter. You do it so that the object which is the binding of the parameter will come to have the proper slots.

Assume you're writing a method action in the `Dog` class called `go_to_kennel_action`. It will become the value of the `go_home` method defined at the `Mammal` class level, whose default action is `default_go_home_action`. And that default action belongs to a schema, `go_home_schema`, which provides the input parameter `mammal_self`, representing the `Mammal` which goes home. The `mammal_self` input parameter is typed as a `Mammal`.

```
(input: instance<Mammal> mammal_self, ?)
```

Since your `go_to_kennel_action` also belongs to the `go_home_schema`—it had better, or it won't be allowed as the value of the method slot—it also inherits the `mammal_self` parameter. But your action wants to do something with the `Kennel` slot on the object passed in as `mammal_self`. A `Dog` doesn't just go to any old home: it has to be a kennel. And the problem is that a generic `Mammal` instance doesn't have a `Kennel` slot. For that, you have to be an instance of `Dog`. So you have a problem, because if your action refers to the nonexistent slot `Kennel` on the object bound to the `mammal_self` parameter, crashes will result. So you have to (down)cast the binding of `mammal_self` in order to alter its type. This insures that it will be interpreted as a `Dog` instance.

The syntax is:

```

action<go_home_schema> go_to_kennel_action (local: instance<Dog>
locvar_dog_self, ?)

locvar_dog_self = mammal_self

```

Once you've done this, and only then, can you confidently refer to Kennel and any other slots that only a Dog would have on the object passed in as `mammal_self` and now bound to `locvar_dog_self`.

Writing a PepperCode Action

Here is an example of an action definition that shows some main elements:

```

// Action

action print_simple_string


// Action parameters

(input: string pstring = "Null",
 local: int string_length = 0,
 output: int printed)


// Action body
{
    printed = 0;

    string_length = STRLEN (pstring);

    if (string_length < 2) {
        PRINTF("\n%s", pstring);
        printed = 1;
        succeed();
    }
}

```

Following are general guidelines for writing a PepperCode action. Although you can use these guidelines in any order, you could follow them sequentially as you design an action.

Using no_context

Use the `no_context`: keyword appropriately.

Every action generates a new context by default. If the `no_context`: keyword is used in the parameter list of an action, then that action will not generate a new context.

Determining when an action should have a context is not always clear. Here are a few rules that make the decision easier.

An action should generate a new context if it can fail after PepperCode objects have been modified. In this case, the modification of PepperCode objects includes the following:

- slot modification on instances or classes
- the creation of PepperCode instances and classes
- the deletion of PepperCode instances and classes

An action should not generate a new context if no data values are modified, no objects are created, and no objects are deleted. This is commonly done in reports.

Whenever possible, don't allow actions to generate unnecessary contexts. Extra context processing is very inefficient.

Avoiding Static Parameters

Beware of static action parameters.

Because static action parameters maintain their values until the calling action exits, it is easy to introduce bugs while using them. This section lists the action parameter defaults.



For more information about action parameters, refer to [Writing Action Parameters](#).

Here are some practical rules that should help. When an action is called multiple times from the same action, do not rely on its default values unless you reset them before the action exits. An example of resetting default values is in the action `create_object` from the file `scheduler/spl/dispatch.spl`. The `create_object` action resets the parameter `object_name` before it succeeds or fails.

```
action create_object

{

    if (object_name == "") {

        PRINTF("\n\n\nHEY!!!  YOU CANNOT CREATE AN OBJECT OF CLASS  %s  WITH AN
EMPTY NAME!!!\n\n\n",

            class_name);

    }
```

```

    new_object = 0;

    object_name = "__Anonym__";    // reset because of static action parameters

    fail();

}

else {

    new_object = CREATE_OBJECT(object_name, class_name);

    object_name = "__Anonym__";    // reset because of static action parameters

    succeed();

}

}

```

Checking The Output Variable On An Action

When using an output variable on an action, make sure that it has a value before the action exits.

This will allow other actions to use the output variable, even if the action fails. Default values cannot always be used reliably for this, because action parameters are static and maintain values across multiple function calls.

This is the correct way:

```

action get_some_value

(input: instance<Some_Spl_Object> object,

 output: int some_value,

 no_context:)

{

    some_value = 0;                // Provide an output for the output parameter.

    //

    // Other code here to find and set some_value.

    //

    succeed();

}

```

Here are two incorrect ways:

```

action get_some_value

```



```

    (input: instance<Some_Spl_Object> object,
     output: int some_value,
     no_context:)
{
    // No default provided here

    //

    // Other code here to find and set some_value.

    //

    succeed();
}

action get_some_value

    (input: instance<Some_Spl_Object> object,
     output: int some_value = 0,           // Won't work for multiple calls.
     no_context:)
{
    //

    // Other code here to find and set some_value.

    //

    succeed();
}

```

Grouping Action Parameters

For readability, group action parameters together by type.

This is the correct way:

```

action create_production_in_period

    (input: instance<Part> part,
     input: float quantity,
     input: date period_start,

```

```

input: date period_end,

input: instance<Equipment_Resource> equipment_resource,

input: instance<Build_Option> build_option,

input: int batch_method = 0,

input: int return_production = 0,

input: string routing_class_name = "Routing_Parent",

local: oset[instance<Spl_Class>] objects,

local: oset[string] strings,

local: action<choose_build_option> choose_build_option,

local: action<set_current_resource_on_bors> set_current_resource_on_bors,

local: action<clear_current_resource_on_bors> clear_current_resource_on_,

local: action<equipment_matches_any_bor> equipment_matches_any_bor,

local: action<production_supply_in_period> production_supply_in_period,

local: action<create_production> create_production,

output: oset[instance<Routing_Parent>] new_production,

output: string exit_msg = "",

no_context:)

```

This is the incorrect way:

```

action create_production_in_period

(input: instance<Part> part,

local: oset[instance<Spl_Class>] objects,

output: oset[instance<Routing_Parent>] new_production,

local: oset[string] strings,

input: float quantity,

input: int batch_method = 0,

input: int return_production = 0,

local: action<choose_build_option> choose_build_option,

input: string routing_class_name = "Routing_Parent",

local: action<set_current_resource_on_bors> set_current_resource_on_bors,

```

```

output: string exit_msg = "",

local: action<clear_current_resource_on_bors> clear_current_resource_on_,

input: date period_start,

local: action<equipment_matches_any_bor> equipment_matches_any_bor,

input: date period_end,

local: action<production_supply_in_period> production_supply_in_period,

input: instance<Equipment_Resource> equipment_resource,

local: action<create_production> create_production,

input: instance<Build_Option> build_option,

no_context:)

```

Writing A PepperCode Transaction

An PepperCode transaction is a type of action. Transactions are special actions for having the Production Planning interface with the user. Due to this, all transaction inputs are strings, floats, and ints (for readability's sake). A transaction is formatted text that represents a command to execute inside Planning. Transactions exist in many different places, such as:

- command files generated by the data bridge,
- command files edited in a text editor,
- log files sent from the client to the server for normal operations,
- and log files sent via the communications API to command the server.



Note: When placing filenames in double quotes (" ") for Windows NT, you can use a file delimiter of "/" instead of "\".

The format of every transaction is as follows:

```
transaction_name (:keyword1 value1 :keyword2 value 2)
```

Transactions are actions that are of type action<transaction>. By convention, they have transaction_ as the beginning of their name.

Following are general guidelines for writing a PepperCode transaction. Although you can use these guidelines in any order, you could follow them sequentially as you design a transaction.

Starting Transaction Names With transaction_

Always use transaction as the first word of the transaction name.

Some of the substrate code depends on the fact that transaction names begin with transaction.

This: transaction_create_sales_order

Not this: create_sales_order

Using The Action Schema Transaction

Always use the action schema transaction when writing a transaction.

This schema provides special parameters and behavior for transactions.

This: action<transaction> transaction_create_sales_order

Not this: action transaction_create_sales_order

Putting Minimal Code Into A Transaction

Whenever possible, don't put too much application code in a transaction.

The purpose of a transaction is to collect input, perform error checking, call an action to do the "real work," and then provide error, warning, and informational messages.

For example, the transaction transaction_create_sales_order does not create a sales order within its action body. Instead, it calls the action create_sales_order which creates the sales order object.

Including No Instances, Classes, Histories, Or Actions

The input parameters for a transaction should never include instances, classes, histories, or actions.

Using an instance or class as input to a transaction is commonly referred to as "passing a uid" to a transaction. Passing uids to a transaction as input will cause the reload of a log file to break.

Using Default Values For Input Parameters

Use default values for input parameters whenever possible.

Notice that this advice is different for writing actions. Because transactions are usually called from the "top level" from a page or menu file, the default values can be trusted. Because transactions are rarely called from other actions or transactions, the default values will always be maintained correctly.



For more information, refer to Writing Action Parameters.

Performing Error Checking

Perform error checking on the transaction inputs.

Every transaction must validate its input values. When a problem is discovered, the transaction should return an error message or warning message through the `exit_msg` output parameter. The following transactions are good examples of transactions that perform error checking (they are in the file `mfg/spl/sales_order/mfg_sales_order_transactions.spl`):

```
transaction_create_sales_order
```

```
transaction_add_sales_order_line
```

Here is the transaction `transaction_create_sales_order`:

```
action<transaction> transaction_create_sales_order

    (input: string site_name = "",
     input: string sales_order_name,
     input: string class_name = "Sales_Order",
     input: string order_date = "",
     input: string customer = "",
     local: instance<Site> site,
     local: instance<Environment> environment,
     local: date o_date,
     local: instance<Customer> customer_instance,
     local: instance<Sales_Order> so,
     output: instance<Sales_Order> sales_order = 0)
{
    if (site_name == "") {
        site_name = GET_PARENT_ENV().default_site.name;
    }

    site = GET_INSTANCE_BY_NAME(site_name);
```

```

if (NOT(site)) {

    exit_msg = NLSPRINT("Unit '%1$s' does not exist.",site_name);

    fail();

}

// Check whether the instance comes from right class .

if (NOT (TYPEP(Site,GET_CLASS_OF_INSTANCE (site)))) {

    exit_msg = NLSPRINT("'%1$s' is not of type
'%2$s'.",site_name,Site.class_display_name);

    fail ();

}

if(class_name == "")

    class_name = "Sales_Order";

so = GET_INSTANCE_BY_NAME (APPEND_STRINGS(site.name,sales_order_name));

if (so) {

    exit_msg = NLSPRINT("Unit/Sales order '%1$s/%2$s' already
exists.",site_name,sales_order_name);

    fail();

}

// ensures class exists, but not that class is appropriate (ie subclass of
Sales_Order).

if (NOT (GET_CLASS_BY_NAME(class_name))) {

    exit_msg = NLSPRINT("Class '%1$s' does not exist as a Sales Order
class.",class_name);

    fail();

}

// Check for the validity of the class.

if (NOT (TYPEP(Sales_Order,GET_CLASS_BY_NAME (class_name)))) {

    exit_msg = NLSPRINT("'%1$s' is not of type Sales_Order.",class_name);

```

```
fail ();
}

if (EQ(customer, "")) {
    exit_msg = NLSPRINT("Customer '%1$s' is Invalid.",customer);
    fail ();
}

customer_instance = GET_INSTANCE_BY_NAME(customer);

if (NOT(customer_instance)) {
    execute transaction_create_customer(:name customer);
}
else {
    if (NOT(TYPEP(Customer,GET_CLASS_OF_INSTANCE(customer_instance)))) {
        execute transaction_create_customer(:name customer);
    }
}

customer_instance = GET_INSTANCE_BY_NAME(customer);

environment = GET_PARENT_ENV();

o_date = STRING_TO_DATE(order_date);
if (NOT (o_date)) {
    exit_msg = NLSPRINT("Order Date '%1$s' is invalid.",order_date);
    fail();
}
```

```

execute create_sales_order(:sales_order_name sales_order_name,

                           :site site,

                           :order_date o_date,

                           :class_name class_name,

                           :customer customer_instance);

if (create_sales_order.status == SUCCEEDED) {

    sales_order = create_sales_order.new_sales_order;

    succeed();

}

else {

    fail();

}

}

```

Here are some basic rules about performing error checking in transactions.

- When referencing a named instance, use the function GET_INSTANCE_BY_NAME.

If GET_INSTANCE_BY_NAME returns 0, the transaction should fail. For an example, see the error check on the input parameter `sales_order_name` in the transaction `transaction_create_sales_order`, seen above and in the `mfg_sales_order_transactions.spl` file.

- When referencing a named class, use the function GET_CLASS_BY_NAME.

If GET_CLASS_BY_NAME returns 0, the transaction should fail. For an example, see the error check on the input parameter `class_name` in the transaction `transaction_create_sales_order`, seen above and in the `mfg_sales_order_transactions.spl` file.

- When appropriate, provide a default value for date inputs.

Most dates default to `early_fence`, `late_fence`, `start_of_time`, or `end_of_time`. For an example, see the error check on the input parameter `start_time` in the transaction `transaction_add_bom_to_build_option`, in the `mfg_routing_transactions.spl` file). The following excerpt from that transaction shows only the relevant code.

```

action<transaction> transaction_add_bom_to_build_option

    (input: string build_option_name,

     input: string part_name,

     input: string site_name = "",

     input: int step,

```



```

    input: float quantity,

    input: string constraint_class_name = "Standard_RM_Constraint",

    input: int blowthrough = 0,

input: int configurable = 0,

    input: string start_time = "",

    input: string end_time = "",

    // locals declared here
{

    // more transaction code goes here.

    if (start_time == "") {

        s_time = GET_PARENT_ENV().start_of_time;

    }

    else {

        s_time = STRING_TO_DATE(start_time);

    }

    // remainder of the transaction code goes here

}

```

- The function `STRING_TO_DATE` returns 0 when passed an invalid date string.

If `STRING_TO_DATE` returns 0 in a transaction, the transaction should fail. For an example, see the error check on the input parameter `order_date` in the transaction `transaction_create_sales_order` (seen above and in the `mfg_sales_order_transactions.spl` file).

Using #document and #end_document

Write documentation by using `#document` and `#end_document`.

Use `transaction_add_sales_order_line` as an example for writing transaction documentation.



For more information, refer to Adding and Retrieving Documentation.

Always use the primary key of an object for identification.

Writing A PepperCode Method

In a class definition, a slot of type action is an implementation of a PepperCode method, like a C++ member function or method:

```
action<schema_name> name
```

The action method stored in an action slot can be referenced and executed. Any action of that schema type can be assigned to that slot.

The “dispatch” of a method—the process of calling the correct method associated with a class—is not performed automatically. Instead, the value of a local action parameter is defined and the action is called through the local parameter.

Following are general guidelines for writing a PepperCode method. Although you can use these guidelines in any order, you could follow them sequentially as you design a method.

Writing Actions That Dispatch The Method

When possible, write an action that dispatches the method.

This will provide a consistent API—application programming interface—for other PepperCode code to use. For example, the following action `delete_object` dispatches the `delete` method for any PepperCode object of type `Spl_Class`. Notice that the input to the dispatcher action is the same as the input defined on the `delete` action schema.

```
action_schema delete_schema

(input: instance<Spl_Class> object,
 no_context:);

class Spl_Class : Base_Class {
    action<delete_schema> delete_action
};

slot Spl_Class.delete_action { default: default_delete class_slot: };

// Note: default_delete is not shown.

action delete_object

(input: instance<Spl_Class> object,

 local: action<delete_schema> delete_action,
```

```

        no_context:)
    {
        delete_action = object.delete_action;           // Lookup the delete
method.

        execute delete_action(:object object);         // Call the delete method.

        succeed();
    }

```

Implementing Input And Output Parameters

Implement all of the input and output parameters in the action schema for the method.

This will allow the dispatch of the method to use a standard API.

Including The Object As An Argument

Always include the object on which the method is stored as an argument to the method.

Because there is no method data type in PepperCode, you use action schemas to implement methods. Because of this, you have to provide our own backpointer to the object of the method. In the previous example, the input parameter `object` serves this purpose.

Casting The Inner Object To The Class

In the body of the method, cast the input object to the PepperCode class that the method assumes.



For more information about casting, refer to Using Casting.

For example, here is the delete method for a sales order. Notice that without the cast, you would not be able to reference any of the specific slots of a `Sales_Order` (like `name` and `sales_order_lines`).

```

action<delete> delete_sales_order

    (local: instance<Sales_Order> sales_order)

    {

        sales_order = object;
    }

```

```
MSG(25, "\nDeleting Sales Order  %s", sales_order.name);

execute delete_object_list(:object_list sales_order.sales_order_lines);

execute delete_object_list(:object_list sales_order.ship_sets);


DELETE_OBJECT(sales_order);

succeed();

}
```

Writing a C++ Utility

This section describes how to write C++ utilities that are referenced directly from PepperCode. Do not confuse this discussion with writing substrate or interface utilities.

Following are general guidelines for writing a C++ utility to use in your PepperCode code. Although you can use these guidelines in any order, you could follow them sequentially as you design a C++ utility.

Checking That A Corresponding Function Is Not Defined

Before writing a new C++ function for use in PepperCode, make sure that a corresponding function has not already been defined.

Most of the existing C++ function declarations can be found in files that begin with `cpp_` in the `scheduler/spl/` directory. Also, make sure that a PepperCode intrinsic operator or function does not exist for the function you need.



For more information about the PepperCode intrinsic functions and operators, refer to [Understanding Infix and Intrinsic Operators and Functions](#).

Putting C++ Code In The Proper Location

Put the C++ code in the proper location.

If you are writing a complex utility involving a C++ class that must be referenced from PepperCode, use the following files as an example:

```
scheduler/utls/intersector.h

scheduler/utls/intersector.cc
```

```
scheduler/utils/intersector_access.h
scheduler/utils/intersector_access.cc
scheduler/spl/cpp_intersect.spl
```

For example, to add a new C++ sorting utility function for the scheduler module, you would do the following:

- place your function definitions (the body of the code) in `scheduler/utils/cpp_spl_sort.cc`, and
- place the top-level function signature (the declaration, showing only the function name and its parameters) in `scheduler/utils/cpp_spl_sort.h`.

Capitalizing C++ Function Names

The declared name for a C++ function should always be in all capital letters.

This: `cpp_function void PRINTF (string) "printf";`

Not this: `cpp_function void printf (string) "printf";`

And not this: `cpp_function void Printf (string) "printf";`

Providing Meaningful PepperCode Types

Provide meaningful PepperCode types in the C++ function declaration.



For more information about the types, see [Typedefs Used With C++ Functions](#).

The types provided in the C++ function declaration should correspond to the types in the actual C++ function. Currently, the PepperCode compiler does not check these types at compile time.

For example, the `FLOAT_TO_INT` function takes one float as an argument and returns the float as an integer. If you look at the declaration, the input argument types and return types are obvious.

This:

```
cpp_function int FLOAT_TO_INT (float) "rps_float_to_int";
```

Not this:

```
cpp_function float FLOAT_TO_INT (int) "rps_float_to_int";
cpp_function void FLOAT_TO_INT (string) "rps_float_to_int";
cpp_function time FLOAT_TO_INT (date) "rps_float_to_int";
```

Using RPS_IMPORT When Defining External C++ Functions

You must use the RPS_IMPORT macro when you define external C++ functions in order to allow for system patchability.

When you put the definition of an external C++ function "myfunc" into a file "mydir/myfile.cc", do the following:

- Put an external declaration or "prototype" into a file "myfile.h", and always put '#include "mydir/myfile.h"' into "myfile.cc". (The same rule applies to external C++ variables.)
- Put the macro RPS_INCLUDE in front of a prototype or variable declaration (if you declare a class, however, put it after the word "class").
- Always put '#include "mydir/rps_import_def.h"' into "mydir/myfile.h" as the last "#include" statement.
- If "myfile.h" or "myfile.cc" needs to include other files, put the inclusions from other directories first; then the ones from directory "mydir" last. Use the appropriate directory name in each #include statement.

Here is an example of using the RPS_IMPORT macro. This is taken from cpp_spl_dump.h.

```
/* Copyright 1994-1997 by PeopleSoft, Inc. */
/* All U.S. and world rights reserved. */

#ifndef CPP_SPL_DUMP_H
#define CPP_SPL_DUMP_H

#include <utilsCC/cpp_types.h>
#include <scheduler_utils/rps_import_def.h>

RPS_IMPORT CPP_INT cpp_open_dump_file (CPP_STRING filename, CPP_STRING mode);

RPS_IMPORT CPP_INT cpp_close_dump_file ();

RPS_IMPORT void cpp_dump_newlines (CPP_INT number);

RPS_IMPORT void cpp_dump_spaces (CPP_INT number);

RPS_IMPORT void cpp_dump_int (CPP_INT int_to_dump, CPP_INT columns, CPP_INT
right_justified);

RPS_IMPORT void cpp_dump_float (CPP_FLOAT float_to_dump, CPP_INT columns,
CPP_INT right_justified);

RPS_IMPORT void cpp_dump_float_for_export (CPP_FLOAT float_to_dump, CPP_INT
columns,
```

```

    CPP_INT right_justified);

// Write UTF-8 "string_to_dump" to export file using the appropriate export
// locale's character set. No language translation takes place. String is
// left- or right-justified with enough blanks to occupy specified number of
// "columns". If "columns" is less than 1, the string is printed without
// any padding blanks.

RPS_IMPORT void cpp_dump_string (CPP_STRING string_to_dump, CPP_INT columns,
    CPP_INT right_justified);

RPS_IMPORT void cpp_dump_date (CPP_DATE date_to_dump, CPP_INT columns, CPP_INT
right_justified);

RPS_IMPORT void cpp_dump_time (CPP_TIME time_to_dump, CPP_INT columns, CPP_INT
right_justified);

RPS_IMPORT CPP_INT cpp_dump_test_reset_status();

RPS_IMPORT CPP_INT cpp_dump_reset_status();

// Like cpp_dump_string, but uses the translation table to convert the
// "string_to_dump" argument to the local language and character set
// specified by the "export" locale.

RPS_IMPORT void nls_dump_string (CPP_STRING string_to_dump, CPP_INT columns,
    CPP_INT right_justified);

#endif

```

Adding and Retrieving Documentation

PepperCode has a documentation feature which lets you put text into the .spl file for use in generating documentation. The compiler always ignores the text as if it were a comment, but optionally it will write the text to a file named sourcefile.doc.

For a transaction named `transaction_build_bicycle`, the syntax would be:

```
#document transaction_build_bicycle

Your description goes here, and continues

for as many lines as you like.

#end_document transaction_build_bicycle
```

This block of documentation can occur anywhere within the source file.

A -d option causes the compiler to generate the .doc file.

The compiler automatically puts into the .doc file the name of the source file and the line number at which the comment started, so there's no need to put that information inside the comment by hand, where it might easily get out of date if you suddenly need to move some transactions to a different source file.

Using #include Files

Assume there is a custom module "cus" built on top of the standard "mfg" module. It also assumes a standard directory structure of:

```
$RPS_SDK/cus/
$RPS_SDK/cus/spl/           for *.spl files, and
$RPS_SDK/cus/utils/        for *.cc & *.h files
```

where \$RPS_SDK is the partial path from the root of the network file system to the directory where Planning files reside.

To include Planning spl source files from scheduler or mfg in custom module spl files, use angle-brackets:

```
#include <spl/foo.spl>

#include <spl/mfg_foo.spl>
```

To include custom module spl source files in custom module spl files use double quotes:

```
#include "cus_foo.spl"
```

To include custom module C++ source (*.cc) files in custom module spl files, use header files with double quotes:

```
#include "../utils/cus_cpp_foo.h"
```

A custom module C++ source (*.cc) file (for example, cus_cpp_foo.cc) need only include its own header file with double quotes:

```
#include "cus_cpp_foo.h"
```


To include Planning C++ source files in custom module C++ header (*.h) files, include the associated header file using angle-brackets:

```
#include <string.h>

#include <utilsSPL/interface_History.h>

#include <objectcore/rps.h>
```

Finally, and this case should be rare, to include custom module C++ source (*.cc) files in custom module C++ header (*.h) files, include the associated header file using double quotes:

```
#include "cus_cpp_base.h"
```



Note: When placing filenames in double quotes (" ") for Windows NT, you can use a file delimiter of "/" instead of "\".



For more information about using #include, see Writing PepperCode #include Statements.

Customizing and Displaying Class Names

The `Named_Object` class is not built-in, but it is part of the Planning product. `Named_Object` provides a slot called `display_name`, which you set if you want the user to see a more descriptive name for a class than the real name of the class. `display_name` is the name that is displayed to the end user when they display a class; it defaults to the real name of the class.

For example, a transfer option will have from-unit and to-unit information appended to its class name. You might want to just display the name of the transfer option, without the unit information appended to it. Also, a task has appended to its name the build option and the part being built. You might want to have a display name without the build option in it.



Note: In some PepperCode files, “site” is often used instead of “unit”.

Following is the code for `named_object`.

```
Named_Object

class Named_Object: Spl_Class {

    string display_name

    action <set_display_name> set_display_name_action
```

```
};
```

To set `display_name`, you must perform the following tasks:

- Have your class inherit from `Named_Object`.
- Write the `set_display_name` method for that class, and set the `display_name` slot in that method.

Customizing PepperCode Methods And Actions

PepperCode can be customized in several ways, some more ambitious than others. It's best to familiarize yourself with existing facilities before undertaking radical changes.

Since PepperCode is an object-oriented system, much customization involves modifying or adding to the standard objects supplied by the system. Since these changes are made at run time only, the standard definitions remain intact. Some customizing techniques, in rough order of ambitiousness:

- You can modify the values of system slots, thus overriding the local or inherited values. When the relevant slots are methods, the slot values are actions (functions); so replacement values are actions which you must supply. Our first example—refer to “Replacing Standard Method Actions”—shows such replacement of a method slot action, where the purpose is to modify the system's reporting behavior.
- You can add new data slots to the standard ones supplied by the system. New data slots are normally added in new subclasses of existing object classes.
- You can add new method slots to those supplied by the system. Like new data slots, new method slots are often added in new subclasses of existing object classes. If the new action you supply will share parameters with other actions, you can create an action schema to manage the sharing. Our first example—refer to “Adding Method Slots”—shows how. It also shows how to invoke the new method. Our second example extends similar techniques to create a new subclass of constraints (refer to “Adding a Constraint”). It also demonstrates the use of a system-defined C++ function—rather than a specialized transaction—to create instances of the new class.

Replacing Standard Method Actions

The system presently contains a `Dispatch_List` object used to control the printing (or “dumping”) of scheduling reports. This object contains a method slot relevant, `human_dump_action`. The slot's value is the action `dispatch_list_dump`. Here is its code:

```
action<human_dump> dispatch_list_dump
(
    (local: instance<Dispatch_List> dispatch_list)
{
```

```

dispatch_list = object;

DUMP_STRING("Dispatch List For ", 0, 0);

DUMP_STRING(dispatch_list.equipment_resource.name, 0, 0);

DUMP_STRING(" From ", 0, 0);

DUMP_DATE(dispatch_list.start_time, 0, 0);

DUMP_STRING(" To ", 0, 0);

DUMP_DATE(dispatch_list.end_time, 0, 0);

DUMP_NEWLINES(3);

DUMP_SPACES(3);

DUMP_STRING("Production", 40, 0);

DUMP_NEWLINES(1);

DUMP_SPACES(3);

DUMP_STRING("——", 40, 0);

DUMP_NEWLINES(1);

foreach element in dispatch_list.pending_elements {

    DUMP_NEWLINES(1);

    DUMP_SPACES(3);

    execute dump_object(:spl_object element, :human_dump TRUE);

}

succeed();

}

```

Assume that a different format is desired, and that the following action produces it.

```

action<human_dump> special_dispatch_list_dump

(local: instance<Dispatch_List> dispatch_list)

{

    dispatch_list = object;

    DUMP_NEWLINES(3); // This new line adds three blank lines.

    DUMP_STRING("Dispatch List For ", 0, 0);

    DUMP_STRING(dispatch_list.equipment_resource.name, 0, 0);

```

```

...

// From here on, the code is the same as the original.

}

```

The problem, then, is to replace the old action with the new one at run time. There's a standard transaction for this purpose, `transaction_set_action_method`, with arguments `:class_name`—a string, the class object where the slot can be found; `:slot_name`—a string, the slot in which the new value is to be installed; and `:action_name`—a string, the action which is to be installed as the new slot value. So all you have to do is to include the following line in a command file and then arrange to load the file into the system. Loading can be done via the user interface, or via load files.

```

execute transaction_set_action_method (:class_name "Dispatch_List"

:slot_name "human_dump_action"

:action_name "special_dispatch_list_dump").

```

A similar modification could be performed on the slot `machine_dump_action`, which calls the standard action `dispatch_list_mdump`. The “m” in “mdump” is a convention which indicates a format readable by machine, as opposed to humans. Further, the slots `machine_dump_action` and `human_dump_action` are found in the system object `Dispatch_Element` as well as `Dispatch_List`, with respective standard actions `dispatch_element_mdump` and `dispatch_element_dump`.

Adding Method Slots

As you have seen, system behavior can sometimes be modified by replacing existing method actions with new ones. In other cases, however, you modify by creating new method slots with brand new actions. As noted, since slots cannot be added to existing objects at run time, new slots need new classes to hold them. This section demonstrates the second, more ambitious sort of modification.

A previous section demonstrated the creation of a new subclass in order to add a new slot. This example is no different with respect to subclass creation, but since the new slot is a method slot this time, this example also needs to create the action which will become the slot's value.

As background for the exercise, consider the existing class `Routing_Option`. A routing option is a way of obtaining a needed material for some planning Task. One way of obtaining material is to build it. The existing `Build_Option` class represents this possibility.

To accommodate a new method for build option instances which can calculate the cost of the option—the specific goal of the example—the following example adds a new subclass, `Build_Option_CUS`, for “customized” or “customer”, below `Build_Option`. The new subclass will inherit all of the slots of the parent class and add one more, our new method slot. Any instances of `Build_Option_CUS` would inherit all the old slots plus the new one.

The new method slot will be `build_cost_method`. As a default action for the slot, to be passed—all else equal—to objects lower in the inheritance hierarchy, use an action, or function,

`default_build_cost_action`. And since actions, like data objects, can be usefully categorized according to their parameters, the following example also creates a sample schema upon which the new action can be based, `build_cost_schema`. Once all of this machinery is in place and an instance of `Build_Option_CUS` containing the new method has been made, the method must be invoked. For illustration, the following example provides a calling action `atp_cost`.

The following example shows the necessary code in the order required by the compiler: with dependent code following the code it depends on.

As mentioned, this example provides a default action for the new `build_cost_method`. However, a more specialized action may be appropriate for instances of even more specialized classes even lower in the hierarchy. In this case, it is possible to override the default action by replacing it with a specialized action at the lower level. The following example shows a specialized action `build_and_ship_cost_action` which might replace `default_build_cost_action`, as the value of the `build_cost_method` method slot. For brevity, this example doesn't show the definition of the lower subclass or the installation of the code as the new method slot value. The purpose of the specialized action is to include consideration of the shipping cost when calculating the overall cost of a build option.

Here are the steps in detail.

1. Create an action schema `build_cost_schema`.

It expects an input argument object, an instance of the existing class `Routing_Option`. It has an output parameter `cost`.

```
// All Action schema that are designed for use as methods (a slot on an
// object) should have an input argument "object" whose type is the same
// as the object it was designed for (here, Routing_Option).

// Action schema with output parameter cost.

action_schema build_cost_schema

    (input: instance<Routing_Option> object,

    input: string sales_order_name = "",

    input: string site_name = "",

    output: float cost,

    no_context);
```

2. Define a new action `default_build_cost_action` which uses the new `build_cost_schema` action schema and thus inherits its parameters.

The new default action uses the schema's input parameter `object` and its output parameter `cost`. The action also uses the slot `cumm_cost`, which has been defined at the `Routing_Option` level: it returns the value of the `cumm_cost` slot as the cost of an

instance of `Build_Option_CUS`. More elaborate procedures for cost calculation are possible, and can be made to override this default calculation; this will be demonstrated below.

```
// New action using the build_cost_schema action schema and its object and
// cost parameters.

action<build_cost_schema> default_build_cost_action ()
{
    cost = object.cumm_cost;

    succeed();
}
```

3. Define the class `Build_Option_CUS` as a subclass of `Build_Option`, adding a method slot `build_cost_method`. Make the new `default_build_cost_action` the default value of this new method slot.

```
Class Build_Option_CUS : Build_Option {

    action<build_cost_schema> build_cost_method //New method slot.

};

slot Build_Option_CUS.build_cost_method
    {default: default_build_cost_action}; //Method slot gets default value.
```

4. Write an action that invokes the newly defined method. Here, the new calling action `atp_cost` takes as input an instance of the `Build_Option_CUS` class. It gets the value of the `build_cost_method` slot in that instance—an action; makes that value the binding of the local variable `locvar_build_cost_action`; and then executes the action by its local name—within the `if` expression.

```
action atp_cost

    (input: instance <Build_Option> build_option,

     input: string sales_order_name,

     input: string site_name,

     local: instance<Part> part,

     local: action<build_cost_schema> locvar_build_cost_action,

     local: action<dku_part> dku_part_action,

     output: float cost)

{

    // This function will return the build option cost
```

```

// Called by: schedule_atp_option

// Get behavior.

locvar_build_cost_action = build_option.build_cost_action;

part = build_option.part;

dku_part_action = part.dku_part_action;

// Execute behavior.

execute dku_part_action();

if (EQ(dku_part_action.dku_part_flag, TRUE)) {

    execute locvar_build_cost_action(:object build_option,

                                     :sales_order_name sales_order_name,

                                     :site_name site_name);

}

else {

    execute locvar_build_cost_action(:object build_option);

}

// Use what behavior returns.

cost = locvar_build_cost_action.cost;

succeed();

}

```

5. Create a specialized cost-building action, `build_and_ship_cost_action`, which can be used to override the inherited default action, `default_build_cost_action`, in the class `Build_and_Ship_Option`. Neither the class nor the replacement is shown here, however. The purpose of the specialized action is to take shipping cost into account when calculating the cost of a `Build_Option_CUS`.

```

// Given a source (site) and destination (customer region),

// return the freight cost.

action<build_cost_schema> build_and_ship_cost_action

(local: instance<Build_Option_CUS> build_option,

 local: instance<Sales_Order> sales_order,

 local: instance<Customer_Region> cr,

 local: instance<Site> site,

```

```

    local: class<Freight_Cost> fc_class,

    local: oset[instance<Freight_Cost>] FCs)

{
    FCs.flush();    // superstition

    sales_order = GET_INSTANCE_BY_NAME(sales_order_name);

    cr = sales_order.customer_region;

    site = GET_INSTANCE_BY_NAME(site_name);

    fc_class = Freight_Cost;

    GET_DESCENDANTS(FCs, fc_class, 1);

    foreach fc in FCs {

        if ((fc.source == site) &&

            (fc.destination == cr)) {

            // Return freight cost for source/destination shipment.

            cost = fc.cost;

            succeed();

        }

    }

    // else return default value

    cost = 0.0;

    succeed();

}

```

Adding A Constraint

Previous sections have demonstrated (1) replacement at run time of existing method actions in existing classes and (2) creation of a subclass in order to add a method slot containing a new action. In this section the example combines elements of both techniques: it creates a new subclass, but rather than create new method slots, it overrides the values of existing, inherited method slots with new actions. The previous example used new transactions to enable creation of instances of the new class. This time, though, the example uses the prepared C++ function `CREATE_OBJECT` for instance creation.

The new subclass will represent a new type of scheduling constraint. Constraints are, of course, represented as objects in PepperCode, like almost everything else. There's an existing constraint

class, `Milestone_Constraint`, which this example intends to specialize. It's a reparable constraint, meaning that it comes with a repair method as well as an indication of the penalty which is imposed if the constraint is violated. The goal here is to represent a new subtype of milestone constraint, namely that all of the lines on a sales order should ship together.

For this purpose, the example creates a constraint subclass, `Shipset_Milestone_Constraint`. The constraint itself will need four actions:

- to display information about the constraint;
- to define the penalty imposed if the constraint is violated;
- to define the repair action if the constraint is violated;
- and to specify the time interval within which the constraint must be satisfied.

In addition, the example needs two supporting actions:

- to create an instance of a `shipset_milestone_constraint` and make it the value of the `milestone_constraint` slot in a relevant sales order;
- and to delete a sales order having a `shipset_milestone_constraint`, making sure that the latter constraint is deleted as well to prevent hanging pointers.

The following sections describe the creation of the class and the actions discussed above.

Creating the Class `Shipset_Milestone_Constraint`

Create the class `Shipset_Milestone_Constraint` as a subclass of `Milestone_Constraint`. Four existing methods get new actions as default slot values.

```
class Shipset_Milestone_Constraint : Milestone_Constraint {

    oset[date] ship_dates

};

slot Shipset_Milestone_Constraint.display_action

    { default: display_shipset_milestone_constraint class_slot: };

slot Shipset_Milestone_Constraint.penalty_action { default:
shipset_milestone_penalty };

slot Shipset_Milestone_Constraint.repair_action { default:
shipset_milestone_repair };

slot Shipset_Milestone_Constraint.start_and_end_action { default:
shipset_start_and_end

    class_slot: };
}
```

Writing an Action to Display Information

Write an action to display information about the constraint.

```

action<display> display_shipset_milestone_constraint

    (local: instance<Shipset_Milestone_Constraint> milestone_constraint,

    local: instance<Sales_Order_CUS> sales_order,

    local: action<violated> violated_check)

{

    milestone_constraint = object;

    sales_order = milestone_constraint.object;

    execute violated_check(:the_constraint milestone_constraint);

    PRINTF("\n%s", milestone_constraint.class_name);

    PRINTF("\nSales Order:  %s", sales_order.name);

    PRINTF("\nPenalty X Weight:  %lf X  %lf  =  %lf\n",

        violated_check.penalty, violated_check.weight,

        violated_check.penalty_times_weight);

    succeed();

}

```

Writing An Action To Define The Penalty

Write an action to define the penalty imposed if the constraint is violated. Constraint penalties must be normalized to values between 0 and 1.

```

// Loop over all sales order lines and determine the set of unique ship dates
// and their frequency of occurrence.

// No violation means either there are no sales order lines for the sales
// order, or that all lines have the same ship date.

action<penalty> shipset_milestone_penalty

    (local: instance<Shipset_Milestone_Constraint> milestone_constraint,

    local: oset [instance<Sales_Order_Line>] sales_order_lines,

    local: instance<Sales_Order_CUS> sales_order,

```

```

    local: date et,           // end time of shipment as scheduled

    local: date latest_ship,  // latest of all et

    local: date earliest_ship, // earliest of all et

    local: oset[date] ship_dates,

    local: oset[date] dates,

    local: int so_line_count,

    local: string date_inserted_p)
{
    milestone_constraint = the_constraint;

    sales_order = milestone_constraint.object;

    sales_order_lines = sales_order.sales_order_lines;

    so_line_count = sales_order_lines.length();

    if (sales_order_lines.empty() == 1) {

        penalty = 0.0;

        succeed();

    } // Otherwise examine sales order lines.

    // Initialize latest ship date and set of ship dates.

    latest_ship = sales_order_lines.first().shipment.end_time;

    ship_dates.flush();

    ship_dates.push(latest_ship);

    // Determine distinct scheduled sales order line shipment dates.

    foreach sales_order_line in sales_order_lines {

        et = sales_order_line.shipment.end_time;

        if (et == latest_ship) {

            latest_ship = et; // Update latest_ship.

            ship_dates.enqueue(et);

        } // Do next sales_order_line.

        else {

            dates.flush(); // Temporary list

```

```

date_inserted_p = "FALSE";

foreach ship_date in ship_dates {

    if (NOT (date_inserted_p == "TRUE")) {

        if (et < ship_date) {

            // Enque new date on list, add existing date
after.

            dates.enqueue(et);

            dates.enqueue(ship_date);

            date_inserted_p = "TRUE";

        }

        else {

            if (et == ship_date) {

                // Save existing date.

                dates.enqueue(ship_date);

                date_inserted_p = "TRUE";

            }

            else {

                dates.enqueue(ship_date);

            }

        }

    }

    else {

        // Just collect the rest of the ship dates into the
temp list.

        dates.enqueue(ship_date);

    }

} // foreach ship_date

ship_dates = dates;

}

} // foreach sales_order_line

```

```

earliest_ship = ship_dates.first();

latest_ship = ship_dates.last();

if (earliest_ship == latest_ship) {
    penalty = 0.0;
}

else {
    penalty = DIV(SUB(INT_TO_FLOAT(ship_dates.length()), 1.0),
                  so_line_count);

    milestone_constraint.ship_dates = ship_dates;
}

succeed();
}

```

Writing An Action To Specify The Repair

Create an action to specify the repair if the constraint is violated.

```

// Should be in mfg_repair.spl

action<constraint_repair> shipset_milestone_repair

(local: instance<Shipset_Milestone_Constraint> milestone_constraint,

 local: oset [instance<Sales_Order_Line>] sales_order_lines,

 local: instance<Sales_Order_CUS> sales_order,

 local: instance<Routing_Parent> routing_parent,

 local: date et,                // end time of shipment as scheduled

 local: date earliest_ship,     // earliest of all et

 local: date latest_ship,       // latest of all et

 local: date mode_ship,         // most common et

 local: date new_time,          // time to move shipset tasks to

 local: oset[date] ship_dates,

 local: oset[int] ship_date_frequency,

 local: int random_value,

```

```

        local: int index,

        local: int count_index,

        local: int count)
{
    ship_dates.flush();

    ship_date_frequency.flush();

    milestone_constraint = the_constraint;

    ship_dates = milestone_constraint.ship_dates;

    sales_order = milestone_constraint.object;

    sales_order_lines = sales_order.sales_order_lines;

    // Initialize earliest ship and latest ship dates.
    earliest_ship = ship_dates.first();
    latest_ship = ship_dates.last();

    // Determine the most common scheduled sales order line shipment date.
    foreach sdate in ship_dates {
        count = 0;

        foreach sales_order_line in sales_order_lines {
            et = sales_order_line.shipment.end_time;
            if (et == sdate) {
                count = count + 1;
            }
        }

        ship_date_frequency.enqueue(count);
    }

    // Find the max of ship_dates (the most common ship date).
    index = 0;

    count = 0;

    count_index = 0;

    foreach frequency in ship_date_frequency {

```

```
        if (frequency > count) {
            count = frequency;
            count_index = index;
        }

        index = index + 1;
    }

    mode_ship = ship_dates.nth(count_index);

    // Sometimes move all to mode_ship
    // Sometimes move all to latest_ship
    // Sometimes move all to earliest_ship

    MSG(25, "\nShipset Milestone Repair");

    random_value = RANDOM(100);

    if (random_value < 40) {
        new_time = mode_ship;

        MSG(25, "    mode");
    }

    else {
        if (random_value < 70) {
            new_time = latest_ship;

            MSG(25, "    latest");
        }

        else {
            new_time = earliest_ship;

            MSG(25, "    earliest");
        }
    }

    MSG(25, "    moving shipset to:  %s\n", DATE_TO_STRING(new_time));

    foreach sales_order_line in sales_order_lines {
        routing_parent = sales_order_line.shipment;
```

```

        execute reschedule_task(:task routing_parent,
                                :target_time new_time, :st_or_et END_TIME);

        status = reschedule_task.status;

        if (status == SUCCEED) {

            MSG(25, "      S\n");

        }

        else {

            MSG(25, "      F\n");

            fail();

        }

    }

    succeed();

}

```

Writing An Action To Specify The Time Interval

Write an action to specify the time interval within which the constraint must be satisfied.

```

action<start_and_end> shipset_start_and_end

    (local: instance<Environment> environment)

{

    environment = GET_PARENT_ENV();

    start_time = environment.early_fence;

    end_time = environment.late_fence;

    succeed();

}

```

Writing An Action That Creates A Constraint Object

Write an action which creates an instance of a `Shipset_Milestone_Constraint`, making the instance the value of the `milestone_constraint` slot in a sales order. This action will be executed at sales order creation time.

```

// Create a constraint for each Sales_Order at sales order creation time.

```



```
action create_shipset_milestone

  (input: instance<Sales_Order_CUS> sales_order,

   output: instance<Shipset_Milestone_Constraint>

        shipset_milestone_constraint,

   no_context: )

{

  execute create_object(:class_name "Shipset_Milestone_Constraint");

  shipset_milestone_constraint = create_object.new_object;

  shipset_milestone_constraint.object = sales_order;

  sales_order.milestone_constraint = shipset_milestone_constraint;

  succeed();

}
```


CHAPTER 12

Compiling And Linking PepperCode

This section describes how to compile your code and link it with existing Planning software for testing purposes.

The directory that contains the PepperCode compiler is /home/v8vm/product/splcompiler. If you are familiar with the PepperCode Release 7.5 compiler and have written applications using Release 7.5 PepperCode, the following subsections should get you up to speed on the new Release 8.0 compiler. This information was derived from refman.txt, a UNIX flat file that can be found in the same directory that contains the PepperCode compiler, experiments with the compiler, and conferences with Development.

Starting in Release 8.0, you can write “standalone” PepperCode, compile it, then execute it, and you don’t have to use RPSMake or The Project Manager (also known as splsh). This is referred to as standalone mode. To use the Release 8.0 Compiler in standalone mode, you will need a UNIX account on the PeopleSoft San Mateo Office's UNIX network. Your system administrator can help you set up your account.

Setting Up and Using Your Own PepperCode Sandbox

At this point, it is suggested that you create a directory for your PepperCode programs. You can create this directory anywhere on UNIX where you have read, write, and execute permission. However, it is suggested that you use your own UNIX directories (/home/<your name>). Here, your programs and their source files will be easier for you to maintain and control.

1. Create a directory to hold your PepperCode programs and their components.
2. Change directories to the directory that you just created, write the following code using your favorite UNIX text editor, then save it to a file called hello_world.spl in your newly created directory:

```
action spl_main()
{
    PRINTF("Hello, world!\n");
}
```

spl_main is a special action in PepperCode that is automatically executed when the program is run.

3. Remain in your newly created directory and compile hello_world.spl. This will generate the

object file `hello_world.o`. Do so with the following command:

```
> /home/v8vm/product/splcompiler/spl hello_world.spl
```



Note: It is important that you stay and keep all files that the compiler might need in the newly created directory throughout this procedure.

This compilation will generate the following files and place them in your newly created directory:

Files Generated by Default Compilation

File(s)	Description
<code>hello_world.o</code>	object file (used in the next step)
<code>hello_world.pchs</code>	pre-compiled header file. This file is used when including PepperCode source files. For more information, see Writing PepperCode <code>#include</code> Statements.
<code>hello_world.cc</code>	C++ code that is the PepperCode source file translated into C++

4. Link the object file you just created to generate the executable `hello_world` with the following command:

```
> /home/v8vm/product/splcompiler/spl --make_program hello_world hello_world.o
```

5. Now, try your new program by typing the program name.

```
> hello_world
```

You should see the familiar output “Hello, world!” followed by the server output normally seen when starting the Supply Chain server.

The compiler option `--make_program` along with other compiler options are described in PepperCode Compiler Reference.

Running the Compiler

The program “spl” can be used to compile one .spl source file and generate an object file; or to bind one or more object files into a shared library (DLL); or to bind one or more object files and

shared libraries into an executable program. Following is the syntax you use from a Unix command line:

```
./spl options filename
```

filename is the name of the file you want to compile. Only one filename is allowed. If filename ends in .spl, the compiler expects a PepperCode source file. If filename ends in .cc, the compiler expects to compile C++ source from a previous PepperCode file compilation.



For more information about options, see PepperCode Compiler Reference.

Solaris example

To compile one .spl source file and generate an object file called "myfile.o", on Solaris, for example:

```
spl myfile.spl
```

To build a shared library called "libmylibrary.so":

```
spl --make_library libmylibrary.so myfile.o anotherfile.o
```

To build a program called "myprogram" using that library plus another object file:

```
spl --make_program myprogram yetanotherfile.o libmylibrary.so
```

To execute the program, first make sure that your LD_LIBRARY_PATH variable is not set (in the C shell, say "unsetenv LD_LIBRARY_PATH"). Then say:

```
./myprogram
```

On Solaris, by default, the program looks for the PepperCode runtime libraries (often called the "substrate" relative to the directory in which the compiler resides, followed by the current working directory ".". If you wish, you may use the environment variable LD_LIBRARY_PATH to override this, searching first in the directories named by LD_LIBRARY_PATH and then in the compiler's own directories. If you move the runtime libraries to a different location after building the program (the nightly v8vm server build script does this, for example) or if you use non-absolute pathnames for your own libraries and you expect to run the server from a directory other than the current working directory, then you will need to set LD_LIBRARY_PATH to specify a colon-separated list of the directories where the libraries reside.

In the Solaris world, there is only one complexity not shown in the example above. If two compilations a.spl and b.spl both use "#include" on one another, you must first generate header files using the "--no_header" option, then compile the files normally:

```
spl --header_only a.spl
```

```
spl --header_only b.spl
```

```
spl a.spl  
  
spl b.spl
```

(Actually, since there's no need to generate the headers a second time, you could use "--no_header" on the latter two compilations.)

HP-UX example

HP-UX behaves like Solaris, with two exceptions. First, by convention the shared library suffix is ".sl". Second, 32-bit versions of the operating system use SHLIB_PATH rather than LD_LIBRARY_PATH to tell the runtime loader where to find libraries at program-startup time.

Digital Unix (OSF/1) and Linux examples

Digital Unix and Linux behave like Solaris, with one exception: the directories specified by LD_LIBRARY_PATH are searched only after the directories specified by the compiler itself. For most users this works fine (your libraries aren't in the compiler's directories, so searching them first is a harmless waste of time), but programmers maintaining the substrate libraries will need to relink with the "--rt_path" option (described later) instead of relying on LD_LIBRARY_PATH if they wish to override the directories normally specified by the compiler.

NT example

To compile one .spl source file and generate an object file called "myfile.obj" on NT:

```
spl myfile.spl
```

To build from it a program "myprogram":

```
spl --make_program myprogram.exe myfile.obj
```

Using shared libraries on NT is more complicated than on Solaris, for two reasons. First, the code which NT generates to access a symbol generally varies depending on whether the access crosses the boundary between one library and another. Second, NT does not permit mutual dependency between two libraries unless you first create at least one "import library" to describe the interface of one of the libraries. The PepperCode compiler will help handle both problems, but it requires you to use extra command-line options.

Suppose you want to build three mutually dependent libraries a.dll (based on files a0.spl and a1.spl), b.dll (based on b0.spl and b1.spl), and c.dll (based on c0.spl and c1.spl). Then you want to link them together with myprogram.obj to generate program myprogram.exe. We'll describe a general method which extends to an arbitrary number of mutually dependent libraries.

The easy part is to compile myprogram.spl into myprogram.obj as usual:

```
spl myprogram.spl
```

Next, compile all the library source files with `--header_only`, using the `--lib_tag` option to tell each of them which library it will belong to:

```
spl --header_only a0.spl --lib_tag a
spl --header_only a1.spl --lib_tag a
spl --header_only b0.spl --lib_tag b
spl --header_only b1.spl --lib_tag b
spl --header_only c0.spl --lib_tag c
spl --header_only c1.spl --lib_tag c
```

Now compile the files for library a and then build an import library "a.lib" which describes interfaces (this does not build a real shared library--we'll do that later). This step implicitly builds an "export library" called "a.exp" as well:

```
spl --no_header a0.spl --lib_tag a
spl --no_header a1.spl --lib_tag a
spl --make_implib a.lib a0.obj a1.obj
```

Repeat the preceding steps for the other libraries, using `--lib_tag b` and `--lib_tag c` as appropriate:

```
spl --no_header b0.spl --lib_tag b
spl --no_header b1.spl --lib_tag b
spl --make_implib b.lib b0.obj b1.obj
spl --no_header c0.spl --lib_tag c
spl --no_header c1.spl --lib_tag c
spl --make_implib c.lib c0.obj c1.obj
```

Now build the real shared libraries. For each one, specify on the command line the export library corresponding to the DLL being built, the object files for that DLL, and the the import library which describes the other DLLs:

```
spl --make_library a.dll a.exp a0.obj a1.obj b.lib c.lib
spl --make_library b.dll b.exp b0.obj b1.obj a.lib c.lib
spl --make_library c.dll c.exp c0.obj c1.obj a.lib b.lib
```

Finally build the program, specifying the import libraries:

```
spl --make_program myprogram.exe myprogram.obj a.lib b.lib c.lib
```

NT uses the `PATH` variable to tell the runtime loader where to find libraries at program-startup time, so generally you must make sure your `PATH` variable includes the directory in which the

PepperCode compiler resides (because the PepperCode runtime libraries reside there too). Then type:

```
./myprogram
```

Command-line rules in detail

As the examples above show, by default the compiler knows what suffixes are appropriate to the platform on which it is running (for example, on Unix it generates .cc for C++ files and .o for object files, whereas on NT it generates .cpp for C++ files and .obj for object files). You can override this using a command-line option like "--object_suffix", as explained later.

Unlike typical Unix compilers, PepperCode does not use file suffixes to decide what a particular file contains. For example, if you say "spl myfile.obj", the compiler will not automatically decide that because the file ends in ".obj" it must be an object file. Instead, you must explicitly use "--make_program" or "--make_library" to tell the compiler to generate a library or program instead of attempting to read PepperCode source code from "myfile.obj". Also, PepperCode will not take a combination of source and object files in the same command; you must first compile each source file, one at a time, and then link the objects.

Also unlike typical Unix compiler, PepperCode attempts to remove an output file before truncating and writing it, so it will succeed if the file is removable but unwritable; and it puts the object file in the same directory as the source file.

No matter which operating system you are using, a filename or directory name may use either northwest (NT) or northeast (Unix) slashes.

Installation and Configuration Issues

The Release 8.0 Compiler is very flexible. You can choose to "install" the Release 8.0 compiler in your own UNIX directories or use the existing installation at /home/v8vm/product/splcompiler.

Look here if you are installing the Release 8.0 Compiler or you wish to customize it.

Installation and configuration of the compiler is much easier and more compact because PepperCode has been streamlined. Installation and configuration is, however, quite a different procedure in Release 8.0, so these special instructions are provided here.

LD_LIBRARY_PATH



Note: The library path variable is different for each target machine. For example, older HP systems use SHLIB_PATH instead of LD_LIBRARY_PATH; NT provides no default and uses PATH instead of LD_LIBRARY_PATH; etc.

By default, the compiler looks for *.so libraries that it needs in the directory containing the compiler and in the current directory. You may override this behavior by setting the LD_LIBRARY_PATH environment variable. If set, LD_LIBRARY_PATH must be set to a list of directories that includes a directory containing the *.so libraries the compiler needs.

You may wish to set your own LD_LIBRARY_PATH if you move the runtime libraries to a different location after building the program (the nightly v7vm build does this, for example) or if you use non-absolute pathnames for your own libraries and you expect to run the server from a directory other than the current working directory.

LD_LIBRARY_PATH was implemented in this way to allow you more flexibility in your development efforts. It allows you the choice of either using your own custom-made shared libraries or using the default libraries that can be found in the same directory as the compiler.

If you want to execute the program using the default path to the libraries that is set by the compiler, first make sure that your LD_LIBRARY_PATH variable is not set (in the C shell, say "unsetenv LD_LIBRARY_PATH"). Then (assuming that myprogram is your program) say:

```
./myprogram
```

List of Necessary Files

This is the list of files that must be present in the same directory for proper operation of the compiler:

<i>File Name</i>	<i>Description</i>
spl or spl.exe	the executable compiler
.splrc	global configuration file
splrt_stripped.h	declarations required by the C++ source code generated by the compiler
*.so or *.dll	the "substrate" libraries which provide the compiler runtime system
auto_timestamp.o	startup code to be linked into the executable program

The global configuration file .splrc contains a series of command-line flags. When the compiler starts up, it first reads the global configuration file. Then it reads an optional per-user configuration file \$HOME/.splrc (on Unix) or "%USERPROFILE%\Application Data\PeopleSoft\SPL\splrc" (on NT). Finally it reads the flags on the actual command line.

.splrc

This is an optional per-user configuration file. With it, you can customize the behavior of the compiler. Use the options listed below as a guide for modifying .splrc. On UNIX, .splrc will be

located in the \$HOME directory. On Windows NT, it will be located in %USERPROFILE%\Application Data\PeopleSoft\SPL.

It is possible to customize the behavior of the compiler by modifying the global .splrc file. For example, you could change the --cpp_suffix option to use a different suffix for C++ files, or you could change the -cpp_fmt flag to use a different C++ compiler, or you could use --loud to make the compiler treat a particular warning as an error. The individual user can often override the settings in the global .splrc file by specifying different ones in the \$HOME/.splrc file, although flags like "--cpp_fmt" require great care, and flags like "--include" only allow you to add directories to the list, not to remove them. It is intended that the compiler will eventually use the Win32 registry instead of .splrc files when executing on that system. However, this implementation is not planned for Release 8.0.

Compiler Options (For Use During Installation)

The following options are normally put into .splrc and are used for installing the compiler, although you are allowed to specify them on the command line if you wish:

```
--cpp_suffix <suffix>
--object_suffix <suffix>
--dll_suffix <suffix>
--ar_suffix <suffix>
--implib_suffix <suffix>
--explib_suffix <suffix>
--exe_suffix <suffix>
```

These tell the compiler what file suffix to expect for C++ sources, object files, shared library (DLL) files, archive (static) library files, import library files, export library files, and executables. The <suffix> should not include a ".". The compiler sets these to default values, so it's not necessary to specify them in the configuration files unless you want to override the defaults.

On Unix systems which do not use import libraries per se, the "implib_suffix" is normally set to the suffix of static libraries (e.g. "a"). The explib_suffix is left unset, and the exe_suffix is set to the empty string.

```
--cpp_fmt <format>
--cpp_d_fmt <format>
--cpp_o_fmt <format>
--lib_fmt <format>
--lib_d_fmt <format>
--lib_o_fmt <format>
```

```
--implib_fmt <format>

--implib_d_fmt <format>

--implib_o_fmt <format>

--prog_fmt <format>

--prog_d_fmt <format>

--prog_o_fmt <format>
```

These tell the compiler how to run the C++ compiler, how to build a shared library (DLL), how to build an import library, and how to build a program. Each comes in three forms: the compiler uses the "_d_" form when the --debug flag is in effect, and uses the "_o_" form when the --optimize flag is in effect. The format strings use a printf-like syntax, as described earlier in this section.

```
--purify_fmt <format>

--quantify_fmt <format>
```

On Unix systems, these provide prefix formats which we prepend to the appropriate "--prog_*_fmt" string if you use "--purify" or "--quantify" along with "--make_program".

```
--verbose
```

This prints on the console a list of all the flag values in effect. If there are no other flags which call for the compiler to do work, then it exits. Otherwise, the compiler proceeds to do its normal work, and when it issues a command to the operating system (e.g. to run the C++ compiler) it first prints that command on the console.

```
--no_debug

--warn

--no_optimize
```

These are opposites of the usual options, provided only so that one can override a configuration file.

```
--rt_fmt <list>
```

This specifies the list of runtime (substrate) libraries, using whatever syntax the target OS expects. For example, on Solaris, this will probably consist of "-laintpr -lrtoe ...". Note that the string may not begin with "-", so we usually put a blank at the beginning.

```
--oslib_fmt <list>

--oslib_d_fmt <list>
```

Similar to --rt_fmt, this specifies the list of OS or C++ compiler libraries required in building programs. --oslib_d_fmt specifies libraries to use when --debug is in effect.

```
--include_fmt <format>
```

```
--define_fmt <format>
```

This tells how to format each `--include` or `--define` option before passing it along to the target machine C++ compiler. The `<format>` should contain a single `"%s"` to mark the spot where we should substitute the directory or macro definition. If the format would begin with `"-"`, put a blank in front of it (for example, `"-I%s"` or `"-D%s"`).

Each of the `<format>` strings described in the preceding list of options may use the following printf-like codes:

<code>\$* or %*</code>	List of all input files
<code>\$@ or %@</code>	Output file from <code>--make_program</code> , <code>--make_library</code> , <code>--spl_to_object</code> , or <code>--cpp_to_object</code>
<code>\$w or %w</code>	Directory in which the compiler executable resides
<code>\$\$</code>	Substitute a dollar sign
<code>%%</code>	Substitute a percent sign
<code>\$r or %r</code>	Substitute the list of runtime libraries obtained from the <code>--rt_fmt</code> flag
<code>\$l or %l</code>	Substitute the list of OS and C++ compiler libraries obtained from the <code>--oslib_fmt</code> or <code>--oslib_d_fmt</code> flag.
<code>\$i or %i</code>	Substitute all of the <code>--include</code> and <code>--define</code> options, formatted according to the <code>--include_fmt</code> and <code>--define_fmt</code> strings, here.
<code>\$# or %#</code>	Store the list of input files in a temporary file and substitute the name of the temporary file in place of the <code>'\$#'</code> sequence.
<code>\$p or %p</code>	Substitute the value of <code>--rt_path</code>
<code>\$. or %.</code>	Substitute the value of <code>--make_program</code> , <code>--make_library</code> , <code>--spl_to_object</code> , or <code>--cpp_to_object</code> with suffix removed.

PepperCode Compiler Reference

The procedures for using the new compiler are very different from those for using previous versions of the compiler. This section explains options for using the Release 8.0 PepperCode compiler. It provides command line syntax, descriptions of command line syntax, and examples of command line syntax for the 8.0 compiler.

Command-line Rules in Detail

By default the compiler knows what suffixes are appropriate to the platform on which it is running. For example, on Unix it generates .cc for C++ files and .o for object files, whereas on NT it generates .cpp for C++ files and .obj for object files. You can override this using a command-line option like "--object_suffix", as explained in Compiler Options (For Use During Installation).

Unlike some compilers, PepperCode does not use file suffixes to decide what a particular file contains. For example, if you say "spl myfile.obj", the compiler will not automatically decide that because the file ends in ".obj" it must be an object file. Instead, you must explicitly use "--make_program" or "--make_library" to tell the compiler to generate a library or program instead of attempting to read PepperCode source code from "myfile.obj". Also, PepperCode will not take a combination of source and object files in the same command; you must first compile each source file, one at a time, and then link the objects.

Also unlike some compilers, PepperCode attempts to remove an output file before truncating and writing it, so it will succeed if the file is removable but unwritable; and it puts the object file in the same directory as the source file.

No matter which operating system you are using, a filename or directory name may use either northwest (NT) or northeast (UNIX) slashes. The following sections give a list of all of the command-line options, grouped according to their purpose.

Most-Used Compiler Options

These compiler options are used for most compilations.

default (no option switch)

The default generates a single object file from a single PepperCode (*.spl) file. The object file is given the same name as the *.spl file by default.

Usage:

```
> spl <PepperCode source file>
```

Example:

```
> spl myfile.spl
```

This example creates an object file called myfile.o (myfile.obj on Windows NT). The generated file is placed in the same directory as the *.spl file.

--make_program

This compiler option generates an executable program file from one or more object files and libraries.

Usage:

```
> spl --make_program <executable file> <object files> [source libraries]
```

Example:

```
> spl --make_program myprogram myfile.o
```

This example generates an executable program from the object file generated in the <no option> example above.

--make_library

This option tells the compiler to convert one or more object files into a shared library and put the result into file <name>. On Win32 systems it is important that all of those object files were built with the same --lib_tag option.

Usage:

```
> spl --make_library <shared library name> <one or more object files>
```

Example:

```
> spl --make_library libmylibrary.so myfile.o
```

This example makes a shared library libmylibrary.so from myfile.o.

Options That Dictate Which Compiler or Linker to Run

These compiler options let you use the PepperCode compiler as a machine-independent interface to the C++ compiler when you work with human-written C++ code. They are mutually exclusive as the compiler does only one thing at a time. As mentioned earlier, the compiler does only one thing at a time, so the options in this section are mutually exclusive unless otherwise noted.

--spl_to_object <object file name> <PepperCode file name>

This option translates a single *.spl file <PepperCode file name> into an object file called <object file name>. The object file must end with the standard object suffix. There should be exactly two filenames on the command line. One must have the .o or .obj object file suffix, and the other must have the spl suffix. This compiler option is used when you want your object file to have a different name than your *.spl file.

Example:

```
> spl --spl_to_object y.o x.spl
```

This option will generate an object file named y.o from spl file x.spl.

--cpp_to_object <object file name> <cpp file name>

This option translates a single C++ source file <cpp file name> into an object file called <object file name>. The object file must end with the standard object suffix (*.o on UNIX, *.obj on WindowsNT). Instead of running the PepperCode compiler, we run the C++ compiler.

Example:

```
> spl --cpp_to_object y.o x.cc
```

This option will generate an object file named y.o from C++ file x.cc.

--preprocessor

Applies the C++ preprocessor to a single C++ source file, which must end with the standard C++ suffix (not .h), and write the result to the standard output.

-c--***no_object***

Translate a single *.spl file into a C++ source file (.cc or .cpp file) and a pre-compiled header file (.pch file). However, no object file is generated when using this option.

--make_implib <name>

Like --make_library, but for use on Win32 systems which requires you to generate import libraries before generating shared libraries or DLLs. This does nothing on other systems.

--debug***--optimize***

Generate debuggable or optimized code. These are mutually exclusive. They always affect PepperCode compilations and C++ compilations, and on some target machines they affect the creation of libraries and executable programs as well, so it is wise to specify them consistently in every command.

Options Used When Compiling PepperCode

These options are used when compiling PepperCode. They are ignored otherwise.

--include <directory>***-I<directory>***

Add the directory to the list of directories in which we search for "spl" files mentioned in "#include" statements. The path may use either northwest or northeast slashes.

For an example of how this is used, see Rules for Inclusion and Writing #include Statements. As mentioned under Options Used Only When Compiling C++ Source Code, you can also use these

options with `--cpp_to_object`. Note that if you use these options when compiling C++ code, the list of directories gets passed to the C++ compiler. If you use these options when compiling PepperCode, the list of directories gets passed to the PepperCode compiler but not to the C++ compiler, since PepperCode-generated code contains only a single, canned `"#include"` statement anyway.

--no_warn

Suppress printing of any warnings. This takes effect after any `--loud` and `--quiet` options. First the compiler establishes the severity of each message, and then `--no_warn` globally suppresses all messages whose severity is lower than "error".

--loud <integer>

--quiet <integer>

This option raises or lowers the severity of the message specified by `<integer>` (the integer corresponding to a particular warning appears after the word "Warning" when the compiler prints it.) You can use `--loud` to turn a warning into an error, or you can use `--quiet` to suppress a warning (and then you can use `--loud` to turn it back into a warning again).

You can also use `--loud` to turn a normal error into a fatal error, which causes the compiler to quit immediately. You cannot use `--quiet` to lower the severity of an error or fatal error.

Example:

If you issued the following:

```
spl --make_program --quiet 71
```

Message number 71 would not appear. This only applies to warnings. If it is an actual error message, you cannot make it go away in this manner. You will get a compiler message informing you that this is already an error.

--lib_tag <tag>

When compiling PepperCode, this option generates C++ code for the dynamically linked library that is associated with string `<tag>`. This is required only on Win32 systems, although using it on other operating systems is harmless.

Normally the compiler generates C++ code to be built into the main portion of a program, as opposed to a shared library or DLL. If you use this option, it generates code to be linked into a shared library (DLL). The "tag" is a cookie used to distinguish one DLL from another in a situation where one DLL imports symbols from another: it need not match the name of the DLL file, but must be different than the "name" you use for any other DLL.

You should specify this whenever compiling a *.spl file whose code will ultimately become part of a Win32 DLL. In particular it is important to use this option consistently both when compiling with `--header_only` and with `--no_header`.

--header_only

This compiler option generates a precompiled header for this source file, but does not attempt to compile the source file. This is useful when two source files use "#include" on one another. A precompiled header (a binary file whose name ends in ".pch") must exist for each "#include" statement in a source file before you can compile that source file. When two *.spl files include one another directly or indirectly, you must generate the predefined header for one source file, then compile the other, and finally compile the first.

Example:

Say you have two files a.spl and b.spl. They include each other in their source code. If you were to try to compile these files normally with the following command:

```
spl a.spl
```

You would get an error. Because you included b.spl in a.spl, the compiler will look for the file b.pch. It won't be able to find this file because it hasn't been generated yet. The same thing would happen if you tried to compile b.spl. To avoid this error, you must generate a.pch and b.pch before you actually compile. The following is an example of how you would proceed:

```
spl --header_only a.spl //generates a.pch (needed to generate b.o)

spl --header_only b.spl //generates b.pch, (needed to generate a.o)

spl --no_header a.spl //generates a.o using b.pch (generated above)

spl --no_header b.spl //generates b.o using a.pch (generated above)
```

Actually, you could compile with no option on the latter two compilations. The no_header option is being used here because you have already generated header files for a.spl and b.spl in the first two compilations.

--no_header

If you use this option, you will not generate a precompiled header. When building a program from scratch, you may find it easier to process all the files first with --header_only, then compile them all with --no_header.

For an example of this, see header_only in this section.

--doc

This compiler option generates a *.doc file as specified in the PepperCode Documentation Comments section. To generate only a *.doc file for a particular *.spl file, use the --header_only and --no_header compiler options in conjunction with this compiler option.

Example:

To generate the file xyz.doc without actually compiling xyz.spl:

```
> spl --no_header --header_only --doc xyz.spl
```

To generate the file xyz.doc along with compiling xyz.spl (to an object file):

```
> spl --doc xyz.spl
```



For more information on using documentation comments and #document blocks, see Writing PepperCode Documentation Comments.

Options Used Only When Compiling C++ Source Code

You may use the PepperCode compiler as a machine-independent interface to the C++ compiler.



For more information, see Options That Dictate Which Compiler or Linker to Run.

--define <macroname>=<value>

Define the specified macro. This option is ignored if you are compiling a PepperCode source file, since the PepperCode language itself does not perform macro substitutions. Omitting the "*=<value>*" portion sets the macro to "1".

--include <directory>

-I<directory>

Add the directory to the list of directories in which we search for files mentioned in "#include" statements. The path may use either northwest or northeast slashes.



For more information, see Options Used When Compiling PepperCode.

Options to be Used With --make_program Option

These options are used with the --make_program compiler option. They are ignored otherwise.

--client

Build a program which lets clients call actions via TCP/IP networking. By default, networking is disabled, so to call actions at runtime you must either create a main-program action called "spl_main" (which will be invoked automatically at startup whether or not you built the program with --client) or you must run the program with the "-I" option, which prompts for action invocations on the keyboard.

--no_main

Disable automatic generation of a main program. This is useful when employing the PepperCode compiler as a front end to hide the syntax of the local C++ compiler while linking human-written C++ code which provides its own "main". In the absence of this option, we generate a main program which sets a build stamp and invokes certain startup functions in the "substrate" (PepperCode runtime) libraries.

--no_rt

Disable linking of the "substrate" (PepperCode runtime) libraries. This is useful when employing the PepperCode compiler as a front end to hide the syntax of the local C++ compiler while linking human-written C++ code which does not depend on those libraries. Typically *--no_rt* requires *--no_main*, since the automatically generated main program calls functions in those libraries.

--rt_path <directory>

Adds <directory> to the link-time search list for substrate (runtime) libraries.

--purify***--quantify***

Use these options along with *--make_program* and a list of object files and libraries to prepare a version of the program which checks the heap (*--purify*) or generates a performance profile (*--quantify*).

Machine-Specific Escape Clause

--quote <argument>

This option passes <argument> to the C++ compiler or linker without interpreting it. This may appear repeatedly. For example, you could say:

```
--quote "-O parallel"
```

to pass "-O parallel" to the C++ compiler or linker. All of the *--quote* options are passed to the compiler or linker prior to the first input file name.

Options for Compiler Maintenance

These options are meant to be used only by those that maintain the PepperCode compiler.

--keep
-K

Do not delete temporary files and output files in case of error.

--parse_dump

Generate on stdout a diagnostic listing of the syntax tree after parsing but before reading precompiled headers corresponding to "#include" statements.

--include_dump

Like "--parse_dump", but after reading precompiled headers.

--check_dump

Like "--parse_dump", but after semantic checking.

--pch_dump

Instead of reading the source file, read the corresponding precompiled header (which must already exist) and print a diagnostic listing on stdout.



For more information about some additional options, see Compiler Options (For Use During Installation).

Using Hush

The Hush feature allows you to remove the executable code within the actions from a PepperCode source file, while leaving the interface intact to allow other source files to include the Hushed source file. You use Hush to protect proprietary PepperCode code.

Hush can perform the following tasks:

- Replaces the PepperCode source file, *filename.spl*, with a PepperCode source file that has the executable code within the actions deleted. This does not delete the entire action; it removes the code within the {}, leaving the rest of the action intact. The write-permission bits are cleared on this file to prevent editing of the source file and recompiling of the object file by the make command.
- Replaces the C++ source file, *filename.cc*, with a zero-length non-writable file that has the same modification time as the C++ source file. This conceals the code and prevents recompiling of the object file by the make command.

Following is the syntax you use from a UNIX command line:

```
hush options filename
```

options can be one or more of the following:

-outfile newfile	Do not replace the PepperCode source file, <i>filename.spl</i> . Instead, put the PepperCode source file with the deleted actions into a file named <i>newfile</i> .
-keepcc	Do not replace the C++ source file, <i>filename.cc</i> .
-help	Print a help message for the Hush command.

For example, if you want to share interfaces and object files, but not proprietary code, do the following:

1. Use the make command to make the directory normally.
2. Using csh, enter the following commands:

```
foreach x (*.spl)
    hush $x
end
```

If you were to perform Hush upon the sample code in the section Getting Started with PepperCode, the code would look as follows:



For more information, including a listing of the sample code, see Getting Started with PepperCode.

```
// Include the .spl file containing PepperCode runtime functions.

// By convention, these functions appear in all uppercase letters in code.
#include "cpp_utility.spl"

// Create an enumeration containing the possible bike materials.
enum material { steel, aluminum, carbon_fiber, titanium, other };

// Define a basic class for a vehicle.
class Vehicle : Base_Class {
    int serial_number
    int passengers
```

```

        int price
    };

    slot Vehicle.passengers { default: 4 };

    // Derive the class Bicycle from the class Vehicle.

    // Add two new slots, in addition to those from the Vehicle class.

    // Override the default number of passengers to a more realistic value
    // for a bike.

    class Bicycle: Vehicle {

        string model_name

        enum<material> frame_material

    };

    slot Bicycle.frame_material{ default: steel };

    slot Bicycle.passengers{ default: 1 };

    // Create an instance of the Bicycle class (or one of its subclasses).

    // The instance of the class is an object.

    action create_vehicle

        (input: int serial_number,

         input: string model_name,

         input: string class_name,

         output: instance<Bicycle> new_bike,

         no_context:)

    {

    }

    // Create instances of the classes Atb and Bicycle.

    // Test it by generating a list of the instances of Bicycle and iterating
    // through the list-printing the serial number of each Bicycle or Atb.

    action spl_main

        (input: int argc,

         input: oset[string] argv,

```

```
    input: string identity,  
    local: action<create_vehicle> create_vehicle,  
    local: oset[instance<Bicycle>] list)  
{  
}
```


CHAPTER 13

Understanding PepperCode Syntax

The following grammar describes the syntax recognized by the parser in the current PepperCode compiler. You will notice a few constructs that are not documented elsewhere in the manual. Some of these are recognized by the parser but then are rejected later in the compiler; they represent possible future enhancements to the language. Others are accepted by the compiler; they represent obsolete features that are still accepted.

The most notable example of the latter involves trailing semicolons. For historical reasons, the compiler accepts extra semicolons after right braces (}) in certain places where C and C++ do not require them—for example, after the left brace ({) at the end of an action definition.

```
<spl_statements>

 ::= <spl_statement> <spl_statements>

 | <spl_statement>

<spl_statement>

 ::= <toplevel_statement>

<toplevel_statement>

 ::= <class_statement>

 | <slot_statement>

 | <action_statement>

 | <enum_statement>

 | <action_scheme_statement>

 | <function_statement>

 | <function_scheme_statement>

 | <cpp_fn_statement>

 | <cpp_fn_scheme_statement>

 | <directive_statement>

<directive_statement>

 ::= <include_directive>
```

```

<include_directive>

    ::= #include <filespec>

    | #remote_include <filespec>

<filespec>

    ::= < <filename> >

    | " <filename> "

<keyword_parameter_list>

    ::= ( )

    | ( <keyword_parameter_list_recur> )

<keyword_parameter_list_recur>

    ::= <keyword_parameter_declaration>

    | <keyword_parameter_list_recur> , <keyword_parameter_declaration>

<keyword_parameter_declaration>

    ::= <keyword_parameter_type> <type_specifier> <identifier>

    <keyword_parameter_initial_value>

    | <keyword_parameter_attribute>

<keyword_parameter_type>

    ::= input:

    | inout:

    | local:

    | output:

    | returntype:

<keyword_parameter_attribute>

    ::= explain:

    | interpret:

    | audit:

    | audit_no_replay:

    | no_context:

    | context:

```

```

<keyword_parameter_initial_value>

    ::= = <constant>

    | = <identifier>

    | <empty>

<parameter_list>

    ::= ( )

    | ( <parameter_list_recur> )

<parameter_list_recur>

    ::= <parameter_declaration>

    | <parameter_list_recur> , <parameter_declaration>

<parameter_declaration>

    ::= <type_specifier>

<class_statement>

    ::= class <identifier> <base_classes> <class_body_semi>

    | <CLASS> <identifier> ;

<class_body_semi>

    ::= <class_body> ;

<class_body>

    ::= { <slot_specs> <class_specs> }

    | { <class_specs> }

    | { <slot_specs> }

    | { }

<base_classes>

    ::= <empty>

    | : <base_class_id>

    | <base_classes> <base_class_id>

<base_class_id>

    ::= <identifier>

<slot_specs>

```

```

    ::= <slot_spec>
    | <slot_specs> <slot_spec>
<slot_spec>
    ::= <slot_type> <identifier>
<slot_type>
    ::= <type_specifier>
<class_specs>
    ::= <class_spec>
    | <class_spec> <class_specs>
<class_spec>
    ::= <after_init_spec>
    | <class_interface_value>
    | <temporary_instances>
<after_init_spec>
    ::= after_init: <compound_after_init_statements>
<class_interface_value>
    ::= class_interface_value: <identifier>
<temporary_instances>
    ::= temporary_instances:
<compound_after_init_statements>
    ::= { <after_init_statements> }
    | { }
<after_init_statements>
    ::= <after_init_statements> <after_init_statement>
    | <after_init_statement>
<after_init_statement>
    ::= <if_statement>
    | <compound_after_init_statements>
    | <execute_statement>

```

```

    | <while_statement>

    | <assignment_statement>

    | <expression_statement>

    | <foreach_statement>

<enum_statement>

    ::= enum <identifier> { <enumerator_list> } ;

<enumerator_list>

    ::= <identifier>

    | <enumerator_list> , <identifier>

<slot_statement>

    ::= slot <identifier> . <identifier> <slot_clause_body> ;

<slot_clause_body>

    ::= { }

    | { <slot_clauses> }

<slot_clauses>

    ::= <slot_clauses> <slot_clause>

    | <slot_clause>

<slot_clause>

    ::= <slot_default_clause>

    | <slot_db_clause>

    | <slot_documentation_clause>

    | <slot_cardinality_clause>

    | <slot_interface_type>

    | <slot_class_slot_clause>

<slot_default_clause>

    ::= default: <slot_default_specifier>

<slot_default_specifier>

    ::= <constant>

    | <identifier>

```

```

    | <type_specifier>

    | new

    | new <identifier>

<slot_db_clause>

    ::= db: { }

<slot_documentation_clause>

    ::= documentation: <string_constant>

<slot_cardinality_clause>

    ::= cardinality: <constant>

<slot_interface_type>

    ::= slot_interface_value: <identifier>

<slot_class_slot_clause>

    ::= class_slot:

<cpp_fn_statement>

    ::= cpp_function <type_specifier> <identifier> <parameter_list>

        <string_constant> ;

    | cpp_function < <identifier> > <identifier> <string_constant> ;

<cpp_fn_scheme_statement>

    ::= cpp_function_schema <type_specifier> <identifier> <parameter_list> ;

<action_statement>

    ::= <action_declaration> ;

    | <action_declaration> <keyword_parameter_list> ;

    | <action_declaration> <action_body_semi>

    | <action_declaration> <keyword_parameter_list> <action_body_semi>

<action_declaration>

    ::= action <identifier>

    | action < <identifier> > <identifier>

<action_scheme_statement>

    ::= <action_scheme_declaration> ;

```

```

    | <action_scheme_declaration> <keyword_parameter_list> ;
<action_scheme_declaration>
    ::= action_schema <identifier>
<action_body_semi>
    ::= <compound_action_body_statements>
    | <compound_action_body_statements> ;
<compound_action_body_statements>
    ::= { <action_body_statements> }
    | { }
<action_body_statements>
    ::= <action_body_statements> <action_body_statement>
    | <action_body_statement>
<action_body_statement>
    ::= <if_statement>
    | <action_body_semi>
    | <execute_statement>
    | <while_statement>
    | <exit_statement>
    | <assignment_statement>
    | <expression_statement>
    | <foreach_statement>
<target_statement>
    ::= <action_body_statement>
<if_statement>
    ::= <if_simple_statement>
    | <if_simple_statement> else <target_statement>
<if_simple_statement>
    ::= if ( <expression> ) <target_statement>
<execute_statement>

```

```

    ::= execute <first_variable> <action_keyword_list> ;

<foreach_statement>

    ::= foreach <identifier> in <expression> <target_statement>
    | foreach <identifier> in reverse <expression> <target_statement>

<return_statement>

    ::= return ( <identifier> ) ;
    | return ( <int_constant> ) ;
    | return ( <float_constant> ) ;
    | return ( <string_constant> ) ;

<exit_statement>

    ::= leave ;
    | succeed ( <first_variable> ) ;
    | succeed ( ) ;
    | fail ( <first_variable> ) ;
    | fail ( ) ;

<while_statement>

    ::= <WHILE> ( <expression> ) <target_statement>

<assignment_statement>

    ::= <expression> = <expression> ;

<expression_statement>

    ::= <expression> ;

<expression>

    ::= <constant>
    | <identifier> <path>

<path>

    ::= <link> ( <arg_list>
    | <link>

<link>

    ::= <path> . <identifier>

```



```

    | <empty>
<arg_list>
    ::= )
    | <multi_arg_list> )
<multi_arg_list>
    ::= <multi_arg_list> , <expression>
    | <expression>
<first_variable>
    ::= <identifier>
<action_keyword_list>
    ::= ( <keyword_list> )
    | ( )
<keyword_list>
    ::= <keyword_list> , <keyword_value>
    | <keyword_value>
<keyword_value>
    ::= : <identifier> <expression>
<type_specifier>
    ::= int
    | void
    | float
    | date
    | time
    | string
    | <instance_type_specifier>
    | <class_type_specifier>
    | <function_type_specifier>
    | <cpp_fn_type_specifier>
    | <action_type_specifier>

```

```

    | <oset_type_specifier>

    | <enum_type_specifier>

    | <history_type_specifier>
<instance_type_specifier>
    ::= instance < <identifier> >

<class_type_specifier>
    ::= class < <identifier> >

<cpp_fn_type_specifier>
    ::= cpp_function < <identifier> >

<action_type_specifier>
    ::= action < <identifier> >

<enum_type_specifier>
    ::= enum < <identifier> >

<oset_type_specifier>
    ::= oset [ <type_specifier> ]

<history_element_type_specifier>
    ::= int

    | float

    | string

    | instance

<history_type_specifier>
    ::= history < <history_element_type_specifier> >

<constant>
    ::= <string_constant>

    | <int_constant>

    | <float_constant>

    | - <int_constant>

    | - <float_constant>

<identifier>

```

```
::= [A-Za-z] [A-Za-z0-9_]*  
  
<int_constant>  
  
::= [0-9]+  
  
<string_constant>  
  
::= "[^"]*"; recognizes the same backslash escape characters as ISO C,  
including '\\"'.  
  
<float_constant>  
  
::= [0-9].[0-9]+  
  
<filename>  
  
:: recognizes the syntax of a Unix filename
```


CHAPTER 14

Debugging PepperCode

PepperCode code is converted to C++ code during compilation:

- Actions become C++ instances.
- Parameters become C++ member variables.

To debug PepperCode code, there are a number of tools, including symbolic debuggers for C++ and the debugging functions that you can use to print methods and their descriptions. This section describes these and other methods of debugging your code.

Avoiding Common Mistakes

Some common mistakes programmers make when writing PepperCode code are the following:

- Omitting opening or closing parentheses (()) or braces ({ })
- Adding extraneous semicolons (;)

PepperCode doesn't accept semicolons (;) in all of the places that C and C++ do. The main exception is within a class definition:

```
class c {  
  
    int i // no semicolon here  
  
    float f // or here  
  
};
```

- Not including the proper header files or forward declarations

Be sure to include declaration or action files that define needed actions.

- Not declaring an action as a local parameter within the calling action

PepperCode requires you to declare an action as a local parameter before you can invoke it. The local parameter should be in the form:

```
action<action_name> param_name  
  
not:
```

```
action<schema_name> param_name
```

The tricky part is that in code the syntax looks very similar, so you could accidentally specify a schema, as in the following example:

```
action_schema draw();

action<draw> draw_circle() { ... }

action<draw> draw_square() { ... }

action draw_something

(local: action<draw> draw_circle)

{

    // Will fail at execution time because "draw" is a schema,
    // not an action whose schema is draw. draw_circle is just a
    // parameter and doesn't refer to an action definition.

    execute draw_circle();

    succeed();

}
```

If the action has one or more parameters, the compiler may be able to detect the problem at compilation time. In this example, it will complain that size isn't a parameter in draw. (If the compiler doesn't detect the error, this problem results in a segmentation violation at execution time.)

```
action_schema draw();

action<draw> draw_circle

    (input: int size)

{ ... }

action<draw> draw_square

    (input: int size)

{ ... }

action draw_something

    (local: action<draw> draw_circle)

{

    execute draw_circle(:size 5); // Compiler issues an error message

    succeed();

}
```

```
}
```

The correct version of `draw_something` specifies that the local parameter isn't just an action whose schema is `draw`, but is the specific action `draw_circle`:

```
action draw_something
    (local: action<draw_circle> draw_circle)
{
    execute draw_circle(:size 5);
    succeed();
}
```

- Forgetting to reset static action parameters when an action is called multiple times



Even local parameters must be reset. For more information, refer to the discussion on static action parameters in [Writing Action Parameters](#).

- Not assigning a value to an instance input parameter of an action

This will cause the action to break when executed because the value of the instance parameter will be zero. For example:

```
action evaluate_score
    (input: instance<Spl_Class> an_object,
     local: int current_score)
{
    current_score = an_object.score;    // will break when an_object = 0
    //
    // do some other stuff here
    //
    succeed();
}

...

execute evaluate_score();    // OOPS!!! This will break
```

- Calling methods directly, instead of calling their corresponding dispatcher actions
- Assuming that expressions are evaluated from left to right (like C/C++)

The following code will break when the `_object` is zero. In C/C++, the expression stops evaluating when it hits `the_object`; in PepperCode, the expression continues to evaluate `(the_object.score, 100)` as well. When `the_object` is zero, this code crashes in PepperCode, but not in C/C++.

```
if (AND (the_object,
        EQ (the_object.score, 100)))
```

- Calling C++ functions when corresponding actions exist

For example, `CREATE_OBJECT` is a C++ function that should never be called directly. Instead, the action `create_object` should be used.

Also, `DELETE_OBJECT` should only be called from within delete methods. The action `delete_object` is the top-level call for deleting an object. This is important because every PepperCode object has a default delete method that is called if you use `delete_object`.

- Using `GET_DESCENDANTS` on objects that are in a free store

When an object is in a free store, it should not be accessed with `GET_DESCENDANTS`. One example is spreadsheet row objects.

- Accessing an empty oset

When an oset is empty, any of the oset functions that return values will break. For example:

```
...
local: oset[int] scores,          // local oset parameter
...
scores.flush();                  // empty the oset
scores.first();                   // This will break !!!
scores.last();                    // so will this !!!
...
```

- Modifying an oset while looping over it with `foreach`

The `foreach` statement loops over an oset by incrementing the link-list pointers of the oset. When the link-list pointers are modified during a `foreach`, the system can break with a “stale pointer” error. Note that `foreach` loops over the actual oset, not a copy of the oset. Here is an example:

```
// This action is an example of how NOT to modify osets in a foreach.
//
action bad_oset_usage
```



```

(local: oset[int] scores,
 no_context:)
{
  scores.flush();    // clear the oset

  scores.enqueue(1); // enqueue three numbers on the oset
  scores.enqueue(2); // ...
  scores.enqueue(3); // ...

  foreach score in scores
    if (score == 2)
      scores.delete(score);    // THIS WILL CAUSE THE SYSTEM TO
BREAK

  foreach score in scores
    if (score == 2)
      scores.enqueue(score);    // THIS WILL CAUSE THE SYSTEM TO BREAK

  succeed();
}

// This action is an example of how to modify osets in a foreach
// by making a copy of the oset.
//
action better_oset_usage
  (local: oset[int] scores,
   local: oset[int] temp_scores,
   no_context:)
{
  scores.flush();    // clear the oset

  scores.enqueue(1);    // enqueue three numbers on the oset
  scores.enqueue(2);    // ...
  scores.enqueue(3);    // ...

  temp_scores = scores;    // make a copy of the original oset

  foreach score in temp_scores // loop over a copy of the original oset

```

```

        if (score == 2)

            scores.delete(score);

foreach score in temp_scores    // loop over a copy of the original oset

    if (score == 2)

        scores.enqueue(score);

succeed();

}

```

To avoid this problem when deleting an entire oset of objects in a delete method, use the action `delete_object_list`.

- Not checking for errors resulting from the limited type checking PepperCode does

Troubleshooting Guide

This section is meant to help developers resolve errors and other problems encountered in the development process. It contains a Frequently Asked Questions (FAQ) section and a PepperCode Error Message Reference. The individual FAQ questions and error messages have links to other sections of this document that pertain to the particular question or error message.

Compiler Frequently Asked Questions (FAQ)

The following are frequently asked questions whose answers may help you in your transition to the Release 8.0 compiler:

Q: How does one compile PepperCode files that `#include` each other?

The pre-compiled header files (*.pch) are used when *.spl files are included.



For more information on inclusion and examples that demonstrate how inclusion works, see Writing PepperCode `#include` Statements.

Pre-compiled header files are generated when *.spl files are compiled. The following example is provided for use in cases where two *.spl files include each other.

You must include a *.spl file in a file you are compiling if you use actions or classes that are defined in it.

Example:

If two compilations a.spl and b.spl both use `"#include"` on one another, you must first generate header files using the `"--header_only"` compiler option, then compile the files normally or

compile them using the `--no_header` option. If you were to try to compile these files to an object file without first generating header files for them, the compiler would look for the header file and issue an error because of the missing file (See error message Error on "%s": %s..). The following is an example of how you would proceed:

```
spl --header_only a.spl //generates a.pchs, cannot create a.o yet
spl --header_only b.spl //generates b.pchs, cannot create b.o yet
spl --no_header a.spl //generates a.o using a.pchs (generated above)
spl --no_header b.spl //generates b.o using b.pchs (generated above)
```

Actually, you could compile with the default option on the latter two compilations. The `no_header` option is being used here because you have already generated header files for `a.spl` and `b.spl` in the first two compilations.

To see a functional example of this type of compilation, see `--header_only`.

Q: Why doesn't an enumeration constant have an integer value?

A: The same constant can appear in multiple enumerations, but it may not be possible to assign the same integer constant in each one. If the following statements appear in separate compilations, `green` would naturally be "2" in one case and "3" in the other. An expression like `"green.integer"` would not tell the compiler which "green" was intended.

```
enum rgb { red, blue, green };
enum rainbow { red, orange, yellow, green, indigo, violet };
```

Q: The rules have changed for declaring actions locally. What about classes?

As long as you declare an action, you can invoke it without declaring a local parameter:

```
action b();

action a() {
    execute b();
}
```

instead of:

```
action b();

action a(local: action<b> b) {
    execute b();
}
```

A: The simple answer is yes. You can use an existing class as it is named without declaring it locally or using GET_CLASS_BY_NAME to retrieve it. In fact, you cannot declare a local class as follows:

```
action a(local: class<b> b) {<body>}
```

This will cause an error.

If, however, you want to use a variable to represent the class or use an instance of the class, you must declare it in the action.



For more information, see Using Predefined Classes. For more information on the new rules for local actions, see New Rule for Invoking Action.

Example:

```
// The following are class definitions. (No parameters are specified.)

class c {

};

class c1:c {

};

class c2:c {

}

action a ()

{

    PRINTF("%s", c.name)           // This action prints the name of the class.

    // The class need not be declared locally.

}

action b (local: class(c) cv,

local: int foo)

{

    if foo = 1
```

```

        cv = c

    else if foo = 2

        cv = c1

    else if foo = 3

        cv = c2;

    endif

    PRINTF("%s", cv.name)          // This action prints the name of the class.

                                   // The name of the class depends on foo.

// For this to work,

// the class needs to be declared locally.

// if foo = 1, the result is c.

                                   // if foo = 2, the result is c1.

                                   // if foo = 3, the result is c2.

}

```

Error Message Reference

This section provides troubleshooting information for error messages that you may encounter while compiling PepperCode. They are separated into two categories, errors that stop compilation and warnings that don't stop compilation. They are then listed in alphabetical order in each section.

Errors (That Stop Compilation)

You must resolve these errors before you continue.

Action should have a parameter list.

This error can occur in an action definition, a forward action declaration, or an action_schema definition when you don't use a parameter list.

Example:

Say you declare an action schema as follows:

```
action_schema schema1;
```

Because the parentheses are missing from the action schema declaration, the compiler determines that the action parameter list, which should be enclosed in the parenthesis, is missing. When an

action is defined later with this action schema, the compiler cannot find the parameter list for the action schema because of the missing parenthesis, and this error message is generated.

An action schema should be defined as follows:

```
action_schema schema1(<parameter list>);
```

The actual parameter list is not required, but the parentheses are required.



For more information, see Action Schema Declarations and Definitions.

Cannot set local parameter "%s" in an execute statement.

This error occurs if you try to set a parameter in an execute statement that is local to the action being called.

Example:

The following line of code will cause this error if "l" is a local parameter in action_local:

```
execute action_local(:l 5);
```

Declaration of "%s" conflicts with the declaration in %s:%d.

This can happen when you have more than one action or class definition. The first "%s" is the action or class in question. The second "%s" is the file name of the file containing the conflicting declaration. The "%d" is the line number of the line of code where the conflict occurs.

Example:

Declaration of "uses_schema1" conflicts with the declaration in schema_example_3.spl:8.

This happens if the forward action declaration for action uses_schema1 has a body. If a forward action declaration has a body, the compiler cannot tell which declaration is the forward declaration and which one is the definition. Try taking the body off of your forward action declaration.



For more information, see Declaring Actions: Forward (or Incomplete) Action Declarations.

Deleting output files and stopping due to an error in the compiler itself

This error can be caused by a variety of serious code problems that cause the compiler to stop.

Example:

In a sample run, this error was caused by the following command:

```
PRINTF("%d", action_output.o);
```

This error occurred because of an attempt to print something that did not yet exist. `action_output.o` did not yet exist because `action_output` had not yet been evoked. This same command caused no problem after `action_output` was invoked because `action_output.o` then existed.

Error on "%s": %s.

This is a generic error. It is used to pass error messages from a variety of sources to the PepperCode compiler. The first string ("%s") represents the object that caused the error. The second string (%s) represents the actual error.

Example:

Error on `"/xyz.pchs"`: No such file or directory.

This error occurs when the compiler cannot find `xyz.pchs`. The cause is either because the file has not been generated, or the directory containing `xyz.pchs` is not in the include path. For more information, see Writing PepperCode `#include` Statements and the error message No such file or directory...

fatal: "%s": open failed: No such file or directory

The compiler could not find a component necessary for compilation. If the error message refers to a `*.so` file (PepperCode Library File), then your `LD_LIBRARY_PATH` environment variable has not been properly set.

Example:

```
fatal: libshello.so: open failed: No such file or directory
```

It couldn't find library `libshello.so` because the directory containing it was not specified in `LD_LIBRARY_PATH`.

If you are a previous user of Supply Chain products, your `LD_LIBRARY_PATH` may be set up for use with Release 7.5. If you temporarily unset your `LD_LIBRARY_PATH` by entering the command `"unsetenv LD_LIBRARY_PATH"`, the Release 8.0 default library path (`./home/v8vm/product/splcompiler`) will be used. In most cases, this will resolve the error. However, if you wish to use your own components that are not in the default `LD_LIBRARY_PATH`, you must reset your `LD_LIBRARY_PATH` to include `/home/v8vm/product/splcompiler` and all other directories containing the components that you want to use. You can do this temporarily from the command line or permanently in your `.cshrc` file with the command `"setenv LD_LIBRARY_PATH <path containing components & default *.so libraries>"`.

Example:

The following will set `LD_LIBRARY_PATH` to the default:

```
> setenv LD_LIBRARY_PATH ./home/v8vm/product/splcompiler
```



For more information, see LD_LIBRARY_PATH.

LNK4049: locally defined symbol ""struct spl_action_info spl_action_info_xyz"" imported



Note: This error message applies to Windows NT only.

When you use "#include" to import the PepperCode source file which declares action "xyz", the compiler and the project manager cooperate to decide whether the action appears a different shared library (DLL) than the file you are currently compiling, and emits appropriate code. In that case, this message does not appear.

But if, instead of using "#include", you rely on an incomplete declaration like "action xyz();", or if you use "default: xyz" to set the default value of a slot without ever declaring the action, then the compiler must be conservative and guess that the action might be imported from some other DLL. The message from the Microsoft linker says that the action really appears in the same DLL as the code which refers to the action, and therefore the compiler did not need to be so conservative.

Mismatch between "#document %s" at line %d and "#end_document %s".

The transaction name specified after #document does not match the transaction name specified after #end_document. These transaction names must match.

Example:

If the transaction names were misspelled as follows:

```
#document transaction_foo
...
#end_document transaction_who
```

The following error would occur:

```
Mismatch between "#document transaction_foo" at line 6 and "#end_document
transaction_who".
```

Mismatch in parameter "%s" (see %s:%d).

The parameter lists do not match up with regard to the first "%s". This error can be caused by

- Misspelling parameter names
- Forward action declaration parameter lists that don't match the action definition parameter lists

The first "%s" is, of course, the parameter that doesn't match. The second "%s" is the file in which the parameter was mismatched, and "%d" is the line of the code in which the error was found.

Example:

Mismatch in parameter "s" (see schema_example_5.spl:6).

This particular error was caused by a forward action declaration parameter list that didn't match the action definition parameter list. Forward action declarations are no longer recommended because the compiler now resolved inclusions in a more efficient manner.



For more information, see Writing PepperCode #include Statements.

Check your forward action declarations and make sure that the parameters match those listed in the action definition, or you could include the file containing the action definition. Either of these solutions should resolve this error without causing other errors.

Missing transaction name after "%s".

This error occurs if you forget a transaction name. The string ("%s") tells you where you forgot it.

Example:

If you forget to put the transaction name after the #document statement in a #document block, you will get the following error:

```
Missing transaction name after "#document".
```

No such file or directory.

The compiler couldn't find the file or directory specified in your code. This message is usually part of a larger error message. Example: Error on "./xyz.pchs": No such file or directory. For more information and other examples see the following:

- Error on "%s": %s..
- fatal: "%s": open failed: No such file or directory

Not found: <cpp function> (<type>*, <type>*)

This error usually occurs at link time when you declare a cpp_function but do not describe the arguments in precisely the same way the C++ code does.

Example:

Say that the following cpp function was declared:

```
cpp_function int STRING_COMPARE(string, string) "nlstrcmp";
```

This will cause the following error:

```
Not found: nlstrcmp(char*, char*)
```

In the case of the STRING_COMPARE function, the C++ code declares the arguments using "const char *", so you must add "const:" to the cpp_function statement in PepperCode as follows:

```
cpp_function int STRING_COMPARE(const: string, const: string) "nlstrcmp";
```

As an alternative to using the STRING_COMPARE function, you could use the operators "==", "<", ">", "<=", or ">=", and the compiler will generate the appropriate C++ function call for you. For more information on making declarations for and using C++ Functions see Accessing C/C++ Functions.

Nothing named "%s" is in scope here.

This error will occur if you try to use an object ("%s" in the error message above) that hasn't been defined with regard to scope. It can occur if you have defined the PepperCode object in a local scope, then try to use it outside the local scope.

To correct the error, define the object that you are trying to use for the scope in which you are trying to use it. If the object is defined in another source file (*.spl), be sure to #include that source file in the source file where the object is being used. If the object is defined in a local scope, i.e. a local variable, try defining it globally by declaring it in an action or class. For more information on scope definitions, see Understanding Scopes and Identifiers.

Depending on your situation, you may also receive parse errors and missing type name errors. In the case of C++ functions, you must first define the C++ function with a cpp_function declaration (see Accessing C/C++ Functions). If you do not, the compiler will not recognize the C++ function. This can potentially cause parse errors and missing type errors along with this error. The only exceptions to this rule are the C++ functions that are already included in the substrate. Declaring them will cause a warning, except in the case of the BREAK and CONTINUE functions.

In the case of the BREAK and CONTINUE functions, you cannot declare them with cpp functions. To correct the error (in the case of BREAK or CONTINUE only), remove the cpp function declarations for BREAK and CONTINUE.

```
parse error
cpp_function void CONTINUE() "continue";
```

^

You may also see one or more of the following type of error messages:

```
jim.spl:1: Missing type name (e. g. "int", "class<>", etc.)
```

BREAK and CONTINUE are now built into the compiler as keywords, so the old trick of declaring them as cpp_functions no longer works. For more information on making declarations for and using C++ Functions see Accessing C/C++ Functions.

Parameter "%s" should be "output:" or "inout:", not "%s:"

You are attempting to perform an operation with a parameter that cannot be performed with the parameter. It can only be performed with an "output" or "inout" parameter.

Example:

The following error occurred because of an attempt to print the value of a local variable that was from another action:

```
Parameter "l" should be "output:" or "inout:", not "local:"
```

parse error

You will get this error if the compiler cannot parse your code. It will not be able to parse your code if it encounters something that is incompatible with the compiler's rules. Of course, the error message always indicates the point in the code where the compiler stopped parsing your code. This is the best indication of the error location. Parse errors are usually due to a syntax problems or a definition problems.

Example:

```
parse error

Copyright 1994-1998 by PeopleSoft, Inc.

^

hello_world.spl:4: Missing "#notice" for this "#end_notice" statement.

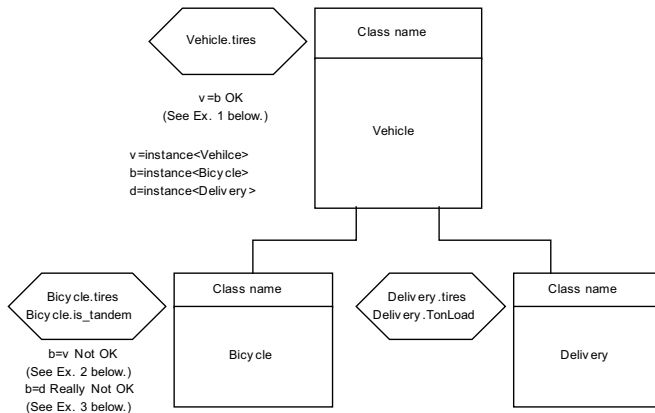
Compilation failed with 2 error(s).
```

In this example, the #notice block was not correct, and the compiler was reading characters that it didn't understand, code that should have been commented out.

SPL: <class_x> is not a subclass of <class_y>

When the source and target class (or instance) of an assignment do not have a proper parent/child relationship, the Release 8.0 Compiler generates a warning at compilation time, and generates code to check the relationship at runtime. If the relationship still isn't correct, the runtime code prints this message: SPL: <class_x> is not a subclass of <class_y>.

The following is an illustration of a typical class/subclass error scenario:



Class/Subclass Diagram

The above Class/Subclass Diagram refers to the following examples:

```
enum Boolean_Flag { TRUE, FALSE };
```

```
class Vehicle {
    int tires;
};
```

```
class Bicycle: Vehicle {
    enum<Boolean_Flag> is_tandem;
};
```

```
class Delivery: Vehicle {
    enum<Boolean_Flag> TonLoad;
};
```

The code in the following examples refers to the code and diagram above.

Example 1: Assigning a Child Class Type Variable to a Parent Type Variable

The following assignment is safe because the only slot we can reference using "v" is "tires", which does exist on "b":

```
action ok(input: instance<Bicycle> b, output: instance<Vehicle> v)
{
    v = b;
}
```

Example 2: Assigning a Parent Class Type Variable to a Child Type Variable

The following assignment is unsafe because a vehicle, such as a delivery van, does not have a slot called "is_tandem", but once we assign "v" to "b", the compiler cannot prevent us from saying "b.is_tandem" and accessing a nonexistent slot. It might be safe if "v" is actually a bicycle (e.g. if we got the value of "v" by calling action "ok" originally), but the compiler can't tell whether the value of "v" originally came from a bicycle, a delivery vehicle, or something else.

Neither the compiler nor the linker can tell whether this is safe or not, so at runtime (provided you used --debug and not --optimize when you compiled) the generated code will check for the unsafe case and issue this error message "SPL: <class1> is not a subclass of <class2>".

```
action not_ok(input: instance<Vehicle> v, output: instance<Bicycle> b)
{
    b = v;
}
```

Example 3: Assigning a Class Type Variable to an Unrelated Class Type Variable

The following assignment is always unsafe because a vehicle, such as a delivery van, has very few slots in common with a bicycle. It could never have a slot called "is_tandem", and bicycle could never have a slot called TonLoad. However, once we assign "d" to "b", the compiler cannot prevent us from saying "b.is_tandem" and accessing a nonexistent slot. Since "d," a delivery van, cannot be a bicycle, it can never be safe, and the compiler can't tell where the value of "d" originally came from.

```
action really_not_ok(input: instance<Delivery> d, output: instance<Bicycle> b)
{
    b = d;
}
```

Neither the compiler nor the linker can tell whether this is safe or not, so at runtime (provided you used --debug and not --optimize when you compiled) the generated code will check for the unsafe case and issue this error message "SPL: <class1> is not a subclass of <class2>".

Remember that "Base_Class" works well as a generic data type, because every class is a child of Base_Class, whereas not all classes are related to "Spl_Class" or "Named_Object".

Target of "%s" is not an lvalue.

This error occurs because of an attempt to make an assignment that is invalid. You can assign values only to expressions that are valid lvalues. To determine which values are valid lvalues, see Writing Assignment Statements.

Example:

If you examine the following code, you will see that an action is called, then an attempt is made to assign a value to the called action's input value.

```
execute action_input(:i 5);    // Caller can pass actual arg in invocation

action_input.i = 6;           // ERROR (caller cannot alter this)
```

Since an input value cannot be a valid lvalue, the following error occurs.

```
Target of "=" is not an lvalue.
```

For more information, see Writing Action Parameters.

The declaration of "%s" is incomplete.

If %s is a class, this error has occurred because of the improper use of a forward (or incomplete) class declaration. Forward class declarations are still allowed to maintain backward compatibility with previous versions, but they can only be used in the same limited manner as they were used in previous releases of PepperCode. For example, when a class is declared only with a forward class declaration in a particular *.spl file, you cannot change or even refer to the slots of that class. You can only declare an instance of that class.

To correct this error, you can either #include the *.spl file that contains the class definition (specified by %s in the error message) or remove the code that caused the error. For more information on forward class declarations and to see examples of the correct and incorrect ways of using them, see Forward Class Declarations.

Undefined symbol: spl_action_info_abc

The linker issues this error. It occurs if you are using a forward action declaration in lieu of including the *.spl file that contains the action definition, and you misspell the action name in the action definition. If you misspell the action name in the forward action declaration you will get an error in the compiler itself (See Deleting output files and stopping due to an error in the compiler itself.)

Example:

The file columns.spl contains the action definition for print_three_columns, and user.spl contains a forward action declaration for print_three_columns, so user.spl can execute the action. If you misspell "print_three_columns" in columns.spl, you will receive the error (from the linker):

```
Undefined                                first referenced

symbol                                in file

spl_action_print_three_columns(spl_action_io*) user.o

ld: fatal: Symbol referencing errors. No output written to user

spl: Command "/disk/u423/compilers/SparcWorks/SUNWspro/bin/CC" failed with
status 256
```

For historical reasons, the compiler allows you to declare actions by saying "action abc();" instead of using "#include" to include the PepperCode source file which defines action "abc". Thus it cannot check for misspellings or missing definitions; the linker will be the first to discover these.

Unterminated string literal.

This error is usually caused by a syntax problem. It occurs if you forget to add a closing quotation mark when specifying a string.

Example:

```
s = "string;
```

will cause this error because there is nothing to tell the compiler where "string" ends. The following is the correct way to make the assignment:

```
s = "string";
```

You must supply a value for "required:" parameter "%s"

The required: keyword is new in Release 8.0. You received this error because you did not assign a value to required: parameter "%s". A parameter becomes required when it is assigned a default of required: as in the following example:

```
action counter(input: int quantity = required: )
{
    PRINTF("%d\n", quantity);
}
```

```
action spl_main()
{
    execute counter();
}
```

Executing this program causes the following error:

```
You must supply a value for "required:" parameter "quantity"
```

If you assign a value to quantity in your action call, as in the following example, the error goes away:

```
action counter(input: int quantity = required: )
{
    PRINTF("%d\n", quantity);
}
```

```
action spl_main()
{
    execute counter(:quantity 15);
}
```

Warnings (These Don't Stop Compilation)

Warnings won't stop you, but they could be the precursor to or an indicator of more serious problems.

Assignment to "input:" variable "%s".

This warning message will occur if an assignment is made to an input variable in a called function. The called function can alter an input variable, but the alteration doesn't have any effect on the calling function. For more information on what data can be passed and where, see Writing Action Parameters.

cpp_function "%s" ignored because it conflicts with a built-in function.

This is a warning message that is caused by making a `cpp_function` declaration for a function that is already built into the code. (The function name will appear in the place of `%s`.) Starting in Release 8.0, many PepperCode C++ functions are pre-defined. This is one of them.

No longer necessary to include C++ files ending in .h

Including `*.h` files is an obsolete practice and is no longer a valid action in Release 8.0. If `*.h` files are included, this warning message is generated. Their inclusion has no other effect but to cause this warning message. Starting in Release 8.0, the `*.h` libraries are built into the substrate, so they no longer have to be included with a `#include` statement. For more information, see Writing PepperCode `#include` Statements.

Source file should have a "#notice" statement.

This warning will occur if you have not included a notice statement in the `*.spl` file that you are trying to compile.

You can eliminate this error by placing a notice statement block in each of your `*.spl` source files. The following is the `#notice` statement block that is used for `*.spl` files:

```
#notice

Copyright 1994-1998 by Peoplesoft, Inc.

All U.S. and World rights reserved.

#end_notice
```

Using Debugging Tools

Since symbolic debuggers don't understand the PepperCode language, you will encounter some limitations when using them to debug PepperCode programs. However, you can use a combination of techniques to make debugging possible.

Using The Action Interpreter

As mentioned earlier, PepperCode has a runtime system that provides an Action Interpreter, which is code that reads a string containing a human-readable action invocation that is similar to the syntax you use in a PepperCode execute statement. The Action Interpreter parses the string, invokes the action, and returns a string containing a human-readable list of output values.

The command line interface to the Action Interpreter executes PepperCode actions that you enter at the prompt. For example, if you have the following action:

```
action compare

    (input: int i,
     input: int j,
     output: int difference)

{
    difference = i - j;
    if (difference == 0)
        fail();
    succeed();
}
```

You could invoke compare from another action:

```
action another

    (local: action<compare> compare_proc)

{
    execute compare_proc(:i 5, :j 6);
    if (compare_proc.status == FAIL)
        fail();
    succeed();
}
```

Or you could invoke compare from the Action Interpreter directly through a command line:

```
compare(:i 5 :j 6)
```

Notice how the comma and semicolon are removed.



For more information, see the process outlined in Compiling And Linking PepperCode. For more information about output from the Action Interpreter, see Writing Methods and Writing Osets.

To start your server in action interpreter mode, allowing you to use action interpreter commands, enter the following command:

```
./server -I
```

In the window where you are running the server, you will get the following prompt, showing that you can now enter an action interpreter command:

```
Enter an action call:
```

To exit the action interpreter, enter the following command:

```
:exit
```

To start the action interpreter from your client, have the Client and Server running and command files loaded. Then perform the following steps:

1. From the Client, select **Help, Tech Support**.

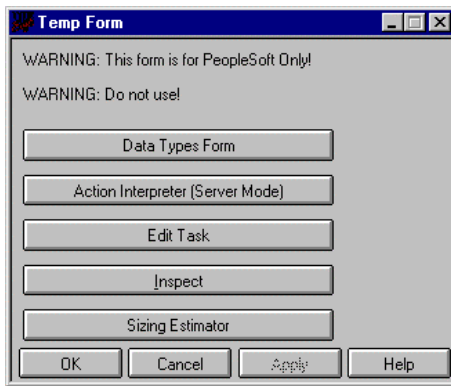
The Technical Support form appears.



Technical Support Form

2. Click on the pepper in the center of the "Powered by Red Pepper" logo.

The Temp form appears.



Temp Form

3. Click on Action Interpreter.

In the window where you are running the server, you will get the following prompt:

Enter an action call:

Now you can enter an action interpreter command.

To exit the action interpreter, enter the following command, then click **OK** on the Temp Form.

```
:exit
```

Using Action Debug Tracing

Action debug tracing helps PepperCode programmers debug problems by tracing the values of action inputs on entry and action outputs on exit.

Action debug tracing lets you trace a particular action and all of its descendants. Tracing begins when you first execute the action, and ends when the function returns. In the meantime, tracing affects every action which you invoke directly from PepperCode code (rather than by using the action interpreter to parse a string containing an action invocation).

On entry to the action, the trace prints a line similar to this on the server console window—C++ file descriptor stderr—showing the values of all “input:” and “inout:” parameters:

```
>my_first_action: (one_input: string "abc")
               (another_input: float 6.5)
```

On return from the action, the trace prints a line similar to this, showing the values of all “output:” and “inout:” parameters. The return status is shown as “F” (fail), “S” (succeed), or “L” (leave) in parentheses after the action name:

```
<my_first_action (S): (one_output: oset[int]: list 3 100 200 300)
```

When one action calls another, the “>” and “<” characters repeat to represent nesting via indentation:

```

01>my_first_action: (one_input: string "abc")
    (another_input: float 6.5)
02>child_of_first: (child_input: instance<Base_Class>
    oid(337 "Detailed_Equipment_Constraint"))
03>grandchild_of_first: (grandchild_input: date 8073996)
03<grandchild_of_first (S):
02<child_of_first (S): (child_output: class<Base_Class>
    oid (445 "Deconflict_Env"))
02>child_of_first: (child_input: instance<Base_Class>
    oid(337 "Detailed_Equipment_Constraint"))
02<child_of_first (S): (child_output: class<Base_Class>
    oid (445 "Deconflict_Env"))
01<my_first_action (S): (one_output: oset[int]: list 3 100 200 300)

```

You can ask to trace many different actions, but only the first action you invoke will actually enable tracing, and, when it returns, disable tracing. While tracing is on, if the program invokes a different action which you have also asked to trace, or if it invokes recursively the action which originally enabled tracing, that has no effect on the state of tracing.

Using The Action Debug Tracing Transaction and C++ Function

Action Debug Tracing adds a single transaction:

```
action action_debug_trace(input: string action_name = 0, input: int enable = 1)
```

Action Debug Tracing also provides a C++ function which you can call from the command line of a debugger if you have compiled the program with debugging information and the debugger is capable of invoking functions from the command line:

```
void action_debug_trace(const char *action_name, int enable)
```

Setting Action Debug Tracing Behavior

By setting the `action_debug_trace` input parameters set in the following ways, you can cause the following behavior. If “enable” isn’t specified, it defaults to 1.

- `action_name` is set to an action name, “enable” is set to a nonzero value: Enable tracing when the action is invoked.
- `action_name` is set to an action name, “enable” is set to zero: Don’t enable tracing when the action is invoked. This doesn’t tell the action to disable tracing when invoked; this restores the

action to the default state, allowing you to invoke the action without starting the trace facility.

- `action_name` is set to a nil or zero-length value, “enable” is set to a nonzero value: Disable action tracing immediately for the current action being traced, and turn it back on the next time an action is invoked that has action debug tracing enabled. In other words, using "" as the `action_name` causes the trace now in progress to stop prematurely, but doesn't clear the "enable" flags which you have turned on with "action_debug_trace" in the past.



Note: If the server is busy executing an action, it won't stop and prompt on the console for another action until it is finished. However, if you ran the server inside a debugger, you can use the debugger to interrupt the server, and then execute the C++ function from the debugger's user interface.

- `action_name` is nil or zero-length, “enable” is set to zero: disables tracing immediately. The next action that is invoked will turn tracing back on, even if that action doesn't have action debug tracing enabled. In other words, a new trace is started the next time you invoke any action, without actually turning on the "enable" flag associated with that action.
- `action_name` is set to a nonexistent action name: An error message is printed on the sever window:

```
action_debug_trace: no such action "my_first_action"
```

Enabling and Disabling Action Debug Tracing

There are several ways to enter the command to enable and disable action debug tracing. To enable, set the enable parameter to a non-zero value; to disable, set it to zero. Some examples of entering the command are:

- Type the command at the command line prompt in the server console window. Typing the following will cause tracing on the transaction `transaction_create_inventory_part`:

```
action_debug_trace(:action_name "transaction_create_inventory_part")
```

- Invoke the `action_debug_trace` action within a command file.
- Invoke the `action_debug_trace` action using an “execute” statement inside a .spl file.
- Call the C++ function `action_debug_trace` from a debugger command line.

When `action_debug_trace` begins a trace, a confirmation message appears on the server console window, so that if you view a log file later on, it is clear that human interventions took place:

```
action_debug_trace: +my_first_action

action_debug_trace: -my_first_action

action_debug_trace: +

action_debug_trace: -
```

Understanding Action Debug Tracing Output

This example started a server and client. Then it selected **Help, Tech Support** in the client to get to the page where **Action Interpreter** was clicked to start the action interpreter.

```
Enter an action call: action_debug_trace(:action_name
"transaction_create_inventory_part")

action_debug_trace: +transaction_create_inventory_part

Enter an action call: :exit
```

This call to `action_debug_trace` will trace the transaction `transaction_create_inventory_part`. One way to use that transaction, and therefore trace it, is to add an item.

In this example, click **Browse**, right-click on **item**, select **add** in the popup menu, in the dialog box type "pedal" as the name, then click **Apply**. The following trace listing occurs.

```
01>transaction_create_inventory_part:

( site_name: string: "" )

( part_name: string: "pedal" )

( class_name: string: "" )

( description: string: "" )

( uom: string: "" )

( planner_code: string: "" )

( mps_type: int: 0 )

( buyer_code: string: "" )

( default_production_area_name: string: "" )

( on_hand: float: 0 )

( configurable: int: 0 )

( aggregate_demand_flag: int: 0 )

( quantity_precision: int: 0 )

( cost: float: 0 )

( unit_price: float: 0 )

( cost_of_goods: float: 0 )

( weight: float: 0 )

( volume: float: 0 )

( inventory_cost_per_day: float: 0 )
```

```
( demand_time_fence: time: 0 )

( consume_sales: int: 1 )

( consume_production: int: 0 )

( consume_transfers: int: 0 )

02> create_inventory_part:

  ( part_name: string: "pedal" )

  ( class_name: string: "Inventory_Part" )

  ( description: string: "" )

  ( uom: string: "Each" )

  ( planner_code: string: "" )

  ( buyer_code: string: "" )

  ( mps_type: int: 0 )

  ( configurable: int: 0 )

  ( quantity_precision: int: 0 )

  ( consume_sales: int: 1 )

  ( consume_production: int: 0 )

  ( consume_transfers: int: 0 )

  ( aggregate_demand_flag: int: 0 )

  ( demand_time_fence: time: 0 )

  ( initial_amount: float: 0 )

  ( cost: float: 0 )

  ( weight: float: 0 )

  ( volume: float: 0 )

  ( unit_price: float: 0 )

  ( cost_of_goods: float: 0 )

  ( inventory_cost_per_day: float: 0 )

  ( site: instance<Base_Class>: oid(1136) )

  ( production_area: instance<Base_Class>: oid(5) )

03> create_base_part:
```

```

    ( part_name: string: "pedal" )

    ( class_name: string: "Inventory_Part" )

    ( site: instance<Base_Class>: oid(1136) )

    ( initial_amount: float: 0 )

    ( description: string: "" )

    ( uom: string: "Each" )

    ( planner_code: string: "" )

04> create_resource:

    ( resource_name: string: "pedal" )

    ( class_name: string: "Inventory_Part" )

    ( site: instance<Base_Class>: oid(1136) )

    ( initial_amount: float: 0 )

05> create_object:

    ( object_name: string: "SM_pedal" )

    ( class_name: string: "Inventory_Part" )

05< create_object(S):

    ( new_object: instance<Base_Class>: oid(2509) )

05> set_object_display_name:

    ( object: instance<Base_Class>: oid(2509) )

06> default_set_resource_display_name:

    ( object: instance<Base_Class>: oid(2509) )

06< default_set_resource_display_name(S):

05< set_object_display_name(S):

04< create_resource(S):

    ( new_resource: instance<Base_Class>: oid(2509) )

03< create_base_part(S):

    ( new_part: instance<Base_Class>: oid(2509) )

```



```

03> encode_part_consumption_flags:

    ( sales: int: 1 )

    ( transfer: int: 0 )

    ( production: int: 0 )

03< encode_part_consumption_flags(S):

    ( consumption_code: int: 4 )

02< create_inventory_part(S):

    ( new_part: instance<Base_Class>: oid(2509) )

    ( output_part_production_area: instance<Base_Class>: oid(5) )

    ( exit_msg: string: " " )

```

Creating Debug Messages With The MSG Function

You can place the C++ MSG function in your code and use it to print debugging information. Following is the cpp definition of the function:

```
cpp_function int MSG (int, string) "dmsg";
```

Afterward, you can use the function in your code:

```
MSG (level, message(s));
```

This function uses the same rules as the printf function, except that the messages appear only if a global debugging level threshold is set to be greater than or equal to the level specified by the first argument.

The PepperCode compiler automatically creates code that calls the MSG function so that a message is printed whenever an action is executed; the messages print whenever the global debug level is at least 50.

The Planning software has a Preferences menu item called Debug Level that lets you interactively specify the level. You can also use the GET_MSG_LEVEL function—as described in “C/C++ Function Access”—to get the current level, and the transaction_set_debug_level transaction to change the level to a new value and return the old value.



For more information about GET_MSG_LEVEL, refer to Accessing C/C++ Functions.

The system will accept any integer value as a debug level. However, only levels 0 through 50 have meaning. Following are the debug levels that have significant meaning:

Level	Meaning
0	Turns off debugging messages.
1	Reports information about serious PepperCode errors right before a crash. Level 1 is the default debug level for the PepperCode system.
2	Reports failures in basic PepperCode actions.
25	Reports what is happening in the system from a functional point of view. Messages at this level should not be verbose. Instead, they should be simple messages that explain what the system is doing.
26 to 49	Reports “how” the “what” is happening. Messages at this level are detailed programmer messages.
50	Reports the name of an action before the action is executed.

When writing PepperCode code, you should provide debugging messages that you can control with the debug level—especially if there are situations where a symbolic debugger isn’t available. For example, the following MSG statement prints a message when the debug level is 25:

```
MSG(25, "\nMoving production task %s to %s\n",
    production_task.name, DATE_TO_STRING(new_production_time));
```

Here are some more examples of good debugging messages:

```
// The software could break because of a missing routing step
//
MSG(1, "\nIn map_bor_entries...routing step %d not found in parent %s\n",
    bor_entry.routing_step, routing_parent.name);
// The software could break because an illegal value was returned.
//
MSG(1, "Warning: timeval returned zero value for item %s on resource %s.\n",
    part.name, resource.name);
// No production can be created when task classes are omitted
//
```

```

MSG(2, "\nIn create_routing_children...No task classes for build option %s.
    build_option.name);

// Could not enforce a hard temporal constraint between two tasks.
//
MSG(2, "\nCould Not Enforce Temporal Constraint between  %s  and %s.\n",
    task1.name, task2.name);

// Display the name of the production being created.
//
MSG(25, "\nCreating Production  %s:", new_routing_parent.name);
// Display the name of the sales order line being deleted.
//
MSG(25, "\nDeleting Sales Order Line  (%s  %d)",
    sales_order_line.sales_order.name, sales_order_line.line_number);
// Display why a task failed in the build query.
//
MSG(25, "\nBuild Query window exceeded for task  %s.\n", routing_task.name);
// Display stuff that interests the programmer (for debugging only).
//
MSG(26, "\nIn calculate_duration_from_quantity:  Calculated duration is %f\,
    duration);
// Display stuff that interests the programmer (for debugging only).
//
MSG(30, "\n new_score = %f  current_score = %f  delta_score = %f\n",
    new_score, current_score, delta_score);

```

Using Debugging Functions

The following functions can be helpful when you are debugging your code. You can execute them with a symbolic debugger, such as dbx, or declare them with a `cpp_function` statement and use them in your code.

In general, the functions can be used with any symbolic debugger, but the exact command you type to invoke them varies. For example, with dbx, you normally use print to invoke a function; with the xdb debugger the command is p.

describe functions return the following line:

```
(class_name uid object_name)
```

An object name of Anonymous means the object isn't a named object.

The describe functions display IS to refer to an instance slot and CS to refer to a class slot.

Following are some variable definitions for the functions:

Variable	Description
rps_verbose	1 is verbose; zero means that slots that are lists are not printed.
uid	A unique integer identifier. Certain functions return the uid so you can use it with other functions.

The rest of this section is a list of the debugging functions you can use.

describe

Description: Prints the slot values of a PepperCode object, including the name and uid of the object

Syntax: void describe

To run describe with dbx on Solaris, reference either the object name or address:

```
call describe((void*)0x1234,1 // 0x1234 is the object address
```

```
call describe(imp_arg0,1) // imp_arg0 is the object name
```

describe_all

Description: Calls the describe function on all PepperCode objects of a given class

Syntax: void describe_all (char *class_name, int rps_verbose)

Transaction:

```
action<transaction> transaction_describe_all (input: string class_name = "",
input: int verbose = 1)
```

Debugger Example:

```
(debugger) call describe_all("Vendor", 1)
```

(Vendor 144 Memory_Is_Us)

name	IS: Memory_Is_Us
editor_class_name	CS: Spl_Class_Form
class_string_id	CS: none
init_action	CS: default_init
delete_action	CS: default_delete
display_action	CS: default_display
machine_dump_action	CS: default_machine_dump
human_dump_action	CS: default_human_dump
compare_dump_action	CS: default_compare_dump
class_interface_value	IS: -1
temporary_instances	IS: 0

(Vendor 145 ACME_Chassis_Company)

name	IS: ACME_Chassis_Company
editor_class_name	CS: Spl_Class_Form
class_string_id	CS: none
init_action	CS: default_init
delete_action	CS: default_delete
display_action	CS: default_display
machine_dump_action	CS: default_machine_dump
human_dump_action	CS: default_human_dump
compare_dump_action	CS: default_compare_dump
class_interface_value	IS: -1
temporary_instances	IS: 0

(Vendor 146 Joes_Monitors)

name	IS: Joes_Monitors
------	-------------------

```

editor_class_name      CS: Spl_Class_Form
class_string_id        CS: none
init_action            CS: default_init
delete_action          CS: default_delete
display_action         CS: default_display
machine_dump_action    CS: default_machine_dump
human_dump_action      CS: default_human_dump
compare_dump_action    CS: default_compare_dump
class_interface_value  IS: -1
temporary_instances    IS: 0

```

(Vendor 147 We_Sell_Everything)

```

name                  IS: We_Sell_Everything
editor_class_name      CS: Spl_Class_Form
class_string_id        CS: none
init_action            CS: default_init
delete_action          CS: default_delete
display_action         CS: default_display
machine_dump_action    CS: default_machine_dump
human_dump_action      CS: default_human_dump
compare_dump_action    CS: default_compare_dump
class_interface_value  IS: -1
temporary_instances    IS: 0

```

(debugger)

4 instances of class Vendor were described.

describe_one

Description: Calls the describe function with one object of a given class

Syntax: void describe_one (char *class_name, int rps_verbose)

Transaction:

```
action<transaction> transaction_describe_one (input: string class_name = "",
input: int verbose = 1)
```

Debugger Example:

```
(debugger) call describe_one("Build_Option", 1)
```

```
(Build_Option 179 Computer_Routing)
```

name	IS: Computer_Routing
editor_class_name	CS: Spl_Class_Form
class_string_id	CS: build_option
init_action	CS: default_init
delete_action	CS: delete_build_option
display_action	CS: display_build_option
machine_dump_action	CS: default_machine_dump
human_dump_action	CS: default_human_dump
compare_dump_action	CS: default_compare_dump
part	IS: (DKU_Part 176 Computer)
purchase_option_action	CS: purchase_option_false
build_option_action	CS: build_option_true
explode_action	CS: build_explode
cost	IS: 0.0000000000000000
cumm_cost	IS: 0.0000000000000000
backflush_step	IS: 0
alternate_routings	IS: ...null_list
task_classes	IS: [ATO_Shipment]
class_containers	IS: [(Class_Container 180 Anonymous)]
bom_entries	IS: [(Bom_Entry 181 Anonymous)]
supply_entries	IS: ...null_list
bor_entries	IS: ...null_list
orderings	IS: ...null_list

```

class_interface_value      IS: -1

temporary_instances        IS: 0

(debugger)

```

describe_by_name

Description: Looks up an object by name and calls the describe function

Syntax: void describe_by_name (char *object_name, int rps_verbose)

Transaction:

```

action<transaction> transaction_describe_by_name (input: string name = "",
input: int verbose = 1)

```

Debugger Example:

```

(debugger) call describe_by_name("Computer", 1)

```

```

(DKU_Part 176 Computer)

```

```

name                      IS: Computer

editor_class_name         CS: DKU_Part_Editor

class_string_id           CS: part

init_action               CS: default_init

delete_action             CS: delete_part

display_action            CS: display_part

machine_dump_action       CS: default_machine_dump

human_dump_action         CS: part_dump

compare_dump_action       CS: default_compare_dump

resource_constraints       IS: ...null_list

resource_supplies         IS: ...null_list

initial_amount            IS: 0.0000000000000000

relevant_status           IS: (Relevant_Status 13
_Relevant_Status_RELEVANT_REPAIR)

initial_history           IS: ...PRKMETH not implemented

resource_history          IS: ...PRKMETH not implemented

```



```

resource_batch_action      CS: resource_batch_false
duration_action            CS: default_resource_duration
quantity_action            CS: default_resource_quantity
consumable_action          CS: consumable_true
production                 IS: ...null_list
sales_order_line_action    CS: sales_order_line_false
planning_period_action     CS: planning_period_false
part_action                CS: part_true

routing_parent_container_action CS: routing_parent_container_false
description                IS: The Customers SPARCstation
routing_options            IS: [(Build_Option 179 Computer_Routing)]
phantom_part_action        CS: phantom_part_false
dku_part_action            CS: dku_part_true
planning_part_action       CS: planning_part_false
inventory_part_action      CS: inventory_part_false
configurable_part_action   CS: configurable_part_from_slot
buyer_code                 IS: default string
configurable_flag          IS: 1
planned_orders             IS: ...null_list
planning_container         IS: (Planning_Container 257 Anonymous)
all_planning_parents       IS: ...null_list
to_planning_parents        IS: ...null_list
consume_forecast_action    CS: part_consume_forecast
class_interface_value      IS: 2
temporary_instances        IS: 0

(debugger)

```

describe_by_uid

Description: Looks up an object by uid and calls the describe function

Syntax: void describe_by_uid (int uid, int rps_verbose)

Transaction:

```
action<transaction> transaction_describe_by_uid (input: int uid = -1, input: int
verbose = 1)
```

Debugger Example:

```
(debugger) call describe_by_uid(257, 1)
```

```
(Planning_Container 257 Anonymous)
```

```
editor_class_name          CS: Spl_Class_Form
class_string_id            CS: none
init_action                CS: default_init
delete_action              CS: delete_planning_container
display_action             CS: default_display
machine_dump_action        CS: default_machine_dump
human_dump_action          CS: default_human_dump
compare_dump_action        CS: default_compare_dump
part                       IS: (DKU_Part 176 Computer)
planning_periods           IS: [(Planning_Period 258 Anonymous)
                                (Planning_Period 259 Anonymous)]
order_to_shipment_map      IS: [1.0000000000000000]
class_interface_value      IS: -1
temporary_instances        IS: 0
```

```
(debugger) call describe_by_uid(257, 0)
```

```
(Planning_Container 257 Anonymous)
```

```
editor_class_name          CS: Spl_Class_Form
class_string_id            CS: none
init_action                CS: default_init
delete_action              CS: delete_planning_container
display_action             CS: default_display
```

```

machine_dump_action      CS: default_machine_dump
human_dump_action        CS: default_human_dump
compare_dump_action      CS: default_compare_dump
part                     IS: (DKU_Part 176 Computer)
planning_periods         IS: ...verbose needed to print this value
order_to_shipment_map    IS: ...verbose needed to print this value
class_interface_value    IS: -1
temporary_instances      IS: 0
(debugger)

```

how_many

Description: Counts the number of instances and classes of a PepperCode class

Syntax: void how_many (char *class_name)

Transaction:

```

action<transaction> transaction_how_many (input: string class_name = "", input:
int verbose = 0)

```

Debugger Example:

```

(debugger) call how_many("Vendor")

There are 4 instances of class Vendor.

There are 1 classes of class Vendor. (Vendor included)

(debugger)

```

list_objects

Description: Displays the class, uid, and name for each instance of a PepperCode class

Syntax: void list_objects (char *class_name)

Transaction:

```

action<transaction> transaction_list_objects (input: string class_name = "")

```

Debugger Example:

```

(debugger) call list_objects("Vendor")

4 instances of class Vendor...

```

```
(Vendor 144 Memory_Is_Us)

(Vendor 145 ACME_Chassis_Company)

(Vendor 146 Joes_Monitors)

(Vendor 147 We_Sell_Everything)

(debugger)
```

display_rhistory

Description: Displays a resource availability history

Syntax: void display_rhistory (char *resource_name, int verbose)

Transaction:

```
action<transaction> transaction_display_rhistory (input: string resource = "",
input: int verbose = 1)
```

display_rinitial_history

Description: Displays a resource initial history

Syntax: void display_rinitial_history (char *resource_name, int verbose)

Transaction:

```
action<transaction> transaction_display_rinitial_history (input: string resource
= "", input: int verbose = 1)
```

display_ahistory

Description: Displays an attribute history

Syntax: void display_ahistory (char *attribute_name, int verbose)

Transaction:

```
action<transaction> transaction_display_ahistory (input: string attribute = "",
input: int verbose = 1)
```

display_chistory

Description: Displays a calendar history

Syntax: void display_chistory (char *calendar_name)

Transaction:

```
action<transaction> transaction_display_chistory (input: string calendar = "")
```

Other Debugging Functions

`display_violated_constraints`

Input: string class_name

`display_resource_constraints`

Input: string resource_name

`transaction_display_resource_supplies`

Input: string resource_name

`transaction_printf`

Input: string pstring

`transaction_printf_with_current_time`

Input: string pstring

`transaction_set_intersect_debug_level`

Input: int debug_level = 0

Using Debugging Actions

This section describes PepperCode actions that are used for debugging Planning products implemented in PepperCode.

PepperCode debugging actions are needed for a variety of reasons. First, it is useful to view the values of PepperCode instance slots without using a debugger: a debugger may take several minutes to load, while a debugging action may take a few seconds or less to run. Second, a debugger may not be available. This occurs frequently at a customer unit, sometimes called “site”. Third, debugging information may not exist in the current environment. This occurs when .o files were not compiled with the -g option, as is the case with our internal and customer releases. Fourth, when information that is necessary for debugging must be calculated. This is where debugging actions are very useful because they can be written to display values of specific data structures, where the values already exist or are calculated in the debugging action.

The actions described in this document are for debugging only!!! Don’t use these actions in the PepperCode application or in any customization. These actions don’t always conform to current coding standards, so don’t use them as coding examples.

Understanding Key Terms

- **Describe:** A PepperCode utility that displays the value of every slot of a PepperCode instance.

- **History:** A C++ data structure that maintains values through time. The history is the backing data structure behind every part and equipment histogram. For items (or parts) and resources (or equipment), a history represents availability in float quantities.
- **Side Effect:** A C++ object that enforces the effect of a PepperCode instance—like a resource constraint or resource supply—on a History. The side effect automatically “fires” when a relevant value on the PepperCode instance is changed. For example, changing the quantity of a resource constraint will cause the side effect for that resource constraint to “fire”, thereby changing the associated resource history, which could be for an resource or item.
- **UID:** The unique identifier of a PepperCode instance. The uid of a PepperCode instance is the value of the “uid” slot.

Setting The Debugging Message Level

The following is some general PepperCode debugging advice.



For more information and specific debugging advice, see [Deciding Which Debugging Action To Use](#).

To understand what the system is doing, set the message level to 25. When the message level is 25, the system displays "what" it is doing and why any failures occur. These messages are particularly useful when the system isn't behaving as expected.

To see what actions are being called, set the message level to 50. At message level 50, the system displays the name of the action that is currently executing.



WARNING: The system runs significantly slower when message level 50 is used.

Running The Debugging Actions

The debugging actions described in this document are not connected to the GUI. They must be run from a command file or from the action interpreter.



For more information, see [Understanding Debug Command Files](#) or run the action interpreter.

Understanding The Debugging Action Categories

There are several different categories of PepperCode debugging actions. Each of these categories is described below along with the debugging actions that fit the category. Use these categories to find the debugging action that you need. A short description of each debugging action and its input parameters is provided in a later section of this document.

Displaying PepperCode Instance Information

The following actions display information about PepperCode instances. Most of these actions display the values of the slots of instances.

- transaction_describe_all
- transaction_describe_one
- transaction_describe_by_name
- transaction_describe_by_uid
- transaction_how_many
- transaction_list_objects

Displaying History Information

The following actions display information about C++ history objects. These actions are very useful for debugging the values of resource (equipment) and item (part) histograms, as well as debugging side effects.

- transaction_display_rhistory
- transaction_display_rinitial_history
- transaction_display_ahistory
- transaction_display_chistory

Displaying Task Reschedule Information

Task rescheduling involves the following kinds of operations:

- User reschedules from a task form or gantt chart
- Reschedules performed by the optimizer
- Compress / Expand from by either the user or optimizer
- Build Query reschedules
- Generate Initial Schedule reschedules

There is currently only one debugging action that is used to display task reschedule information, other than setting the message level. It is called `transaction_set_intersect_debug_level`. This action is very useful when debugging the various kinds of rescheduling listed above. For example, the debugging information produced by this action can tell you which resource (part or equipment resource) causes a task compress to fail.

Debugging Side Effects

The following actions are useful for debugging side effects. These actions allow the side effects of resource constraints and resource supplies to be asserted and retracted manually. It is **extremely** important that you **NOT** use this code as an example for performing side effect processing of any kind. These actions should be used for debugging only.

- `retract_resource_constraint`
- `assert_resource_constraint`
- `retract_resource_supply`
- `assert_resource_supply`
- `retract_task_side_effects`
- `assert_task_side_effects`
- `retract_resource_side_effects`
- `assert_resource_side_effects`

Displaying Time Period Information

The following actions display information about starting and ending time periods. They are currently used only for debugging C++ date/time functions for finding the start and end of a time period (day, week, month).

- `transaction_start_of_day`
- `transaction_end_of_day`
- `transaction_start_of_week`
- `transaction_end_of_week`
- `transaction_start_of_month`
- `transaction_end_of_month`

Miscellaneous Debugging Actions

The following debugging actions don't fit into any existing category.

- transaction_printf
- transaction_printf_with_current_time
- display_violated_constraints
- display_resource_constraints
- display_resource_supplies
- repair_me
- object_is_alive

Deciding Which Debugging Action To Use

This section lists some common debugging situations and which debugging action(s) can be used to debug the problem.

Deciding Which Debugging Action You Want To Use

Debugging Situation	Debugging Action To Use
Need to see the slot values of a PepperCode instance.	transaction_describe_by_name transaction_describe_by_uid
Need to see how many instances of a PepperCode class currently exist.	transaction_how_many
Need to see a one-line description of every instance of a PepperCode class.	transaction_list_objects
Need to see the availability of a part or equipment resource.	transaction_display_rhistory
The values on a part or equipment histogram appear to be incorrect.	transaction_display_rhistory
All equipment constraints are violated and the optimizer cannot repair them.	transaction_display_rhistory
Need to see if a PepperCode instance has been deleted.	object_is_alive
Some PepperCode instances which could have been deleted are displayed in the GUI.	object_is_alive
The equipment histogram appears to have been initialized incorrectly.	transaction_display_rhistory transaction_display_rinitial_history
A calendar may have been initialized incorrectly.	transaction_display_chistory

Testing a resource constraint side effect.	retract_resource_constraint assert_resource_constraint
Need to see how long it takes to load a command file.	transaction_printf_with_current_time
Need to see a listing of the violated constraints without using the scorecard.	display_violated_constraints
Need to see the resource constraints on a resource.	display_resource_constraints
Need to see the resource supplies on a resource.	display_resource_supplies
Need to test a constraint repair without running the optimizer.	
A reschedule operation is failing (such as a user reschedule, compress, expand, or build query).	transaction_set_intersect_debug_level

Understanding Debugging Action Descriptions

The following is a brief description of each PepperCode debugging action, along with its input and output parameters.



Note: Site is another name for unit; this name occurs in several of the following descriptions.

transaction_describe_all

Calls the DESCRIBE function on every instance of a PepperCode class.

```
action<transaction> transaction_describe_all

(input: string class_name = "",
input: int verbose = 1)
```

transaction_describe_one

Calls the DESCRIBE function on one randomly selected instance of a PepperCode class.

```
action<transaction> transaction_describe_one

(input: string class_name = "",
input: int verbose = 1)
```

transaction_describe_by_name

Calls the DESCRIBE function on a named PepperCode instance.

Accepts a site name as input for all objects that have a site. If no “site_name” is entered, only the “name” parameter is used to look up the PepperCode instance for describe.

```
action<transaction> transaction_describe_by_name

(input: string name = "",

input: string site_name = Does_Not_Apply",

input: int verbose = 1)
```

transaction_describe_by_uid

Calls the DESCRIBE function on a PepperCode instance that is referenced by UID.

```
action<transaction> transaction_describe_by_uid

(input: int uid = -1,

input: int verbose = 1)
```

transaction_how_many

Displays how many objects of a PepperCode class exist in memory. This action doesn't count objects that have been deleted in context. When the verbose flag is 1, the number of PepperCode instances for every subclass of the input class is displayed.

```
action<transaction> transaction_how_many

(input: string class_name = "",

input: int verbose = 0)
```

transaction_list_objects

Displays a one-line summary for every instance of a PepperCode class.

```
action<transaction> transaction_list_objects

(input: string class_name = "")
```

transaction_display_rhistory

Displays the values of every history element for a given resource history. The resource history can be a part history, equipment resource history, or standard resource history. When the verbose flag is 1, the resource constraints of each history element are also displayed.

Accepts a site name. If no “site_name” is entered, on the “name” parameter is used to look up the resource. If no resource is found, the default site is used to look up the resource.

```
action<transaction> transaction_display_rhistory

(input: string resource = "",

input: string site_name = "",

input: int verbose = 1)
```

transaction_display_rinitial_history

Displays the values of every history element for a given initial resource history. The resource history can be an equipment resource history or standard resource history. This transaction doesn't really apply to part histories.

Accepts a site name. If no “site_name” is entered, on the “name” parameter is used to look up the resource. If no resource is found, the default site is used to look up the resource.

```
action<transaction> transaction_display_rinitial_history

(input: string resource = "",

input: string site_name = "",

input: int verbose = 1)
```

transaction_display_ahistory

Displays the values of every history element for a given attribute history. The values displayed are state information. When the verbose flag is 1, the changers and dependents of each history element are also displayed.

```
action<transaction> transaction_display_ahistory

(input: string attribute = "",

input: int verbose = 1)
```

transaction_display_chistory

Displays the values of every history element for a given calendar history. The values represent the LEGAL and ILLEGAL periods of the calendar.

```
action<transaction> transaction_display_chistory

(input: string calendar = "")
```

transaction_set_intersect_debug_level

Displays the internal values of the intersector during a task reschedule, or during an optimize "next time to try". The displayed intersector values often show why a task cannot be compressed to a specific point in time, or why a task reschedule is failing. The debug level values are as follows:

0 = debugging off

1 = show output only

2 = please show me more than just output

3 = let me have it, show me everything

```
action<transaction> transaction_set_intersect_debug_level
    (input: int debug_level = 0)
```

transaction_printf

Displays a given string to the server shell window. This transaction is useful for placing messages inside of a command file.

```
action<transaction> transaction_printf
    (input: string pstring,
```

transaction_printf_with_current_time

Displays a given string to the server shell window, along with the current system clock time. This transaction is useful for placing messages inside of a command file. Specifically, this transaction is great for showing how long command files take to load.

```
action<transaction> transaction_printf_with_current_time
    (input: string pstring,
```

transaction_start_of_day

Displays the result of calling the PepperCode function START_OF_DAY on a given date.

```
action<transaction> transaction_start_of_day
    (input: string date_string,
     output: date start_of_day)
```

transaction_end_of_day

Displays the result of calling the PepperCode function END_OF_DAY on a given date.

```

action<transaction> transaction_end_of_day

(input: string date_string,

output: date end_of_day)

```

transaction_start_of_week

Displays the result of calling the PepperCode function START_OF_WEEK on a given date.

```

action<transaction> transaction_start_of_week

(input: string date_string,

output: date start_of_week)

```

transaction_end_of_week

Displays the result of calling the PepperCode function END_OF_WEEK on a given date.

```

action<transaction> transaction_end_of_week

(input: string date_string,

output: date end_of_week)

```

transaction_start_of_month

Displays the result of calling the PepperCode function START_OF_MONTH on a given date.

```

action<transaction> transaction_start_of_month

(input: string date_string,

output: date start_of_month)

```

transaction_end_of_month

Displays the result of calling the PepperCode function END_OF_MONTH on a given date.

```

action<transaction> transaction_end_of_month

(input: string date_string,

output: date end_of_month)

```

display_violated_constraints

Displays a short description of all violated constraints of a given PepperCode class. It is assumed that the PepperCode class is a descendant (or equal to) the class Repairable_Constraint.

Now takes a score card as input instead of a constraint class name.

```
action<transaction> display_violated_constraints

(input: string score_card_name = "mfg_score_card",
```

display_resource_constraints

Displays the resource constraints of a resource.

Has a parameter for turning on and off the displaying of constraints. This parameter is for internal use only.

```
action<transaction> display_resource_constraints

(input: string resource_name,

input: int display_constraints = 1,
```

display_resource_supplies

Displays the resource supplies of a resource.

Has a parameter for turning on and off the displaying of supplies. This parameter is for internal use only.

```
action<transaction> display_resource_supplies

(input: string resource_name,

input: int display_supplies = 1,
```

retract_resource_constraint

Retracts the side effect of a resource constraint by calling the PepperCode function RETRACT.

```
action retract_resource_constraint

(input: instance<Resource_Constraint> resource_constraint,
```

assert_resource_constraint

Asserts the side effect of a resource constraint by calling the PepperCode function ASSERT.

```
action assert_resource_constraint

(input: instance<Resource_Constraint> resource_constraint,
```

retract_resource_supply

Retracts the side effect of a resource supply by calling the PepperCode function RETRACT.

```
action retract_resource_supply

(input: instance<Resource_Supply> resource_supply,
```

assert_resource_supply

Asserts the side effect of a resource supply by calling the PepperCode function ASSERT.

```
action assert_resource_supply

(input: instance<Resource_Supply> resource_supply,
```

retract_task_side_effects

Retracts the side effects of the resource constraints and resource supplies on a task. The uid or name of the task can be used to identify the task. If the uid is passed as the value of parameter “duration_task”, then the “task_name” parameter isn’t used to look up the task. Non-zero values for the parameters “constraints” and “supplies” specify if constraints and supplies will be retracted.

```
action retract_task_side_effects

(input: instance<Duration_Task> duration_task = 0,

input: string task_name = "",

input: int constraints = 1,

input: int supplies = 1,
```

assert_task_side_effects

Asserts the side effects of the resource constraints and resource supplies on a task. The uid or name of the task can be used to identify the task. If the uid is passed as the value of parameter “duration_task”, then the “task_name” parameter isn’t used to look up the task. Non-zero values for the parameters “constraints” and “supplies” specify if constraints and supplies will be asserted. This action assumes that retract_task_side_effects has already been called on duration_task—the side effects are already retracted.

```
action assert_task_side_effects

(input: instance<Duration_Task> duration_task = 0,

input: string task_name = "",

input: int constraints = 1,

input: int supplies = 1,
```


retract_resource_side_effects

Retracts the side effects of the resource constraints and resource supplies on a resource. The uid or name of the resource can be used to identify the resource. If the uid is passed as the value of parameter “resource”, then the “resource_name” parameter isn’t used to look up the resource. Non-zero values for the parameters “constraints” and “supplies” specify if constraints and supplies will be retracted.

```
action retract_resource_side_effects

(input: instance<Resource> resource = 0,

input: string resource_name = "",

input: int constraints = 1,

input: int supplies = 1,
```

assert_resource_side_effects

Asserts the side effects of the resource constraints and resource supplies on a resource. The uid or name of the resource can be used to identify the resource. If the uid is passed as the value of parameter “resource”, then the “resource_name” parameter isn’t used to look up the resource. Non-zero values for the parameters “constraints” and “supplies” specify if constraints and supplies will be asserted. This action assumes that retract_resource_side_effects has already been called on resource—the side effects are already retracted.

```
action assert_resource_side_effects

(input: instance<Resource> resource = 0,

input: string resource_name = "",

input: int constraints = 1,

input: int supplies = 1,
```

repair_me

Calls the repair method of a repairable constraint. This action is useful for testing constraint repairs without using optimize.

```
action repair_me

(input: instance<Repairable_Constraint> repairable_constraint,

input: string deconflict_env_name = "deconflict_env",
```

object_is_alive

Calls the PepperCode function OBJECT_IS_ALIVE on a given PepperCode object. The OBJECT_IS_ALIVE function makes sure that an object has not been deleted in context. This

action is useful if you think a specific PepperCode object has been deleted, but yet continues to be used in the system.

```
action object_is_alive

(input: instance<Spl_Class> object,
```

resource_info

Displays all possible values about a resource that most programmers would ever want to know. If you plan to use this action, the input parameters should be self-explanatory. Otherwise, you probably should not use this action.

```
action<transaction> resource_info

(input: string resource_name = "",
input: int describe = 0,
input: int display_history = 0,
input: int verbose = 0,
input: int retract_constraints_before_displaying_history = 0,
input: int retract_supplies_before_displaying_history = 0,
input: int display_constraints = 0,
input: int display_supplies = 0,
```

create_some_objects

Creates a given number of PepperCode objects of a given PepperCode class. This action is useful for testing any situation where PepperCode objects must be created and then deleted—context problems, command file loads, and efficiency questions. This action can be used with action delete_some_objects.

```
action create_some_objects

(input: int number_of_objects = 100,
input: string class_name = "Named_Object",
input: string object_name = "bogus_object",
```

delete_some_objects

Deletes a given number of PepperCode objects of a given PepperCode class. This action is useful for testing any situation where PepperCode objects must be created and then deleted—context problems, command file loads, and efficiency questions. This action can be used with action create_some_objects.

```

action delete_some_objects

(input: int number_of_objects = 100,

input: string class_name = "Named_Object",

```

Understanding Debug Command Files

The following command file was run on the standard bike dataset after optimize had reduced all constraint violations. The output section follows the command file section.

```

transaction_printf_with_current_time(:pstring "Starting Debugging Actions
Example...")

transaction_list_objects(:class_name "Score_Card_Element")

transaction_describe_one(:class_name "Score_Card_Element")

transaction_describe_by_name(:name "main_environment")

transaction_describe_by_uid(:uid 1000)

transaction_how_many(:class_name "Base_Task" :verbose 0)

transaction_how_many(:class_name "Base_Task" :verbose 1)

object_is_alive(:object 1000)

transaction_display_rhistory(:resource "SM_Aluminum" :verbose 0)

transaction_display_chistory(:calendar "calendar_all_time")

transaction_printf_with_current_time(:pstring "...Finished Debugging Actions
Example")

```

Following is the output of command file load.

```

Loading file /home/daun/data/bike/debugging-example.command

(09/11/96 10:24:17)-> Starting Debugging Actions Example...

12  instances of class  Score_Card_Element...

```

```

(Score_Card_Element 1103 request_milestone_constraint_element)
(Score_Card_Element 1104 promise_milestone_constraint_element)
(Score_Card_Element 1105 fg_constraint_element)
(Score_Card_Element 1106 standard_rm_constraint_element)
(Score_Card_Element 1107 detailed_equipment_constraint_element)
(Score_Card_Element 1108 aggregate_equipment_constraint_element)
(Score_Card_Element 1109 inventory_target_constraint_element)
(Score_Card_Element 1110 capacity_by_period_element)
(Score_Card_Element 1111 change_over_constraint_element)
(Score_Card_Element 1112 safety_stock_constraint_element)
(Score_Card_Element 1113 excess_stock_constraint_element)
(Score_Card_Element 1114 reduce_routing_wip_element)

```

```

(Score_Card_Element 1103 request_milestone_constraint_element)

score_card                                IS: (Score_Card 1102 mfg_score_card)

constraint_class                          IS: Request_Milestone_Constraint

description                              IS: Requested Deliveries

dump_header                               IS:   Request Date      Customer
Shipment Start      Shipment End      Shipment              Site      Sales
Order

violated_constraints                      IS: ...null_list

number_of_violations                     IS: 0

penalty                                  IS: 0.0000000000000000

display_name                             IS: request_milestone_constraint_element

human_dump_action                         CS: score_card_element_dump

set_display_name_action CS: default_set_display_name

editor_class_name                        CS: Spl_Class_Form

init_action                             CS: default_init

```

```

delete_action          CS: default_delete
display_action         CS: default_display
machine_dump_action    CS: default_machine_dump
compare_dump_action    CS: default_compare_dump
class_interface_value  -1
temporary_instances    0

```

```
(Environment 725 main_environment)
```

```

min_system_time        IS: 01/01/95 00:00:00
max_system_time        IS: 01/01/96 00:00:00
start_of_time          IS: 01/01/95 00:00:00
end_of_time            IS: 01/01/96 00:00:00
early_fence            IS: 03/01/95 00:00:00
late_fence             IS: 06/01/95 00:00:00
current_time           IS: 03/20/95 00:00:00
leveling_fence         IS: 01/01/95 00:00:00
old_early_fence        IS: 03/01/95 00:00:00
old_late_fence         IS: 06/01/95 00:00:00
default_site           IS: (Plant_Site 723 SM)
task_queue             IS: ...null_list
msg_level              IS: 1
debug_slot             IS: 0
reschedule_env IS: (Reschedule_Env 726 reschedule_environment)
mfg_env               IS: (Mfg_Env 729 main_mfg_environment)
global_constraints     IS: ...null_list
resource_gantt_st      IS: 03/01/95 00:00:00
resource_gantt_et      IS: 06/01/95 00:00:00
max_tasks_per_resource_gantt IS: 500

```

```

display_name          IS: main_environment
set_display_name_action CS: default_set_display_name
editor_class_name     CS: Spl_Class_Form
init_action           CS: default_init
delete_action         CS: default_delete
display_action        CS: default_display
machine_dump_action   CS: default_machine_dump
human_dump_action     CS: default_human_dump
compare_dump_action   CS: default_compare_dump
class_interface_value -1
temporary_instances    0

```

```
(Generate_Method 1000 __Anonym__1000)
```

```

generate_row_action    IS:
generate_site_aggregate_resource_required_units
editor_class_name     CS: Spl_Class_Form
init_action           CS: default_init
delete_action         CS: default_delete
display_action        CS: default_display
machine_dump_action   CS: default_machine_dump
human_dump_action     CS: default_human_dump
compare_dump_action   CS: default_compare_dump
class_interface_value -1
temporary_instances    0

```

There are 391 instances of class Base_Task.

There are 40 classes of class Base_Task. (Base_Task included)

Base_Task	
Base_Task	391
Duration_Task	391
Non_Split_Child_Task	391
Routine_Task	391
Routine_Task	391
Unsplittable_Task	391
Unsplittable_Parent_Task	98
Routing_Parent	87
Unsplittable_Leaf_Task	293
Routing_Task	137
Transfer_Task	0
Forecast_Task	0
Transport_Task	11
Shipment_Task	11
Routing_Parent_Container	11
Production_Task	126
Shipment_Parent	11
Shipment_Line	11
Shipment_Transport	0
Forecast_Shipment_Parent	0
Negative_Supply_Task	0
Other_Independent_Demand_Task	0
Transfer_Parent	0
Production_Parent	76
Splittable_Task	0

Downtime_Task	0
Scheduled_Downtime_Task	0
Co_Downtime_Task	0
Purchase_Order_Line_Delivery	5
Planned_Order	140
Split_Child_Task	0
Milestone_Task	0
Start_Milestone	0
End_Milestone	0
Achiever_Task	0
Assemble_Bicycle	26
Paint_Frame	46
Paint_Tandem_Bike_Frame	4
Assemble_Tandem_Bike	4
Weld_Frame	46

Object 1000 is ALIVE.

SM_Aluminum History:

(01/01/95 00:00:00 . 03/23/95 00:00:00)	0.0000000000000000
(03/23/95 00:00:00 . 03/24/95 20:59:58)	14.0000000000000000
(03/24/95 20:59:58 . 03/25/95 00:59:58)	10.0000000000000000
(03/25/95 00:59:58 . 03/25/95 04:59:58)	6.0000000000000000
(03/25/95 04:59:58 . 04/06/95 10:59:58)	2.0000000000000000
(04/06/95 10:59:58 . 01/01/96 00:00:00)	0.0000000000000000

calendar_all_time History:

(01/01/95 00:00:00 . 01/01/96 00:00:00)	LEGAL
---	-------

(09/11/96 10:24:18)→ ...Finished Debugging Actions Example

Using Sanity Checks

This section describes the software that verifies the consistency of Planning data models. This software is often referred to as “sanity checks”.

While the existing Planning transactions perform error checking on input data, they cannot check the consistency of the data model. For example, the transaction for creating an inventory item (or part) can verify if the data needed to create a part is correct. However, this same transaction cannot perform error checking on whether a build option exists for a buildable inventory item. In this case, the Planning system assumes that a build option will be created later with a different transaction. If the build option isn’t created, then the buildable inventory item might not be replenished by the optimizer: for example, there might be unexpected results because of an inconsistent data model).

It is because of these types of cases that separate error checks are needed after the data model is loaded into memory.

Understanding What Sanity Checks Do and Don’t Do

The sanity checks perform error checking on specific in-memory data model relationships. The output of sanity checks is a listing of error conditions. This output can be directed to a file or to the server shell window (the default).

The sanity checks are NOT used to perform error checking that should occur in transactions. If a transaction can check for a specific error condition, then that error condition should not be checked by the sanity code. The one exception to this rule is error checking that is very expensive to perform (because it is run once for every transaction called).

Using Sanity Checks

The sanity checks are performed by calling one of the following transactions.

- **transaction_mfg_sanity_check** : Check manufacturing data consistency as well as project-management (scheduler) data consistency.
- **transaction_pm_sanity_check** : Check only project-management (scheduler) data consistency

Both of these transactions have the same two input parameters, which are described here:

- **verbose** : The legal values for this parameter are 0 and 1. When 0, only the number of problems will be displayed. When 1, the objects that are involved in the problems found will be displayed.

- **filename** : This is the path to an output file. If the value of this parameter is "", then the output of the sanity check will be displayed in your server console window. If the value is a legal filename, then the output of the sanity check will be written to that file.

Currently, there is no GUI for running either of the sanity transactions.



For more information including examples, refer to Understanding Sanity Check Output.

Understanding Each Sanity Check

The following is a description of the error checking performed by each sanity check. The first 5 sanity checks are implemented in the scheduler module by **transaction_pm_sanity_check**. The remaining sanity checks are implemented in the manufacturing module by **transaction_mfg_sanity_check**.



transaction_mfg_sanity_check calls **transaction_pm_sanity_check**. Client Services project managers can use this calling structure as an example for adding customer specific sanity checks to a customer module.

A Parent Task Must Have Subtasks

This check ensures that every parent task has at least one subtask, or child task. This is a requirement of the base scheduling system because the start and end times of a parent task are derived from its subtasks, or children.

The following is the output of this check.

```
There are 0 parent tasks without subtasks.
```

Work Duration Check For Unsplittable Leaf Tasks

This check ensures that every unsplittable leaf task has a work duration that is greater than 0 and not greater than the "longest" legal calendar interval (of its calendar). This is a requirement of the base scheduling system.



This check will be affected by the 2.5 feature "Task Wrapping Around Calendars".

The following is the output of this check:

```
There are 0 zero duration tasks (of class Unsplittable_Leaf_Task).
```

```
There are 0 tasks (of class Unsplittable_Leaf_Task) that are too long for
their calendars.
```

A Calendar Must Have Computed Legal Time

This check ensures that every calendar has at least one “legal” time interval. This also includes making sure that the calendar has been computed by the calendar mechanism. This is a requirement of the base scheduling system because without at least one legal calendar interval a calendar cannot be used to reschedule tasks or initialize reusable resources.

The following is the output of this check.

```
There are 0 calendars that have no LEGAL_TIME.

There are 0 calendars that are not COMPUTED.
```

A Resource Constraint Must Have Quantity ≥ 0.0

This check ensures that every resource constraint has a quantity that is greater than or equal to 0.0. This is a requirement of the base scheduling system.

The following is the output of this check.

```
There are 0 resource constraints whose quantity is less than or equal to 0.0
```

A Resource Supply Must Have Quantity ≥ 0.0

This check ensures that every resource supply has a quantity that is greater than or equal to 0.0. This is a requirement of the base scheduling system.



Note: Negative supply tasks will be found in this check. Customers using negative supply tasks (which are non-standard and not generally supported in the product) can ignore this sanity check.

The following is the output of this check.

```
There are 0 resource supplies whose quantity is less than or equal to 0.0
```

Start And End Time Checks Of Effective Entries

This check ensures that the following will be true about every effective entry (bom entry, bor entry, supply entry, planning bom entry, and transfer option):

- The start time of the effective entry will be less than the end time of the effective entry. This is important because the start and end times of effective entries specify an interval of time that is used by the scheduling system (which assumes that it is using legal intervals).
- The start time of the effective entry will be greater than or equal to start of time. This check is

used mainly to find dates that are “out of bounds” and can be ignored. Nothing in the system will break because of these “out of bounds” cases.

- The end time of the effective entry will be less than or equal to end of time. The same rule applies here as well concerning “out of bounds” cases.

The following is the output of this check.

```
There are 0 effective entries whose start times are not less than their end times.
```

```
There are 0 effective entries whose start times are earlier than Start Of Time.
```

```
There are 0 effective entries whose end times are later than End Of Time.
```

A Routing Entry Must Have Quantity >= 0.0

This check ensures that every routing entry (bom entry, bor entry, and supply entry) has a quantity that is greater than or equal to 0.0. This will prevent constraints and supply objects from being created with a quantity of 0.0.



Note: This check should be eliminated as soon as error checking is added to the create and modify transactions for bor entries, bom entries, and supply entries.

The following is the output of this check.

```
There are 0 routing entries whose quantities are not greater than 0.0.
```

A Routing Entry Must Match A Routing Step

This check ensures that every routing entry (bom entry, bor entry, and supply entry) matches a step on the routing; the routing_step slot of the routing entry must match the index of some task entry that is stored on the build option.

The following is the output of this check.

```
There are 0 routing entries that do not match a routing step.
```

A Bor Entry Must Have A Valid Equipment Class

This check ensures that every equipment class on a bor entry will have at least one corresponding equipment resource. Without a corresponding equipment resource, the Bor_Entry is useless to the manufacturing system.

The following is the output of this check.

```
There are 0 bor entries whose equipment class has no equipment instances.
```

A Build Option Must Have At Least One Routing Step

This check ensures that every build option will have at least one routing step. The build option cannot be exploded without a routing step.

The following is the output of this check.

```
There are 0 build options that have no routing tasks.
```

A Build Option Must Supply An Item (Part) For All Time

This check ensures that the supply entries of a build option are effective such that the item of the build option can be supplied at any period of time between the start and end of time. This also means that at least one effective supply entry must exist on the build option.

The following is the output of this check.

```
There are 0 build options whose supply entries do not cover the entire  
schedule.
```

A Build Option Should Have Only One Primary Order Bor

This check ensures that every build option should have only one primary order bor.

The following is the output of this check.

```
There are 0 build options that have multiple Primary Order BORS.
```

A Build Option Should Have Only One Primary Operation Bor Per Routing Step

This check ensures that every build option should have only one primary operation bor per routing step.

The following is the output of this check.

```
There are 0 build options that have multiple Primary Operation BORS.
```

An Inventory Item Must Have A Way To Be Replenished

This check ensures that every inventory item (or inventory part) has at least one way to be replenished; the item has at least one purchase option, build option, or transfer option.

The following is the output of this check.

```
There are 0 inventory items that may not be replenishable (no Build Option,  
Purchase Option, or Transfer Option exists).
```

A Sales Order Must Have Sales Order Lines

This check ensures that every sales order has at least one sales order line.

The following is the output of this check.

```
There are 0 sales orders that do not have sales order lines.
```

A Purchase Order Must Have Purchase Order Lines

This check ensures that every purchase order has at least one purchase order line.

The following is the output of this check.

```
There are 0 purchase orders that do not have purchase order lines.
```

An Equipment Resource Must Have Enough Capacity To Repair Any One Of Its Equipment Constraints

This check ensures that every equipment resource has enough capacity to repair any individual equipment constraint that requests it; the maximum availability of the equipment isn't less than the maximum equipment constraint quantity. This check is very helpful when the capacity of an equipment resource did not get initialized properly.

The following is the output of this check.

```
There are 0 equipment resources without enough capacity to satisfy its constraints.
```

Understanding Potential Sanity Checks

The following are some ideas for sanity checks that have not been implemented.

Every Product Must Map To An Inventory Item

This check would make sure that every product (or DKU part) mapped to an inventory item (or inventory part).

Technical notes: (to help with future implementation).

- Every product must have an effective transfer option.
- Each of these transfer options must have a legitimate part mapping on the from-unit (from-site) of the transfer option, or an item with the same name exists at the from-unit.
- For the legitimate mapped item, there must be a transfer prep option.

Sourcing Logic Checks

This check would ensure the following:

- For each ratio source template, all descriptor ratios must sum to 1.0.

- For each descriptor on a template, there must be a routing option that matches (by attribute); the routing option cannot be a transfer prep. Also, the routing option must be legitimate for the part.

Understanding Sanity Check Output

The following are some examples of the output of **transaction_mfg_sanity_check**. The section headings represent the syntax of the transaction as it would appear in a command file.

transaction_mfg_sanity_check (:verbose 0 :filename "")

```

There are 0 parent tasks without subtasks.

There are 0 zero duration tasks (of class Unsplittable_Leaf_Task).

There are 0 tasks (of class Unsplittable_Leaf_Task) that are too long for
their calendars.

There are 0 calendars that have no LEGAL_TIME.

There are 0 calendars that are not COMPUTED.

There are 0 resource constraints whose quantity is less than or equal to 0.0

There are 0 resource supplies whose quantity is less than or equal to 0.0

There are 0 effective entries whose start times are not less than their end
times.

There are 0 effective entries whose start times are earlier than Start Of
Time.

There are 0 effective entries whose end times are later than End Of Time.

There are 0 routing entries whose quantities are not greater than 0.0.

There are 0 routing entries that do not match a routing step.

There are 0 bor entries whose equipment class has no equipment instances.

There are 0 build options that have no routing tasks.

There are 0 build options whose supply entries do not cover the entire
schedule.

There are 0 build options that have multiple Primary Order BORS.

There are 0 build options that have multiple Primary Operation BORS.

There are 0 inventory parts that may not be replenishable (no Build Option,
Purchase Option, or Transfer Option exists).

There are 0 sales orders that do not have sales order lines.
```

There are 0 purchase orders that do not have purchase order lines.

There are 0 equipment resources without enough capacity to satisfy its constraints.

transaction_mfg_sanity_check(:verbose 0 :filename "")

There are 1 parent tasks without subtasks.

There are 1 zero duration tasks (of class Unsplittable_Leaf_Task).

There are 1 tasks (of class Unsplittable_Leaf_Task) that are too long for their calendars.

There are 1 calendars that have no LEGAL_TIME.

There are 1 calendars that are not COMPUTED.

There are 1 resource constraints whose quantity is less than or equal to 0.0

There are 1 resource supplies whose quantity is less than or equal to 0.0

There are 2 effective entries whose start times are not less than their end times.

There are 2 effective entries whose start times are earlier than Start Of Time.

There are 1 effective entries whose end times are later than End Of Time.

There are 6 routing entries whose quantities are not greater than 0.0.

There are 9 routing entries that do not match a routing step.

There are 1 bor entries whose equipment class has no equipment instances.

There are 5 build options that have no routing tasks.

There are 6 build options whose supply entries do not cover the entire schedule.

There are 1 build options that have multiple Primary Order BORS.

There are 1 build options that have multiple Primary Operation BORS.

There are 1 inventory parts that may not be replenishable (no Build Option, Purchase Option, or Transfer Option exists).

There are 1 sales orders that do not have sales order lines.

There are 1 purchase orders that do not have purchase order lines.

There are 1 equipment resources without enough capacity to satisfy its constraints.

transaction_mfg_sanity_check(:verbose 1 :filename "")

Parent tasks without subtasks:

(1523 childless_parent)

Zero duration Unsplittable_Leaf_Tasks:

(1524 no_work_duration_task)

Unsplittable_Leaf_Tasks that are too long for their calendars:

(1525 too_long_task)

Calendars that have no LEGAL_TIME:

(1526 no_time_calendar)

Calendars that are not COMPUTED:

(1527 not_computed_calendar)

Resource constraints whose quantity is less than or equal to 0.0:

(1530 SM_test_resource 0.0000000000000000 test_task (03/01/95 00:00:00 .
03/02/95 00:00:00))

Resource supplies whose quantity is less than or equal to 0.0:

(1531 SM_test_resource 0.0000000000000000 test_task (03/01/95 00:00:00 .
03/02/95 00:00:00))

Effective entries whose start times are not less than their end times:

(1534 Bogus_Effective_Class (05/01/95 00:00:00 . 04/01/95 00:00:00)
(799311600 . 796723200))

(1535 Bogus_Effective_Class (05/01/90 00:00:00 . 12/31/69 16:00:00)
(641545200 . 0))

Effective entries whose start times are earlier than Start Of Time:

```
(1535 Bogus_Effective_Class (05/01/90 00:00:00 . 12/31/69 16:00:00)
(641545200 . 0))
```

```
(1536 Bogus_Effective_Class (12/31/69 16:00:00 . 04/01/99 00:00:00) (0 .
922953600))
```

Effective entries whose end times are later than End Of Time:

```
(1536 Bogus_Effective_Class (12/31/69 16:00:00 . 04/01/99 00:00:00) (0 .
922953600))
```

Routing entries whose quantities are not greater than 0.0:

```
(1539 Bom_Entry SM_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1542 Bor_Entry SM_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1559 Bor_Entry SM_primary_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1560 Bor_Entry SM_primary_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1561 Bor_Entry SM_primary_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1562 Bor_Entry SM_primary_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

Routing entries that do not match a routing step:

```
(1539 Bom_Entry SM_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1542 Bor_Entry SM_bogus_build_option 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

```
(1544 Supply_Entry SM_bogus_build_option_A 1.0000000000000000 (03/05/95
00:00:00 . 01/01/96 00:00:00) (794390400 . 820483200))
```

```
(1546 Supply_Entry SM_bogus_build_option_M 1.0000000000000000 (01/01/95
00:00:00 . 04/01/95 00:00:00) (788947200 . 796723200))
```

```
(1547 Supply_Entry SM_bogus_build_option_M 1.0000000000000000 (04/01/95
00:00:00 . 05/01/95 00:00:00) (796723200 . 799311600))
```

```
(1548 Supply_Entry SM_bogus_build_option_M 1.0000000000000000 (05/02/95
00:00:00 . 06/01/95 00:00:00) (799398000 . 801990000))
```

```
(1549 Supply_Entry SM_bogus_build_option_M 1.0000000000000000 (06/01/95
00:00:00 . 01/01/96 00:00:00) (801990000 . 820483200))
```

```
(1551 Supply_Entry SM_bogus_build_option_Z 1.0000000000000000 (01/01/95
00:00:00 . 06/05/95 00:00:00) (788947200 . 802335600))
```

```
(1554 Supply_Entry SM_bogus_build_option_S 1.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

Bor entries whose equipment class has no equipment instances:

```
(1542 Bor_Entry Bogus_Equipment_Class 0.0000000000000000 (01/01/95
00:00:00 . 01/01/96 00:00:00) (788947200 . 820483200))
```

Build options that have no routing tasks:

```
(1538 SM_bogus_build_option)
(1543 SM_bogus_build_option_A)
(1545 SM_bogus_build_option_M)
(1550 SM_bogus_build_option_Z)
(1553 SM_bogus_build_option_S)
```

Build options whose supply entries do not cover the entire schedule:

```
(1538 SM_bogus_build_option)
(1543 SM_bogus_build_option_A)
(1545 SM_bogus_build_option_M)
(1550 SM_bogus_build_option_Z)
(1553 SM_bogus_build_option_S)
(1555 SM_primary_bogus_build_option)
```

Build options that have multiple Primary Order BORS:

```
(1555 SM_primary_bogus_build_option)
```

Build options that have multiple Primary Operation BORS:

```
(1555 SM_primary_bogus_build_option)
```

Inventory parts that may not be replenishable (no Build Option, Purchase Option, or Transfer Option exists):

(1552 SM_part_not_for_build_option)

Sales orders that do not have sales order lines:

(1563 SM_sales_order_without_lines)

Purchase orders that do not have purchase order lines:

(1564 SM_purchase_order_without_lines)

Equipment resources without enough capacity to satisfy its constraints:

(1567 SM_resource_that_cannot_be_repaired)

Index

#

- #document 3-7
 - error messages 3-8
 - format 3-9
 - writing comments to a file 3-9
- #include
 - differences in PepperCode prior 8.0 3-5
 - PepperCode compared to C++ 3-2
 - rules for writing 3-2
 - use instead forward declarations 3-4
 - using 11-24
 - using two files that include each other 3-4
 - writing 3-1
- #notice 3-11
 - warning message 14-20

▪

- .spl for PepperCode files 3-1
- .splrc
 - modifying compiler behavior 12-7

<

- <directory> 12-13, 12-16

8

- 8.0 PepperCode
 - #include differences for prior 8.0 3-5

A

- action
 - declaration
 - forward 5-15
 - execution
 - automatic 5-42
 - parameters 5-7
 - spl_main 12-1
 - spl_main definition 5-42
- action debug tracing 14-23
- action interpreter 14-21
- Action Interpreter

- how it is used 5-41
- action parameters
 - see parameters 11-9
- Action_Status 5-40
- actions 5-1
 - Action_Status 5-40
 - actions used for debugging 14-41
 - adding to a class 11-1
 - checking outputs 11-8
 - context 5-33
 - customizing 11-26
 - declaration error 14-10
 - example 2-5, 5-1
 - examples 5-3
 - executing 5-18
 - executing within another action 5-33
 - how they are executed 5-41
 - incomplete and forward declarations 5-4
 - matching parameter lists 5-6
 - new rule for invoking 5-19
 - parameter default values 5-9
 - parameter list 5-2
 - parameters 5-7
 - parameters are no longer static 5-12
 - parent action passing to child 5-34
 - PepperCode and C/C++ comparison 1-4
 - replacing standard method actions 11-26
 - required input parameter 5-8
 - running debugging actions 14-42
 - schema
 - errors 14-9
 - schema declarations & definitions 5-15
 - schemas 5-13
 - syntax 5-2
 - transaction logs 5-42
 - using context and no_context 5-35
 - writing 11-6
 - writing to dispatch methods 11-18
- ADD_TO_HISTORY_VALUE 10-43
- ADD_TO_HISTORY_VALUE_ON_CALENDAR 10-42
- AREA_UNDER_CURVE 10-39
- arrays 8-1
 - accesses 8-8
 - arrays of arrays 8-5
 - associative 8-1
 - exists function 8-2
- bounds
 - associative 8-2

- nonassociative 8-4
- enlarging 8-4
- functions 8-5
 - use with arrays of arrays 8-7
- indexed by float 8-8
- mixing associative and nonassociative 8-6
- multidimensional 8-5
- nonassociative 8-3
 - exists function 8-4
- use with statements 8-8
- assignment statement 6-1
- osets 7-1

B

- Base_Class 4-1, 4-10
- BREAK 6-7
- break statement 6-6

C

- c 12-13
- C/C++
 - compared to PepperCode actions 1-4
 - compared to PepperCode classes 1-3
- C/C++ functions
 - accessing 10-8
 - declaring 10-9
 - passing arguments 10-9
 - using RPS_IMPORT 11-22
 - using typedefs with 10-10, 11-21
- C++ comparison 2-6
- C++ functions
 - checking for corresponding function 11-20
 - naming 11-21
 - placing C++ code 11-20
 - providing PepperCode types 11-21
 - writing 11-20
- C++ header files
 - not necessary to include warning 14-20
- casting 11-5
 - using with methods 11-19
- CD-ROM
 - ordering iii
- changes
 - class
 - Using Name of Class in Expression 4-11
- check_dump *See* Index of Compiler Options
- Child_Task
 - example 11-3
- class
 - declarations & definitions 4-6
 - parent and child relationships 14-15

- warning at compile time 14-15
- class slot 4-9
- class slots
 - see slots 11-2
- class_name
 - using instead of GET_NAME_OF_CLASS 10-22
- classes 4-1
 - adding a constraint class 11-32
 - adding and action to 11-1
 - adding default values 11-2
 - changing attribute of derived class 4-7
 - customizing and displaying names 11-25
 - declaring 4-6
 - default values 4-5
 - example 2-4
 - forward declarations 4-6
 - inheriting redefined values 4-3
 - instance names 4-12
 - instance of 4-3
 - multiple inheritance 4-5
 - naming 11-1
 - PepperCode and C/C++ comparison 1-3
 - predefined 4-10
 - writing 11-1
 - writing new definitions 4-3
 - writing parent and child 11-3
- client 12-16
- CLOSE_DUMP_FILE 10-44
- code reading errors 14-15
- command line
 - rules 12-11
 - syntax 12-10
- comments
 - C++ style 3-7
 - documentation 3-7
 - format for documentation comments 3-9
 - generating for an .spl file 3-11
 - notice 3-11
 - writing to a file 3-9
- compiler
 - fatal error 14-11
 - generic error 14-11
 - severe error 14-10
- compiler options
 - <directory> 12-13, 12-16
 - c 12-13
 - client 12-16
 - compiler maintenance 12-17
 - compiler or linker to run 12-12
 - compiling PepperCode 12-13
 - cpp_to_object 12-13
 - debug 12-13
 - default (no option switch) 12-11
 - define <macroname>=<value> 12-16
 - doc 12-15
 - for C++ source code 12-16
 - header_only 12-15

- include 12-16
- include <directory> 12-13
- installation 12-8
- lib_tag 12-14
- loud 12-14
- machine specific escape clause 12-17
- make_implib 12-13
- make_library 12-12
- make_program 12-11
- most used 12-11
- no_header 12-15
- no_main 12-17
- no_object 12-13
- no_rt 12-17
- no_warn 12-14
- optimize 12-13
- preprocessor 12-13
- purify 12-17
- quantity 12-17
- quiet 12-14
- quote 12-17
- rt_path 12-17
- spl_to_object 12-12
- used with --make_program 12-16
- compiling PepperCode
 - .splrc 12-7
 - command-line rules 12-6, 12-11
 - HP_UX example 12-4
 - installation and configuration issues 12-6
 - LD_LIBRARY_PATH 12-7
 - necessary files 12-7
 - NT example 12-4
 - OSF/1 and Linux example 12-4
 - running the compiler 12-2
 - Solaris example 12-3
 - to object example 12-1
 - using as C++ compiler 12-16
- context 5-6
 - example 5-35
 - multiple 5-34
 - understanding 5-33
- CONTINUE 6-7
- continue statement 6-6
- constraints
 - adding 11-32
- cpp_function 6-7
 - error 14-20
- cpp_to_object 12-13
- CREATE_MULTIPLE_INHERITED_SUBCLASS 10-12
- CREATE_NAME_FROM_OSET 10-12
- CREATE_OBJECT 10-13
- CREATE_SUBCLASS 10-13
- CURRENT_TIME 10-13

D

- data type errors 14-15
- data types 3-11
- DATE_TO_STRING 10-13
- debug 12-13
- debugging 14-1
 - action debug tracing 14-23
 - action interpreter 14-21
 - actions used for debugging 14-41
 - command files 14-55
 - common mistakes 14-1
 - debugging action categories 14-43
 - deciding which debugging actions 14-45
 - descriptions of debugging actions 14-46
 - functions used for debugging 14-31
 - MSG function 14-29
 - running debugging actions 14-42
 - setting message level 14-42
- declaration incomplete error 14-18
- declarations
 - cpp_function
 - Not BREAK or CONTINUE 6-7
- declarations & definitions
 - class 4-6
- default (no option switch) 12-11
- defaults
 - adding to a class 11-2
 - classes 4-5
 - for uninitialized slots 2-5
- define <macroname>=<value> 12-16
- DELETE_OBJECT 10-13
- DESCRIBE 10-13
- doc 3-9, 12-15
 - generating for an .spl file 3-11
- documentation
 - adding and retrieving 11-23
- documentation comments 3-7
- dot notation
 - slots 4-6
 - use in expressions 6-9
- dump functions 10-44
- DUMP_DATE 10-45
- DUMP_FLOAT 10-45
- DUMP_INT 10-45
- DUMP_NEWLINES 10-45
- DUMP_RESET_STATUS 10-46
- DUMP_SPACES 10-45
- DUMP_STRING 10-46
- DUMP_TEST_RESET_STATUS 10-46
- DUMP_TIME 10-46

E

- enumerations

- use in loops 6-7
 - using the same constant names 3-13
- error messages 14-9
 - #document 3-8
- execute
 - actions 5-18
 - automatic 5-42
 - parameter behavior 5-11
 - passing action outputs 5-20
- execute statement 6-5
- exists
 - use with associative arrays 8-2
 - use with nonassociative arrays 8-4
- EXP 10-13
- expressions
 - comparisons 10-23

F

- FAQ 14-6
- files
 - dumping information to 10-44
 - type for PepperCode code 3-1
- float
 - indexing arrays with 8-8
- FLOAT_TO_INT 10-13
- FLOAT_TO_STRING 10-14
- foreach
 - arrays 8-8
 - scope 3-6
- foreach statement 6-3
 - break and continue 6-6
 - enumerations 6-7
 - osets 7-5, 7-6
- forward action declaration 5-15
- forward class declarations 4-6
- forward declarations
 - using #include instead of 3-4
- functions 10-11
 - ADD_TO_HISTORY_VALUE 10-43
 - ADD_TO_HISTORY_VALUE_ON_CALEDAR 10-42
 - ANALYZE_HISTORY 10-40
 - AREA_UNDER_CURVE 10-39
 - arrays 8-5
 - CLOSE_DUMP_FILE 10-44
 - CREATE_MULTIPLE_INHERITED_SUBCLASS 10-12
 - CREATE_NAME_FROM_OSET 10-12
 - CREATE_OBJECT 10-13
 - CREATE_SUBCLASS 10-13
 - CURRENT_TIME 10-13
 - DATE_TO_STRING 10-13
 - DELETE_OBJECT 10-13
 - DESCRIBE 10-13
 - dump 10-44
 - DUMP_DATE 10-45
 - DUMP_FLOAT 10-45
 - DUMP_INT 10-45
 - DUMP_NEWLINES 10-45
 - DUMP_RESET_STATUS 10-46
 - DUMP_SPACES 10-45
 - DUMP_STRING 10-46
 - DUMP_TEST_RESET_STATUS 10-46
 - DUMP_TIME 10-46
 - EXP 10-13
 - FLOAT_TO_INT 10-13
 - FLOAT_TO_STRING 10-14
 - functions used for debugging 14-31
 - GET_ALLOCATED_CHAMBERS 10-44
 - GET_CLASS_BY_NAME 10-14
 - GET_CLASS_OF_INSTANCE 10-14
 - GET_DATE_OF_NEXT_NEGATIVE_VALUE 10-43
 - GET_DATE_OF_PREVIOUS_NOT_ENOUGH 10-43
 - GET_DESCENDENTS 10-15
 - GET_DIRECT_DESCENDANTS 10-16
 - GET_END_OF_HISTORY 10-35
 - GET_HISTORY_VALUE 10-38
 - GET_INITIAL_AMOUNT 10-39
 - GET_INSTANCE_BY_NAME 10-16
 - GET_INVENTORY_AREAS 10-42
 - GET_MSG_LEVEL 10-16
 - GET_NAME_OF_CLASS 10-16
 - GET_NULL_INSTANCE 10-16
 - GET_OVERALLOCATED_CHANGERS 10-36
 - GET_RANDOM_SEED 10-16
 - GET_TYPED_INSTANCE 10-17
 - history 10-35
 - INSTANCE_EXISTS_IN_LIST 10-17
 - IS_ASSERTED 10-32
 - IS_LEGAL_CALEDAR_TIME_FOR_SPLITTING 10-41
 - LIST_FILES_IN_DIRECTORY 10-17
 - MAX_QUANTITY_OVERALLOCATED 10-36
 - MIN_HISTORY_VALUE 10-39
 - MOST_OVERALLOCATED_CHANGERS 10-39
 - MSG 10-18
 - NEXT_CALEDAR_BREAK 10-42
 - NEXT_LEGAL_CALEDAR_TIME 10-40
 - NEXT_TIME_TO_TRY 10-37
 - NLSPRINT 10-28
 - NLSTR 10-28
 - NLSTRCMP 10-28
 - NUMBER_OF_AREAS_SHORT 10-43
 - OBJECT_IS_ALIVE 10-18
 - OPEN_DUMP_FILE 10-44
 - postpone side effects 10-30
 - PREVIOUS_CALEDAR_BREAK 10-42
 - PREVIOUS_LEGAL_CALEDAR_TIME 10-41
 - PRINF 10-18

PRINTF 10-18
 QUANTITY_OF_HISTORY_EXCEEDS 10-36
 QUANTITY_OF_HISTORY_EXISTS 10-36
 QUERY 10-34
 QUERY_OSET 10-34
 RANDOM 10-19
 RANDOMIZE_SEED 10-19
 REGMATCH 10-19
 RENAME_FILE 10-19
 RESYNCH_SE 10-31
 RETRACT_AND_POSTPONE 10-31
 SET_MSG_LEVEL 10-19
 SET_RANDOM_SEED 10-20
 SORT_BY_NAME 10-20
 STATE_EXISTS 10-40
 STATE_NEXT_TIME_TO_TRY 10-38
 STRERROR 10-20
 STRING_COMPARE 10-20
 STRING_CONCAT 10-20
 STRING_TO_DATE 10-20
 STRING_TO_INT 10-22
 STRLEN 10-20
 STRPRINT 10-21
 STRRPL 10-21
 STRSTR 10-21
 TIME_BETWEEN_TWO_POINTS_FOR_CALEN
 DAR 10-41
 TYPEP 10-22
 upstairs objects 10-24
 UPSTAIRS_CLASS 10-25
 UPSTAIRS_INSTANCE 10-24
 UPSTAIRS_OSET_CLASS 10-25
 UPSTAIRS_OSET_INSTANCE 10-25
 using string functions 10-27
 using upstairs objects functions 10-25

G

GET_ALLOCATED_CHAMBERS 10-44
 GET_CLASS_BY_NAME 10-14
 GET_CLASS_OF_INSTANCE 10-14
 GET_DATE_OF_NEXT_NEGATIVE_VALUE
 10-43
 GET_DATE_OF_PREVIOUS_NOT_ENOUGH
 10-43
 GET_DESCENDANTS 10-16
 GET_DESCENDENTS 10-15
 GET_END_OF_HISTORY 10-35
 GET_HISTORY_VALUE 10-38
 GET_INITIAL_AMOUNT 10-39
 GET_INSTANCE_BY_NAME 10-16
 GET_INVENTORY_AREAS 10-42
 GET_MSG_LEVEL 10-16
 GET_NAME_OF_CLASS 10-16
 GET_NULL_INSTANCE 10-16
 GET_OVERALLOCATED_CHANGES 10-36
 GET_RANDOM_SEED 10-16

GET_TYPED_INSTANCE 10-17
 global
 scope 3-6

H

--header
 use with --doc 3-11
 header files
 differences for PepperCode prior 8.0 3-5
 including 3-1
 --header_only 12-15
 histories 9-1
 history functions 10-35

I

if-else statement 6-2
 --include 12-16
 --include <directory> 12-13
 incomplete declarations
 avoiding 5-6
 inheritance
 multiple 4-5
 inout 14-15
 inout parameter 5-7
 input parameter 5-7
 input variable
 error assigning to 14-20
 installation
 compiler options 12-8
 instance names 4-12
 instance slot 4-9
 INSTANCE_EXISTS_IN_LIST 10-17
 IS_ASSERTED 10-32
 IS_LEGAL_CALENDAR_TIME_FOR_SPLITTING
 10-41

L

LD_LIBRARY_PATH 12-7
 leave
 assignment 6-5
 leave statement 6-5
 --lib_tag 12-14
 LIST_FILES_IN_DIRECTORY 10-17
 lists
 arrays 8-1
 osets 7-1
 LKN4049 error 14-12
 local
 scope 3-6
 --loud 12-14

M

- make_implib 12-13
- make_library 12-12
- make_program 12-11
- MAX_ANALYZE_HISTORY 10-40
- MAX_QUANTITY_OVERALLOCATED 10-36
- methods 5-21
 - adding slots 11-28
 - customizing 11-26
 - example 5-21, 5-23, 5-27
 - including object 11-19
 - input and output parameters 11-19
 - replacing standard method actions 11-26
 - using casting 11-19
 - writing actions that dispatch 11-18
- MIN_HISTORY_VALUE 10-39
- mismatch between #document error 14-12
- mismatch in parameter error 14-12
- missing transaction name error 14-13
- MOST_OVERALLOCATED_CHANGERS 10-39
- MSG 10-18
- MSG function 14-29

N

- naming conventions 3-15
- National Language Support 10-27
- NEXT_CALENDAR_BREAK 10-42
- NEXT_LEGAL_CALENDAR_TIME 10-40
- NEXT_TIME_TO_TRY 10-37
- nlscollect 10-29
- NLSPRINT 10-28
- NLSTR 10-28
- NLSTRCMP 10-28
- no such file or directory error 14-13
- no_context 5-3, 5-6
 - using 5-35, 11-7
- no_header 12-15
- no_main 12-17
- no_object 12-13
- no_rt 12-17
- no_warn 12-14
- not found error 14-13
- nothing named in scope error 14-14
- notice comments 3-11
- Null_Instance 4-11
- NUMBER_OF_AREAS_SHORT 10-43

O

- object 4-1
- object model
 - differences in PepperCode and C++ 4-3
- OBJECT_IS_ALIVE 10-18

- objects
 - temporary 4-9
- OPEN_DUMP_FILE 10-44
- operators 10-1
- optimize 12-13
- osets 7-1
 - action parameters 7-4
 - assignment statement 7-1
 - example of functions 7-2
 - foreach 7-6
 - foreach and while 7-5
 - operators and functions 7-1
- output 14-15
- output parameter 5-7
- output variables
 - checking for an action 11-8

P

- parameter
 - matching parameter lists 5-6
- Parameter %s should be output or inout not %s *See*
Error Message Reference in back of document
- parameters
 - action
 - no longer static 5-12
 - avoiding static action parameters 11-7
 - behavior when executes 5-11
 - casting 11-5
 - context, no_context, readonly 5-6
 - default values 5-9
 - grouping 11-9
 - local parameter error setting 14-10
 - non-local action 5-9
 - osets 7-4
 - required 5-8
 - schemas 5-13
- Parent_Task
 - example 11-3
- parse error 14-15
- PeopleBooks
 - CD-ROM, ordering iii
 - printed, ordering iii
- PepeprCode
 - functions 10-11
- PepperCode
 - compared to C/C++ actions 1-4
 - compared to C/C++ classes 1-3
 - data types 3-11
 - debugging 14-1
 - diagram of example 1-3
 - diagram of running 1-2
 - getting started 2-1
 - Getting Started with Compiling 12-1
 - making queris from 10-33
 - naming conventions 3-15

- operators 10-1
- overview 1-1
- performance considerations 3-14
- sample construct 2-1
- sample construct diagram 2-4
- sample construct explained 2-4
- syntax 13-1
- performance considerations 3-14
- Planning software
 - customizing 1-5
- postpone side effect functions 10-30
- predefined classes 4-10
- preprocessor 12-13
- PREVIOUS_CALENDAR_BREAK 10-42
- PREVIOUS_LEGAL_CALENDAR_TIME 10-41
- PRINF 10-18
- PRINTF 10-18
- purify 12-17

Q

- quantity 12-17
- QUANTITY_OF_HISTORY_EXCEEDS 10-36
- QUANTITY_OF_HISTORY_EXISTS 10-36
- queries
 - making form PepperCode 10-33
- QUERY 10-34
- QUERY_OSET 10-34
- quiet 12-14
- quote 12-17

R

- RANDOM 10-19
- RANDOMIZE_SEED 10-19
- readonly 5-6
- refman 12-1
- REGMATCH 10-19
- RENAME_FILE 10-19
- required 5-8
- required error message 14-19
- RESYNCH_SE 10-31
- RETRACT_AND_POSTPONE 10-31
- RPS_IMPORT 11-22
- rt_path 12-17

S

- sanity checks 14-61
- schemas 5-13
 - example 5-14
- scope 3-6
- SET_EPSILON 10-6
- SET_FLOAT_FORMAT 10-6
- SET_MSG_LEVEL 10-19

- SET_RANDOM_SEED 10-20
- side effects 9-1
 - postponing 10-30
- side_effect slot use 4-2
- slot clause list statements 4-7
- slots
 - adding attributes to previous slot 4-7
 - adding to methods 11-28
 - attributes of 4-1
 - declaration statement 4-8
 - default values 4-2
 - defaults 2-5
 - dot notation 4-6
 - forming the slots belonging to a class 4-1
 - method implementation 5-21
 - naming 11-1
 - side_effect use 4-2
 - slot clause list statements 4-7
 - specializing 4-2, 4-5
 - storage of values 4-9
- SORT_BY_NAME 10-20
- slots 11-2
- spl_main 5-42
- spl_to_object 12-12
- splsh 12-1
- STATE_EXISTS 10-40
- STATE_NEXT_TIME_TO_TRY 10-38
- statements
 - arrays 8-8
 - assignment 6-1
 - break 6-6
 - continue 6-6
 - execute 6-5
 - foreach 6-3
 - if-else 6-2
 - leave 6-5
 - succeed 6-5
 - while 6-2
- static action parameters
 - avoiding 11-7
- STRERROR 10-20
- string functions
 - using 10-27
- STRING_COMPARE 10-20
- STRING_CONCAT 10-20
- STRING_TO_DATE 10-20
- STRING_TO_INT 10-22
- STRLEN 10-20
- STRPRINT 10-21
- STRRPL 10-21
- STRSTR 10-21
- succeed statement 6-5

T

- Target of 14-17
- temporary_instance 4-9

- TIME_BETWEEN_TWO_POINTS_FOR_CALENDAR 10-41
- transaction logs 5-42
- transactions
 - action schema 11-12
 - error checking 11-13
 - inputs not to use 11-12
 - naming 11-12
 - using input parameter defaults 11-12
 - writing 11-11
- translation tables
 - collecting strings for 10-29
- Troubleshooting 14-6
- TYPEP 10-22
 - example 10-23

U

- Undefined symbol spl_action_info_abc 14-18
- Unterminated string literal. 14-19
- upstairs objects

- functions 10-24
- upstairs objects functions
 - using 10-25
- UPSTAIRS_CLASS 10-25
- UPSTAIRS_INSTANCE 10-24
- UPSTAIRS_OSET_CLASS 10-25
- UPSTAIRS_OSET_INSTANCE 10-25

W

- while statement 6-2
 - break and continue 6-6
 - enumerations 6-7
- while statements
 - osets 7-5
- methods 11-18

Z

- zero and null instance 4-11