

IBM VisualAge TeamConnection



User's Guide

Version 2.0

IBM VisualAge TeamConnection



User's Guide

Version 2.0

Third Edition (January 1998)

Note

Before using this document, read the general information under "Notices" on page xi.

This edition applies to Version 2.0 of the licensed program IBM TeamConnection and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications by phone or fax. The IBM Software Manufacturing Company takes publication orders between 8:30 a.m. and 7:00 p.m. eastern standard time (EST). The phone number is (800) 879-2755. The fax number is (800) 284-4721.

You can also order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Attn: Information Development
Department T99B/Building 062
P.O. Box 12195
Research Triangle Park, NC, USA 27709-2195

You can fax comments to (919) 254-0206.

If you have comments about the product, address them to:

IBM Corporation
Attn: Department TH0/Building 062
P.O. Box 12195
Research Triangle Park, NC, USA 27709-2195

You can fax comments to (919) 254-4914.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1992, 1995, 1996, 1997. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Figures	ix
Notices	xi
Trademarks	xiii
About this book.	xv
How this book is organized	xv
Conventions	xv
Tell us what you think	xvi
<hr/>	
Part 1. Introducing TeamConnection	1
Chapter 1. An introduction to TeamConnection	3
TeamConnection definitions	4
TeamConnection's client/server architecture	4
TeamConnection database	5
Interfaces	5
Families	6
Users and host lists.	6
Parts	6
Components	7
Releases.	7
Work areas	8
Drivers	9
Defects and features	9
Processes	9
Build	11
Packaging	11
Roles people play	11
<hr/>	
Part 2. Developing a product using TeamConnection	13
Chapter 2. Getting familiar with the TeamConnection client interfaces	15
Using the GUI	15
Starting the GUI	15
Stopping the GUI.	16
Performing tasks with the GUI	16
Using the Settings notebook	17
Online help information	18
Using the command line interface	19
Using the web client	20
Chapter 3. The basics of using TeamConnection	23
Laying the groundwork.	23
Authority to perform tasks	24
Finding objects within TeamConnection	25
Finding parts	25
Using work areas	26
Naming your work areas	26
Creating parts	27
Naming your parts	27

Preparing to build your parts	28
Working with parts	28
Working in serial or concurrent development mode	28
Working with common parts	29
Getting parts from TeamConnection	30
Checking parts in to TeamConnection	31
Finding different versions of TeamConnection objects	31
Versioning releases	32
Versioning work areas	32
Versioning drivers	33
Versioning parts	34
Working with defects and features	34
Testing and verifying part changes	35
 Chapter 4. The states of TeamConnection objects	 37
Defects and features	37
The states of work areas	40
The states of drivers	42
Verification and test records	44
 Chapter 5. Working with no component or release processes	 47
Working in serial development	47
Accepting a defect	48
Creating a work area	49
Checking out a part	50
Searching for a part	51
Checking in a part	53
Verifying and testing part updates	54
Freezing the work area	58
Refreshing the work area	59
Building the application	60
Integrating the work area	61
Closing a defect	62
Working in concurrent development	63
Refreshing the work area	63
Integrating the work area	64
Reconciling differences	65
 Chapter 6. Working with component and release processes	 69
Moving through design, size, and review	70
Changing defect ownership	70
Accepting a defect	71
Approving the fix	72
Checking out a part	73
Verifying the changes	74
Freezing the work area	75
Building the application	76
Accepting fix records	77
Integrating changed parts into a release	78
Adding a driver member	78
Reconciling the differences	79
Refreshing the driver	81
Building the driver	82
Restricting the driver	83
Integrating the parts	84
Completing the driver	85

Testing the built application	85
Using a configured process	86
Retrieving a past version of a part	87

Part 3. Using TeamConnection to build applications 91

Chapter 7. Basic build concepts	93
The physical structure of the build function	93
The build object model.	95
Parent-child relationships in a build tree	96
Working with a build tree	98
Putting the pieces together	99
 Chapter 8. Starting and stopping the servers	 101
Setting up the mail facility	101
Starting the servers	101
Starting servers from the Family Administrator GUI	101
Starting build servers using teamcbld	102
Caching and the build directories	103
An MVS build server	104
Starting a build agent for an MVS build server	105
Creating build startup files	107
Startup file for build servers	107
Startup file for build agents	108
Stopping the servers	108
A build server	108
An MVS build server	109
 Chapter 9. Working with build scripts and builders	 111
Creating a builder	111
Writing a build script	115
Passing parameters to a build script.	115
Writing a simple build script	116
Writing an executable file for a build script	117
Testing a build script	118
Modifying the contents of a build script.	118
Putting a builder to work	119
Removing a builder from a part	120
Working with VisualAge C++ and Templates	120
 Chapter 10. Working with MVS build scripts and builders	 121
Creating a builder for MVS builds.	121
Writing an MVS build script	125
File name conversions for MVS	125
Passing parameters to an MVS build script	126
TeamConnection syntax for MVS build scripts	127
Supported JCL syntax	128
Example of a build script for a C compile	129
Example of a build script for a COBOL compile	131
Example of a build script for a link	132
 Chapter 11. Working with parsers	 135
Creating a parser	135
Putting a parser to work	137
Removing a parser from a part	138
Writing a parser command file	139

Chapter 12. Building an application: an example	141
Starting the build processors and build agents	142
Creating builders and parsers	143
Creating the build tree for the application	143
Starting the build on the client	147
Determining the build scope.	149
Adding the job to the job queue	151
Picking up the work orders	151
Putting the build processors to work.	151
Putting the build scripts to work	151
Finishing the job and reporting the results to the user	152
Monitoring the progress of a build	152
Running a build in spite of errors	153
Building all parts, regardless of build times	153
Finding out which parts will be built	154
Canceling a build.	154
More sample build trees	155
Defining multiple outputs from a single build event	155
Synchronizing the build of unrelated parts	156

Part 4. Using TeamConnection to package products 157

Chapter 13. Using TeamConnection to package a product	159
Setting up your build tree for packaging	160
Setting up a build tree for the gather tool	160
Setting up a build tree for the NVBridge tool.	162
Setting up a build tree for other distribution tools	163
Chapter 14. Using the Gather tool	165
Using the teamcpak command for the Gather tool.	166
Command line flags.	166
Examples of the teamcpak gather command	167
Writing a package file for the Gather tool	168
Syntax rules for a Gather package file	168
Chapter 15. Using the NVBridge tool	173
Using the teamcpak command for NVBridge.	174
Command line flags.	175
Examples of the teamcpak nvbridge command.	176
Writing a package file for NVBridge	176
Syntax rules for an NVBridge package file	177
Keywords for an NVBridge package file	177
Problem determination for NVBridge.	185
NVBridge utilities.	186
FHPSTAT	187
FHPOBDEL.	187
FHPOBMON	187
FHPOBDIF	188
FHPISCAT	189
FHPICAT.	189
FHPUCAT	190
FHPMCAT	191
FHPVERIF	191
FHPRQPUR	192
FHPRQMON	192
FHPTRVER.	193

FHPTRPUR	194
Chapter 16. Using the Tivoli/Courier packaging tool.	195
Using the teamcpak command with Tivoli/Courier	195
Command line flags.	196
Example of the teamcpak softdist command	196
Writing a package file for Tivoli/Courier.	197
Syntax rules for a Tivoli/Courier package file	197
Keywords for a Tivoli/Courier package file	197
Problem determination for the Tivoli/Courier tool	200
Sample package file	200
Appendix A. Environment Variables	203
Setting environment variables	207
Appendix B. Importing makefile information into TeamConnection	209
Creating a rules file	210
Appendix C. TeamConnection Merge	213
Appendix D. Enabling a Workframe project for TeamConnection.	215
Creating a TeamConnection-enabled Workframe project	215
Setting up your project options.	215
Using your TeamConnection Workframe project	216
Project actions.	216
Part actions.	216
Using your project: a simple scenario	217
Appendix E. Enabling and Using the ENVY/Manager-TeamConnection Bridge.	219
Overview of the ENVY/Manager-TeamConnection Bridge	219
Scope of this documentation	219
Description of the ENVY/Manager-TeamConnection Bridge	220
Preparing to use the ENVY/Manager-TeamConnection Bridge	221
Setting up the bridge environment	221
Installing and activating the ENVY/Manager-TeamConnection Bridge.	222
Using the ENVY/Manager-TeamConnection Bridge	224
Setting default properties.	224
Exporting ENVY components to TeamConnection	227
Importing ENVY components from TeamConnection	229
Using the ENVY/Manager-TeamConnection Bridge: a simple scenario for	
VisualAge Generator developers	230
Scenario assumptions	230
Exporting ENVY components to TeamConnection	230
Object mapping in TeamConnection	231
Build generation	232
Making a change to a member.	233
Appendix F. Source Code Control User's Guide	235
Differences between other source code control providers and TeamConnection .	235
Projects vs Families.	235
Installing the TeamConnection source code control DLL	236
Connecting TeamConnection to Visual Basic 4.0	236
Removing the TeamConnection Source Code Control DLL	236
Using TeamConnection as your source code control provider	236
Before you start	237

Opening a project	237
Full features of TeamConnection	238
Appendix G. Supported keywords.	241
Appendix H. Authority and notification for TeamConnection actions	243
Appendix I. Sample REXX execs, build scripts, and parsers	259
Sample REXX execs	259
Sample build scripts	262
Sample parsers	263
Sample package files	263
Appendix J. Program specifications for TeamConnection version 2.0 . . .	265
Customer support	267
Bibliography	269
IBM VisualAge TeamConnection library	269
Tool Builder's Development Kit.	269
TeamConnection Technical reports	270
ObjectStore.	270
IBM Exchange library	271
Related publications	271
Glossary	273
Index	281

Figures

1.	A sample TeamConnection client/server network	5
2.	Sample of a component hierarchy.	7
3.	Parts, releases, and components	8
4.	Tasks window	16
5.	Components window	24
6.	Accept Defects window	48
7.	Create Work Areas window	49
8.	Check Out Parts window	50
9.	Part Filter window	51
10.	Edit Task List window	52
11.	Check In Parts window.	53
12.	Build Parts window	55
13.	Extract Parts window	56
14.	Check Out Parts	57
15.	Check In Parts window.	58
16.	Freeze Work Areas window	59
17.	Refresh Work Areas window.	60
18.	Build Parts window	61
19.	Integrate Work Areas window	62
20.	Verify Defects window	63
21.	Refresh Work Areas window.	64
22.	Integrate Work Areas window	65
23.	Reconcile Collision Record window	66
24.	Modify Defect Owner window	71
25.	Accept Defects window	72
26.	Accept Approval Records window	73
27.	Check Out Parts window	74
28.	Check In Parts window.	75
29.	Freeze Work Areas window	76
30.	Build Parts window	77
31.	Complete Fix Records window	78
32.	Add Driver Members window	79
33.	Fix Work Areas window	80
34.	Activate Fix Records window	80
35.	Refresh Work Areas window.	81
36.	Refresh Drivers window	82
37.	Build Parts window	83
38.	Restrict Drivers window	84
39.	Commit Drivers window	84
40.	Complete Drivers window.	85
41.	Accept Test Records window	86
42.	The physical structure of TeamConnection	94
43.	Sample build object model for msgcat.exe	97
44.	The build tree for the hello application	98
45.	Two versions of a build tree	99
46.	Create Builder window	112
47.	Matching environment values	114
48.	Modify Part Properties window	119
49.	Modify Part Properties window	120
50.	Create Builder window	122
51.	Matching environment values	124
52.	A JCL fragment for an MVS compile	130
53.	A JCL fragment converted to a build script	131

54.	Create Parser window	136
55.	Modify Part Properties window	138
56.	Modify Part Properties window	139
57.	Sample build tree	141
58.	Sample build object model for msgcat.exe	142
59.	Create Parts window	144
60.	Create Parts window	145
61.	Modify Part Properties window	146
62.	Connect Parts window	146
63.	The build tree display	147
64.	Build Parts window	147
65.	Build tree showing build times	150
66.	The build tree for robot.dll.	155
67.	The build tree for robot.app	156
68.	Part of the build tree for robot.app	160
69.	Adding the gather step to the build tree.	162
70.	Adding the NVBridge step to the build tree	163

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the Site Counsel, IBM Corporation, P.O. Box 12195, 3039 Cornwallis Road, Research Triangle Park, NC 27709-2195, USA. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

IBM may change this publication, the product described herein, or both. These changes will be incorporated in new editions of the publication.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	MVS/XA
Common User Access	NetView
CUA	Operating System/2
C/370	OS/2
ENVY*	TeamConnection
IBM	VisualAge
MVS	XGA
MVS/ESA	

* ENVY is a registered trademark of Object Technology International, Inc.

The following terms are trademarks of other companies:

ObjectStore

ObjectStore Design, Inc.

UNIX X/Open Company Limited

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

About this book

This book is part of the documentation library supporting the IBM TeamConnection licensed programs. It is a guide for client users.

For additional information when performing TeamConnection tasks, refer to the *Commands Reference* when entering commands or online help when using the graphical user interface (GUI).

Getting Started with the TeamConnection Clients contains basic information for the client user.

This book is available in PDF format. Because production time for printed manuals is longer than production time for PDF files, the PDF files may contain more up-to-date information. The PDF files are located on the installation CD in directory path `softpubs\enu` (`softpubs/en_us` in UNIX). To view these files, you need a PDF reader such as Acrobat.

How this book is organized

“Part 1. Introducing TeamConnection” on page 1, gives all users an overview of the concepts of TeamConnection and introduces the terminology that is used throughout this book.

“Part 2. Developing a product using TeamConnection” on page 13, describes the different interfaces and basic TeamConnection tasks. It uses scenarios to explain how to do many tasks.

This part is for everyone using TeamConnection to do daily work. The information is meant for both the person who uses the command line interface and the person who uses the GUI, as instructions for both are provided.

“Part 3. Using TeamConnection to build applications” on page 91, tells how to use the TeamConnection build function. For information in installing and administering the build function, refer to the *Administrator's Guide*

“Part 4. Using TeamConnection to package products” on page 157, tells how TeamConnection helps you automate the packaging and distribution of your application.

, contains various pieces of information that you can refer to as you plan for and use TeamConnection.

Information on customer service, a glossary, and a bibliography are included at the back of this book.

Conventions

This book uses the following highlighting conventions:

- *Italics* are used to indicate the first occurrence of a word or phrase that is defined in the glossary. They are also used for information that you must replace.
- **Bold** is used to indicate items on the GUI.
- Monospace font is used to indicate exactly how you type the information.
- File names follow Intel conventions: **mydir/myfile.txt**. AIX and HP-UX users should render this file name **mydir/myfile.txt**.

Tips or platform specific information is marked in this book as follows:



Shortcut techniques and other tips



IBM VisualAge TeamConnection for OS/2



IBM VisualAge TeamConnection for Windows 3.1



IBM VisualAge TeamConnection for Windows/NT



IBM VisualAge TeamConnection for Windows 95



IBM VisualAge TeamConnection for AIX



IBM VisualAge TeamConnection for HP-UX



IBM VisualAge TeamConnection for Solaris

Tell us what you think

In the back of this book is a comment form. Please take a few moments to tell us what you think about this book. The only way for us to know if you are satisfied with our books or if we can improve their quality is through feedback from customers like you.

Part 1. Introducing TeamConnection

Chapter 1. An introduction to TeamConnection	3
TeamConnection definitions	4
TeamConnection's client/server architecture	4
TeamConnection database	5
Interfaces	5
Families	6
Users and host lists	6
Parts	6
Components	7
Releases	7
Work areas	8
Drivers	9
Defects and features	9
Processes	9
Build	11
Packaging	11
Roles people play	11

This section presents an overview of the TeamConnection product. The information in this section should be read and understood by everyone who is going to work with TeamConnection.

Additional conceptual information is provided in Parts 3, 4, and 5.

Chapter 1. An introduction to TeamConnection

TeamConnection provides an environment and tools to make software development run smoothly, whether your development team is small or large. Using TeamConnection, you can communicate with and share data among team members to keep up with the many tasks in the development life cycle, from planning through maintenance.

What does TeamConnection do for you? It takes care of the following:

- *Configuration management*: the process of identifying, organizing, managing, and controlling software modules as they change over time. This includes controlling access to your software modules and providing notification to team members as software modules change.
- *Release management*: the logical organization of objects that are related to an application. The release provides a logical view of objects that must be built, tested, and distributed together. Releases are versioned, built, and packaged.
- *Version control*: the tracking of relationships among the versions of the various parts that make up an application. Version control enables you to build your product using stable levels of code, even if the code is constantly changing. It provides control over which changes are available to everyone and, optionally, allows more than one developer at a time to update a part.
- *Change control*: the controlling of changes to parts that are stored in TeamConnection. TeamConnection keeps track of any part changes you make and the reasons you make them. Your development team can build releases with accuracy and efficiency, even as the parts evolve. The product ensures that the change process is followed and that the changes are authorized. After changes are made, it allows you to integrate the changes and build the application. TeamConnection tracks all changes to the parts across multiple products and environments.

The *change control process* is configurable. Your team can decide how strict the change control should be, from loose to very tight. You can also adjust the level of control as you move through a development cycle.
- *Build support*: the function that enables you to define the structure of your application and then to create it within TeamConnection from your input parts. Independent steps in a build can run in parallel on different servers, thus reducing your build time. You can build applications for platforms in addition to the one TeamConnection runs on—currently, you can use TeamConnection to build applications on AIX, HP-UX, OS/2, Windows NT, Windows 95, Solaris, and MVS.
- *Packaging support*: the preparation of your application for electronic distribution to other users.

TeamConnection provides an open information model for sharing data between a set of integrated tools using TeamConnection. This object-based information model enables an extensible architecture, thus ensuring continued support for new versions of existing tools, as well as new tools that are brought into the repository environment. This support includes the previously mentioned standard services across all objects stored in the information model. TeamConnection's information model and tool builder's functions are provided separately in the Tool Builder's Development Kit. See the bibliography at the back of this book for more information.

IBM Exchange for OS/2, which is a feature of TeamConnection, provides the functions necessary to migrate existing model information into TeamConnection. For more information about IBM Exchange, refer to the *IBM Exchange User's Guide*

This chapter defines the basic terms and concepts you need to make the most of TeamConnection. Read this chapter first; then decide which information you need next:

Topic and description	Page
Developing products using TeamConnection: <ul style="list-style-type: none">• Getting familiar with the interfaces• The basics of using TeamConnection• More about <i>defects</i> and <i>features</i>• Following TeamConnection processes	14
Using TeamConnection to build applications: <ul style="list-style-type: none">• Build concepts• Installing build agents and processors• Working with build scripts and builders• Working with <i>parsers</i>• Building an application	92
Packaging applications: <ul style="list-style-type: none">• Using the packaging function• Using the Gather utility• Using the NVBridge utility	"Chapter 13. Using TeamConnection to package a product" on page 159

TeamConnection definitions

The following definitions are in logical order rather than alphabetical. provides additional information about these terms.

TeamConnection's client/server architecture

Figure 1 on page 5 is an example of a network of TeamConnection clients and servers.

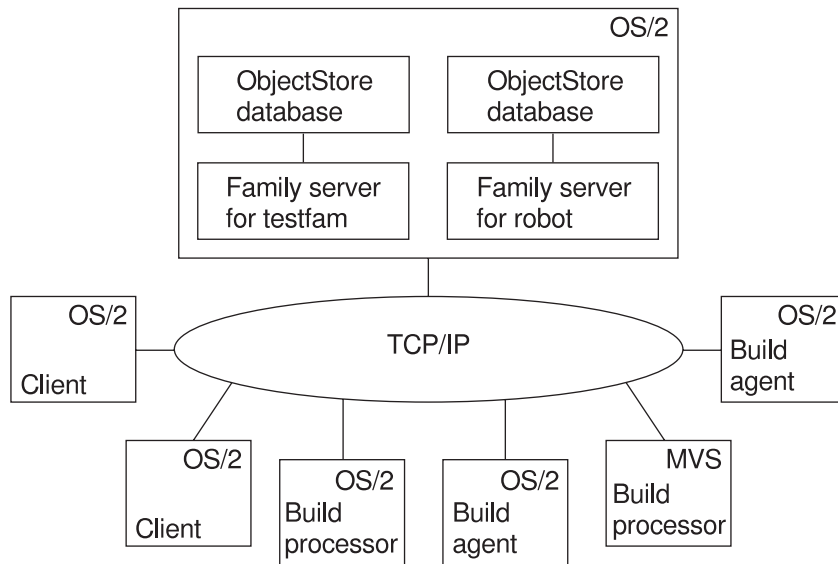


Figure 1. A sample TeamConnection client/server network

TeamConnection *family servers* control all data within the TeamConnection environment. Data stored in a family server's database includes:

- Text objects, such as source code and product documentation
- Binary objects, such as compiled code
- Modeled objects that are stored in the information model by tools such as VisualAge Generator
- Other TeamConnection objects that are metadata about the other objects

Build agents and *build processors* work together to perform product builds. We sometimes refer to the combination of a build agent and its connected build processor as a build server.

A TeamConnection *client* gives team members access to the development information and parts stored on the database server.

TeamConnection database

TeamConnection is built on Object Design, Inc.'s ObjectStore database. The TeamConnection server communicates with the ObjectStore database through an ObjectStore server.

Interfaces

TeamConnection provides the following interfaces that you can use to access data:

- A graphical user interface based on Common User Access (CUA).
- A command line interface that lets you type TeamConnection commands from a prompt or from within TeamConnection
- A web client, that you access through your web browser.

You can use any interface to do your TeamConnection work, or you can switch among them. This book usually gives instructions for using both interfaces.

For more information, see “Chapter 2. Getting familiar with the TeamConnection client interfaces” on page 15.

Families

A *family* represents a complete and self-contained collection of TeamConnection users and development data. Data within a family is completely isolated from data in all other families. One family cannot share data with another.

Users and host lists

Users are given access to the TeamConnection development data in a specific family through their *user IDs*. Each family has at least one *superuser*, who has privileged access to the family. The superuser gives other users the *authority* to perform some set of *actions* on particular data. Depending on the authority granted to a user, that user might in turn be able to grant some equal or lesser level of authority to other users. However, the ability to grant authority for some actions is reserved to the superuser. There are no actions which the superuser cannot perform.

For host-based authentication, each user ID is associated with a *host list*, which is a list of client machine addresses from which the user can access TeamConnection when using that ID.

A single user can access TeamConnection from multiple systems or logins. Likewise, a single system login can act on behalf of multiple users. The set of authorized logins for a TeamConnection user ID makes up the user's host list.

It is also possible to authenticate users through the use of passwords, either in place of host lists, or as an alternative form of authentication.

Parts

TeamConnection *parts* are objects that users and tools store in TeamConnection. They include text objects, binary objects, and modeled objects. These parts can be stored by the user or the tool, or they can be generated from other parts, such as when a linker generates an executable file. Parts can also be groupings of other TeamConnection objects for building and distribution, or simply for convenient reference. Common part actions include the following:

Create

To store a part from your workstation on the server; from that time on, TeamConnection keeps track of all changes made to the part. Or, to create a part to use as a place holder to store the output of a build.

Check out

To get a copy of a part so that you can make changes to it.

Check in

To put the changed part back into TeamConnection.

Extract

To get a copy of the part *without* making changes to the *current version* in TeamConnection.

Edit

To change a part from within TeamConnection using a specified editor.

Build To construct an output part from parts that you have defined to TeamConnection as input to the output part.

These are simplified definitions of part actions; there is more about the actions you can perform against parts in “Chapter 3. The basics of using TeamConnection” on page 23 .

The current version of each part is stored in the TeamConnection database, along with previous versions of each part. You can return to previous versions if you need to.

Components

Within each family, development data is organized into groups called *components*. The component hierarchy of each family includes a single top component, called *root*, and *descendants* of that root. Each *child component* has at least one parent component; a child can have multiple parents.

The following figure depicts a component hierarchy.

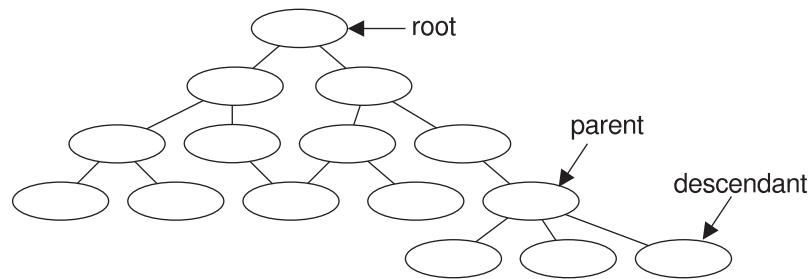


Figure 2. Sample of a component hierarchy

TeamConnection uses components to organize development data, control access to the data, and notify users when certain actions occur. Descendant components inherit access and notification information from ancestor components. Information about the components is stored in the database, including:

- The component's position in its family hierarchy.
- The user who owns the component. The component *owner* is responsible for managing data related to it, including defects or features.
- The users who have access to the component and the level of access each user has. This information makes up the component's *access list*.
- The users who are to be notified about changes to the component. This set of users is called the *notification list*.
- The *process* by which the component handles defects and features.

Releases

An application is likely to contain parts from more than one component. Because you probably want to use some of the same parts in more than one application, or in more than one version of an application, TeamConnection also groups parts into *releases*. A release is a logical organization of all parts that are related to an application; that is, all parts that must be built, tested, and distributed together. Each

time a release is changed, a new version of the release is created. Each version of the release points to the correct version of each part in the release.

Each part in TeamConnection is managed by at least one component and contained in at least one release. One release can contain parts from many components; a component can span several releases. Figure 3 shows the relationships between parts, the releases that contain them, and the components that manage them.

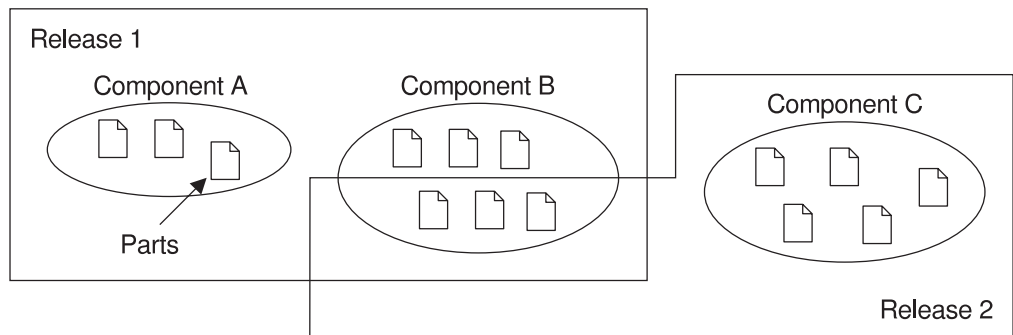


Figure 3. Parts, releases, and components

Each time a new development cycle begins, you can define a separate release. Each subsequent release of an application can share many of the same parts as its predecessor. Thus maintenance of an older release can progress at the same time as development of a newer one. Each release follows a process by which defects and features are handled.

Work areas

A release contains the latest "official" version of each of its parts. As users check parts out of the releases, update them, and then check them back in, TeamConnection keeps track of all of these changes, even when more than one user updates the same part at the same time. To make this possible, TeamConnection uses something called a *work area*.

A work area is a logical temporary work space that enables you to isolate your work on the parts in a release from the official versions of the parts. You can check parts out to a work area, update them, and build them without affecting the official version of the parts in the release. After you are certain that your changes work, you *integrate* the work area with the release (or *commit* the driver that the work area is a member of, if you are using the driver subprocess). The integration makes the parts from your work area the new official parts in the release.

You can do the following with work areas:

- Check out parts from a release
- Update any or all of the checked-out parts
- Get the latest copies of the parts in the release, including any changes integrated by other users
- Get the latest copies of the parts in another work area
- *Freeze* the work area, making a snapshot of the parts as they exist at a particular instant in case you need to return to it later

- Build the parts in the work area
- Move all parts back into the release by integrating the work area

For more information, see “Using work areas” on page 26.

Drivers

A driver is a collector for work areas. You create drivers associated with specific releases so that you can exercise greater control over which work areas are integrated into the release and commit the changes from multiple work areas simultaneously.

When a work area is added to a driver, it is called a *driver member*. A single work area can be a member of more than one driver. By making a work area part of a driver, you associate the parts changed in relation to that work area with the specified driver. These parts must be members of the release associated with the driver.

Drivers enable you to place the following controls over work area integrations:

- Define and monitor prerequisite and corequisite work areas to ensure that mutually dependent changes are integrated in proper order.
- Monitor and resolve conflicting changes to the same part (if you use concurrent development).
- Restrict access to driver members so that they can be changed only by users with proper authority.

Defects and features

A defect is a record of a problem to be fixed. A feature is a record of a request for a functional addition or enhancement. Both are associated with a work area, and both follow the processes defined for the component and release that are associated with the work area. TeamConnection tracks both objects through their life cycles as developers change and commit parts.

You can use defects and features to record problems and design changes for things other than the products you are developing under TeamConnection control. For example, you can use defects to record information about personnel problems, hardware problems, or process problems. You can use features to record proposals for process improvements and hardware design changes.

For more information, see “Working with defects and features” on page 34.

Processes

An application changes over time as developers add features or correct defects. TeamConnection controls these changes according to the *processes* you choose for your application’s components and releases. A process enforces a specific level of control to part changes and ensures that actions occur in a specified order.

Two separate types of processes are defined: component processes, which can be different for each component within a family, and release processes, which apply to

all activities associated with a given release. Component or release processes are built from a number of lower-level processes, or *subprocesses*, that are included with the TeamConnection product.

A defect or feature written against a component moves through successive *states* during its life cycle. The TeamConnection actions that you can perform against it depend on its current state. The component processes define these actions. You can require users to do some, all, or none of the following for tracking defects and features:

dsrFeature

Design, size, and review changes to be made for features

verifyFeature

Verify that the features have been implemented correctly

dsrDefect

Design, size, and review fixes to be made for defects

verifyDefect

Verify that the fixes work

At the release level you can require some, all, or none of the following subprocesses:

track This subprocess is TeamConnection's way of relating all part changes to a specific defect or feature and a specific release. Each work area gathers all the parts modified for the specified defect or feature in one release and records the status of the defect or feature. The work area moves through successive states during its life cycle. The TeamConnection actions that you can perform against a work area depend on its current state.

You must use the *track subprocess* if you want to use any of the other release subprocesses.

approval

This subprocess ensures that a designated *approver* agrees with the decision to incorporate changes into a particular release and electronically signs a record. As soon as approval is given, the changes can be made.

fix This subprocess ensures that as users check in parts associated with a work area, an action is taken to indicate that they have completed their portion. When everyone is done, the owner of the *fix record* (usually the component owner) can change the fix record to complete. The parts are then ready for integration.

driver A *driver* is a collection of all the work areas that are to be integrated with each other and with the unchanged parts in the release at a particular time. The driver subprocess allows you to include these changes incrementally so that their impact can be evaluated and verified before additional changes are incorporated. Each work area that is included in a driver is called a *driver member*.

test The test subprocess guarantees that testing occurs prior to verifying that the fix is correct within the release.

TeamConnection is shipped with several predefined processes. If these do not apply to your organization, you can configure your own processes by defining different combinations of subprocesses.

See “Chapter 4. The states of TeamConnection objects” on page 37 for an explanation of TeamConnection states.

Build

The TeamConnection build function automates the process of building individual parts or entire applications, both in the work group LAN environment and on an enterprise server. This function enables you to reliably and repeatedly build the same output from the same inputs. You can also build different outputs from the same inputs for different environments.

You start a build against an output part that has an associated *builder*. A builder is an object that describes how to translate input parts to get the desired output, such as a linker or compiler. An input part might have an associated parser, which determines the dependencies for the input parts in a build.

The build function does the following:

- Tracks build times of inputs and outputs so that it builds only those parts that are out of date themselves or that have out of date dependants. You can also force a build regardless of the build times.
- Enables you to spread the build over multiple machines running at the same time or into multiple processes running on a single machine, such as on MVS.

For more information, see “Part 3. Using TeamConnection to build applications” on page 91 .

Packaging

Packaging is any of the steps necessary to distribute software and data onto the machines where they are to be used. TeamConnection includes two tools that you can use to automate the electronic distribution of TeamConnection-managed software and data:

Gather

An automated data mover for server or file transfer-based distribution

NVBridge

A bridge utility that automates the installation and distribution of software or data using IBM NetView Distribution Manager/2 as the distribution vehicle

NVBridge

A tool that supports automated distribution between a single NetView DM/2 CC server and its LAN-connected CC clients. It also supports remote distribution to APPC-connected NetView DM/2 servers and mainstream servers.

For more information, see “Part 4. Using TeamConnection to package products” on page 157

Roles people play

Because TeamConnection is extremely flexible, no two projects are likely to use it in the same way, and the jobs that people perform likewise vary. Still, TeamConnection tasks can be grouped into the following general categories:

System administrator

Has *superuser* access to the family server and database administration access to the database management system. This administrator is responsible for the following:

- Installing and maintaining the server
- Maintaining and backing up the database used by TeamConnection

Family administrator

Has superuser access to the family server and database administration access to the database management system. This administrator is responsible for the following:

- Planning and configuring TeamConnection for one or more families
- Managing user access to one or more families
- Maintaining one or more families

Build administrator

This administrator is responsible for the following:

- Setting up and maintaining build servers
- Planning for builds
- Creating builders and parsers
- Starting and stopping build agents and processors
- Defining *pools*
- Monitoring build performance
- Creating driver members
- Committing and completing drivers
- Extracting releases
- Packaging and distributing applications

End user

End users, such as project leaders, programmers, and technical writers, use one or more TeamConnection families to control and maintain application development data.

Part 2. Developing a product using TeamConnection

Chapter 2. Getting familiar with the TeamConnection client interfaces	15
Using the GUI	15
Starting the GUI	15
Stopping the GUI	16
Performing tasks with the GUI	16
Using the Settings notebook	17
Online help information	18
Using the command line interface	19
Using the web client	20
 Chapter 3. The basics of using TeamConnection	23
Laying the groundwork	23
Authority to perform tasks	24
Finding objects within TeamConnection	25
Finding parts	25
Using work areas	26
Naming your work areas	26
Creating parts	27
Naming your parts	27
Preparing to build your parts	28
Working with parts	28
Working in serial or concurrent development mode	28
Working with common parts	29
Getting parts from TeamConnection	30
Checking parts in to TeamConnection	31
Finding different versions of TeamConnection objects	31
Versioning releases	32
Versioning work areas	32
Versioning drivers	33
Versioning parts	34
Working with defects and features	34
Testing and verifying part changes	35
 Chapter 4. The states of TeamConnection objects	37
Defects and features	37
The states of work areas	40
The states of drivers	42
Verification and test records	44
 Chapter 5. Working with no component or release processes	47
Working in serial development	47
Accepting a defect	48
Creating a work area	49
Checking out a part	50
Searching for a part	51
Checking in a part	53
Verifying and testing part updates	54
Extracting a part	55
Checking out the part one more time	56
Checking the part back in	57
Freezing the work area	58
Refreshing the work area	59
Building the application	60

Integrating the work area	61
Closing a defect	62
Working in concurrent development	63
Refreshing the work area.	63
Integrating the work area	64
Reconciling differences	65
 Chapter 6. Working with component and release processes	69
Moving through design, size, and review	70
Changing defect ownership	70
Accepting a defect	71
Approving the fix	72
Checking out a part	73
Verifying the changes	74
Freezing the work area	75
Building the application	76
Accepting fix records	77
Integrating changed parts into a release	78
Adding a driver member	78
Reconciling the differences	79
Returning the work area to the fix state	79
Reactivating the fix record	80
Refreshing the work area.	81
Refreshing the driver	81
Building the driver	82
Restricting the driver	83
Integrating the parts.	84
Completing the driver	85
Testing the built application	85
Using a configured process	86
Retrieving a past version of a part	87

This section is for anyone who uses the TeamConnection client to do daily work. The information is meant for both the person who uses the command line interface and the person who uses the GUI; instructions for both are provided.

All the tasks in this part are done from a client machine.

Before reading this section, you should be familiar with the TeamConnection terminology and concepts presented in “Chapter 1. An introduction to TeamConnection” on page 3.

Chapter 2. Getting familiar with the TeamConnection client interfaces

TeamConnection provides several interfaces that you can use to access data:

- A graphical user interface based on the Common User Access (CUA) architecture
- A command line interface that lets you type TeamConnection commands from a prompt or from within TeamConnection

If you are using Windows 3.1, you must use the entry field at the bottom of the TeamConnection Tasks window just above the user and family names, or use the TeamConnection Command window to enter TeamConnection commands.

- A web client, that you access through your web browser.

You can use any of the interfaces to do all of your TeamConnection work, or you can switch back and forth between the them. You might find that some tasks are easier to do from the GUI or through the web, while others are easier to do from the command line.

The examples throughout “Part 2. Developing a product using TeamConnection” on page 13 give instructions for both GUI and command line interface usage.

This chapter helps you to begin using the TeamConnection client interfaces. It describes the following:

- Using the GUI
 - Starting and stopping the GUI
 - Getting around in the GUI
 - Using the Settings notebook
 - Using the online help that is provided with TeamConnection
- Using the command line interface
- Using the web client

Before you can use TeamConnection, someone in your organization with superuser authority, such as your family administrator, must create for you a unique user ID and a host list entry for the workstation where you installed the client.

Using the GUI

TeamConnection provides a GUI that you can use to do all of your TeamConnection work. To use the GUI efficiently, set your default values in your Settings notebook to suit your working environment, and then become familiar with the Tasks window and how you can save time by adding your most common tasks to it.

Starting the GUI

You can start the TeamConnection client GUI in one of the following ways:

- Select the **TeamConnection Client** icon from the TeamConnection Groupfolder on the desktop.
- Type `teamcgui` from a prompt in the directory where TeamConnection was installed.

If you are using AIX or HP-UX, type `teamcgi` from a prompt in your home directory.

The Tasks window appears.

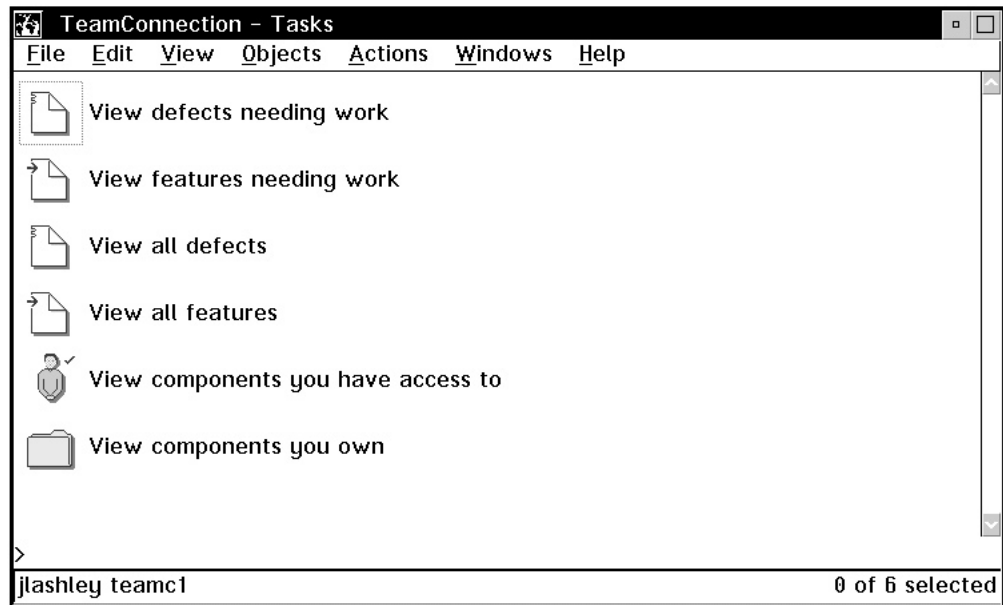


Figure 4. Tasks window

Initially, a set of default tasks appears in your Tasks window. As you become more familiar with TeamConnection and see what tasks you do most often, you can change, delete from, and add new tasks to this list. To learn how to do this, select **How do I** from the Help pull-down menu, and then select **Update tasks on the Tasks** window.

From the Tasks window, you can either select actions from the menu bar or select a task.

Stopping the GUI

To stop the TeamConnection client GUI, do one of the following:

- Select **Close** from the System menu in the Tasks window.
- Select **Exit** from the File pull-down menu of a TeamConnection window.

Performing tasks with the GUI

There are several ways you can perform TeamConnection tasks with the GUI. You can:

- Select an action from the Actions pull-down menu and then select the object you want to work with. For example, if you want to view a specific defect, select **Defects** → **View** from the Actions pull-down menu; then type the name of the defect in the View Defects window.

This method is useful when you know the exact names of the objects you want to work with.

- Select the type of object you want to work with, such as **Defects**, from the Objects pull-down menu. A Filter window appears in which you can specify search criteria. You then get a list of objects that match the search criteria. Online help provides information about using the Filter window.

This method is useful when you do not know the exact name of the object you want to work with, or you want to view a list of objects.

After you have a list of objects, and if you are going to use this list at other times, you can keep the window open. Leave the window in the background as you do your other work, or minimize it. This way, you can quickly retrieve the list when you want to perform another action.

- Select a task from the Tasks window.

This method provides a fast path within the GUI. When you select a task, TeamConnection performs the underlying query or command and then displays the requested information.

- Select an object from an object window (such as the Parts, Defects, or Features window) and then select an action to perform on the selected part from the **Selected** menu. You can also display a pop-up menu listing valid actions for a specific object by placing the mouse pointer over the object and pressing mouse button 2.

Using the Settings notebook

The TeamConnection GUI provides a Settings notebook in which you can set default values for your working environment. To open the Settings notebook, select **Settings** from the Windows pull-down menu. You can set the following values; for more information about them, refer to the online help. The notebook has five pages.

On the Environment page:	<ul style="list-style-type: none"> • Family • Release • Component • Work area • Become user • User ID • Top • Relative directory • Working directory 	The environment variables you specify on this page are relevant for the GUI only, not the command line.
On the Setup page:	<ul style="list-style-type: none"> • NLS path • Log file • Case • Print command • Compare command • Edit command 	

On the GUI page:	<ul style="list-style-type: none"> • Verbose commands • Auto refresh • Multiple object windows • Show query line • Sort pre-defined list values • Use small icons in icon views (not available in Windows clients) • Use small icons in tree views (not available in Windows clients) • Font for object windows (not available in Windows clients) • Font for output windows (not available in Windows clients) • Required field label color • Modified field label color 	
On the Extract page:	<ul style="list-style-type: none"> • Destination directory • Read-only • Expand keywords 	
On the Pool page:	<ul style="list-style-type: none"> • Pool 	

Online help information

Online help information is available from anywhere in the TeamConnection GUI. Use the online help when you need more information about a topic or task.

TeamConnection offers two types of help:

- General help

This is help for a specific window. General help provides an overview of the task and describes the objects on the window, such as menu-bar items, icons, fields, and push buttons. Do one of the following to access general help:

- Select **Help** from a menu bar.
- Select the **Help** push button.
- Press F1.

- How do I

This is where you find step-by-step instructions for doing a specific task. How you do a task depends on the component or release process that is being followed, and this help information takes that into consideration. To access this help, select **How do I** from the Help pull-down menu. Double-click on one of the task items.

At the bottom of each Help window is a **Diagram** push button. Select this push button to view a graphical process diagram. Step your way through the diagram to better understand the processes that TeamConnection components and releases can follow. The processes that your components and releases follow depend on

how the processes are configured for your organization. The defined processes determine the actions that must occur before a defect or feature can move toward completion.

Using the command line interface

To use the command line interface effectively, you must be familiar with the actions that you can perform using TeamConnection commands. A complete description of each command, including examples for each, is available in the *Commands Reference*

To view the syntax of a TeamConnection command online, type the following at a prompt:

```
teamc commandName
```

Where *commandName* is the name of the TeamConnection command.

The *Quick Commands Reference* is a booklet that lists the syntax of each TeamConnection command.

You can also become familiar with the commands by looking at the contents of the log file where TeamConnection stores the commands that are issued as you use the GUI. This file is specified in the **Log file** field on the Setup page of the Settings notebook. The default name is teamc.log; it is stored in the directory where the client is installed, unless you specify a different location in the Settings notebook.

You can type TeamConnection commands from a prompt within any directory; the TeamConnection GUI does not need to be started. Or you can type a command on the command line in the Tasks window.

Before you start to use the command line interface, you might want to set the most used environment variables, such as TC_FAMILY or TC_COMPONENT. You are not required to set these environment variables, but if you do not, you will need to specify them in the command when required.

You set environment variables differently for different platforms: AIX and HP-UX users set environment variables in the .profile (sh, ksh environment), .dtprofile (cde environment), or .cshrc (csh environment). OS/2 and Windows 3.1 users set environment variables in the config.sys file or from a command line prompt. Some environment variables are set in your config.sys file during installation.

Windows 95 and Windows NT users set environment variables in the Windows Control Panel.

You can override the value you set for an environment variable by using the corresponding flag in the command. For example, you have the TC_FAMILY environment variable set to robot, but you need to extract a file from another family named octo, so you issue the following command:

```
teamc part -extract hello.c -family octo -release 9501
```

“Appendix A. Environment Variables” on page 203 provides a complete list of the TeamConnection environment variables.

Using the web client

The TeamConnection Web Client provides family server connectivity and great deal of the functionality provided by a standard TeamConnection client without the overhead required by a standard client installation. Using a web browser, anyone in the organization can access server data (provided the server is configured appropriately) by addressing a machine and port number. Although file input/output functions are not currently available, most other familiar TeamConnection functions are available through the Web client.

To begin using the TeamConnection web client you must point your web browser to the correct URL. The syntax of the URL is: *http://host name of the server:port number of your family*. For example, if your server host name is *bldproc1* and your port number is *7890*, the URL would look like: **http://bldproc1:7890**

Your organization might require that you log in to the TeamConnection family server before you can access TeamConnection objects. If you are accessing TeamConnection through the host, you will not be able to go through a proxy unless you are:

- Logging in using a password.
- Using a smart proxy.

Using the web client is much like using the TeamConnection GUI. The following are some differences you might find:

- For the BuilderView filter the following are available in the TeamConnection GUI but not in the TeamConnection web client:
 - Source File
 - TIME-OUT
 - setupOptions
- A filter for Corequisites does not exist in the TeamConnection GUI.
- For the DefectModify action, orginLogin is available for use with the TeamConnection web client (but not the TeamConnection GUI).
- For the DriverCheck action, Dependencies is available in the TeamConnection GUI but not in the TeamConnection web client.
- For the FeatureModify action, the following are available with the TeamConnection web client (but not the TeamConnection GUI):
 - newName
 - orginLogin.
- For the DriverMemberView action, the following are available with the TeamConnection web client (but not the TeamConnection GUI):
 - state
 - defectName
 - defectAbstract
 - committedVersion.
- For the ChangeView action, the following are available with the TeamConnection web client (but not the TeamConnection GUI):
 - ChangeView
 - workAreaState.
- For the PartBuild action, the following are available with the TeamConnection web client (but not the TeamConnection GUI):

- cancel
 - partType.
- The PartChildInfoView action does not exist in the TeamConnection GUI.
- For the PartDelete action, force is available for use with the TeamConnection web client (but not the TeamConnection GUI).
- For the PartDisconnect action, parentType is available for use with the TeamConnection web client (but not the TeamConnection GUI.)
- For the PartModify action, fileType is available for use with the TeamConnection web client (but not the TeamConnection GUI).
- For the PartUnlock action, Source Directory is available in the TeamConnection GUI but not in the TeamConnection web client.
- For the PartViewContents, Expand Keywords is available in the TeamConnection GUI but not in the TeamConnection web client.
- For the UserView Filter action, the following are available with the TeamConnection web client (but not the TeamConnection GUI):
 - pswStatus
 - pswModifyTime
 - pswCreateTime.

Chapter 3. The basics of using TeamConnection

All users of TeamConnection perform a number of basic tasks, such as checking parts out of TeamConnection and then back in, and testing and verifying part changes. Before you start doing these tasks, you need to understand the basic concepts behind them; that is what this chapter explains.

This chapter assumes that you have read “Chapter 1. An introduction to TeamConnection” on page 3 and are familiar with the different objects, such as components and releases. The other chapters in this part of the book define in more detail how you perform the TeamConnection tasks.

Laying the groundwork

Someone has already created your family’s component structure, and those components manage your parts and control access to the data. Your TeamConnection family also contains releases. A release identifies a version of all the parts that comprise an application at a given point in time. When you create a release, you specify the component that will manage it. One component manages a release, but many components can manage the individual parts associated with that release.

A single part can be associated with more than one release, but it is managed by one component. When you create a part, you specify the release that you want to associate with the part and the component that you want to manage it. At any time, you can link the created part to other releases so that the part can be shared, or you can change its managing component.

Before you start working with parts, you need to be familiar with your family’s component structure. This will help you when trying to locate parts within TeamConnection and when writing defects and features. You can do the following to display your family’s component structure from the GUI:

1. Select **Components** → **Components** from the Objects pull-down menu on the Tasks window. The Component Filter window appears.
2. Type the name of the component that is at the top of your component hierarchy in the **Component** field, and select **OK**. Initially this component is called root. The Components window appears, listing the component.
3. Verify that the component is displayed in tree view (a plus sign (+) appears before the component name). If not, select **Tree** from the View pull-down menu.
4. Select **Expand fully** from the Selected pull-down menu.

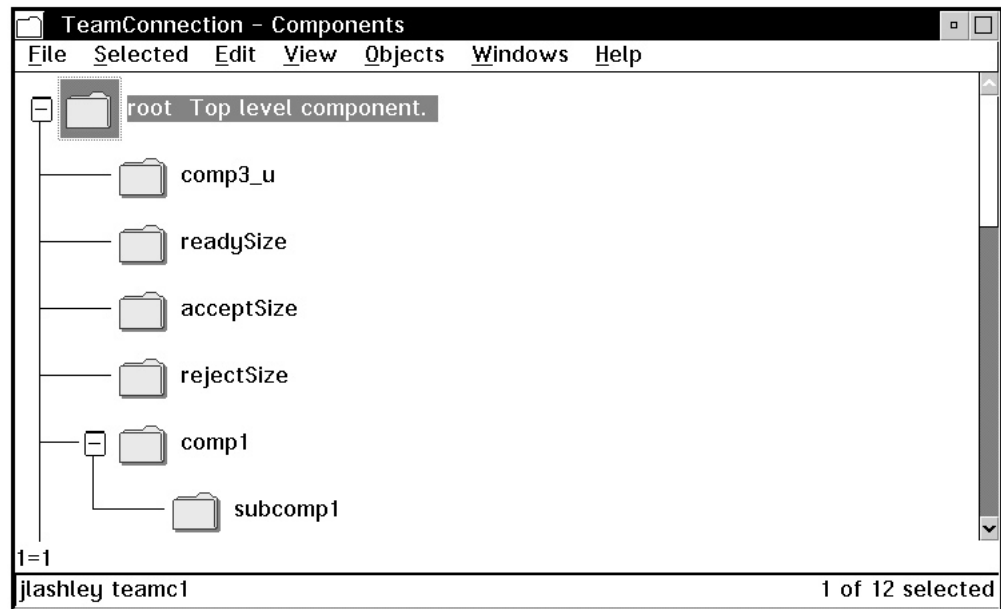


Figure 5. Components window

From a command prompt, you can issue the following command to view the component structure.

```
teamc report -view -raw bCompView -where "name='root'"
```

Authority to perform tasks

As a TeamConnection user, you are automatically given the authority to perform some basic tasks. You can:

- Open defects and features
- Add notes to existing defects and features
- Modify the information for your user ID
- Display information about any user ID
- *Search* for information within TeamConnection to create reports

You receive authority to perform additional actions when you become the owner of a TeamConnection object, such as a component or a part, or when authority is explicitly given to you by the component owners.

If you attempt an action that you do not have authority to do, TeamConnection tells you so. When this happens, you can ask the component owner, the family administrator, or a user with superuser authority to grant you the necessary authority.

Note: You can issue queries to generate reports of data from tables and views using the `— view` action flag. If you do not specify selection criteria, such as the fields and the search conditions you want to use, the report query selects all entries for the table or view indicated that the user has authority to access. This command does not show any objects in components that you are not authorized to access

“Appendix H. Authority and notification for TeamConnection actions” on page 243 lists the types of authority you need in order to perform various TeamConnection actions.

Finding objects within TeamConnection

All TeamConnection objects are stored on a server in a database. To find one or more of these objects within a family, do one of the following:

- Use the report command with the -view action flag from a command line or a command line within TeamConnection.

Command usage is explained in the *Commands Reference*

- Use a Filter window in the GUI.

Online helps explain how to use the Filter windows.

For now, you need to understand that the database is case-sensitive. You need to refer to and search for objects in the correct case. For example, if a component is stored in the database as hand, you would not find it if you typed Hand or HAND. This is why it is important that your organization sets a naming convention, and that everyone follows that decision when creating objects. If you do not know what naming convention has been established for your organization, talk to your family administrator.

Note: It is recommended that you use lowercase as much as possible.

Finding parts

There are three Filter windows that you can use to find parts within TeamConnection:

Parts Use when you want to limit your search to a particular context of a work area or driver in a release, or a particular version of a release. This is generally the view users will use most often.

If you specify only a release, TeamConnection lists the committed parts for that release. However, if you want a list of all parts in a specified work area and release, TeamConnection displays all the parts visible to the work area. This includes parts that are committed to the release as well as changed parts that are visible only to the work area.

BuildView

Use when you want to search for information related to building your application, such as viewing a build tree, or when you want to do build actions.

PartFull

Use when you want to search for parts across releases, components, or work areas. For example, you want a list of all the optics.c parts. Unlike the Parts Filter, you can specify one or more release or work area names.

You can also use this filter to display only parts that have been changed in a work area. For example, you check out robot.c to work area 310:1, and that is the only part that you have changed. If you use the PartFull Filter to query for all the parts in work area 310:1, only one record is returned.

You cannot use this filter to search for build information.

Refer to the online help, in particular **How do I**, for more information on how to use the Filter windows. Select **How do I** from the Help pull-down menu to access the information.

Using work areas

A work area is a logical temporary work space that enables you to isolate your work on the parts in a release from the official versions of the parts. You can check parts out to a work area, update them, and build them without affecting the official version of the parts in the release. You must create a work area before you can create, check out, or check in parts. If your component's process includes a design, size, review subprocess for defects or features and the release follows a tracking subprocess, a work area is automatically created when sizing records exist and the associated defect or feature is accepted. TeamConnection associates these work areas with the appropriate defect or feature.

The parts in a work area do not become available in the release until the work area is integrated. Also, if your release follows a driver subprocess, parts that have been changed do not become available in the release until the associated driver is committed. However, users who have the authority to access the work area can view and work with the parts in it.

You can save intermediate versions of the parts in your work area by *freezing* your work area. Every time you freeze a work area, TeamConnection saves a revision level of the work area. When you freeze work area 123:1, for example, a version called 123:2 is created. This version contains information about each part in the work area and its current version at the time the work area was frozen. It may contain version 1 of part optics.c, for example. If you freeze the work area again later, a new version called 123:3 is created with information about the versions of the parts in the work area when it was frozen. This version may contain version 2 of part optics.c. Each of these work area versions is saved in the database and you can retrieve the versions of the parts they contain before you integrate the work area into the release. Therefore, you should freeze a work area whenever there is a possibility that you will want to return to that version of the work area. For example, you might be adding a major feature to the code, and you want to be able to return to something that works in case the application no longer builds. When you integrate a work area or commit a driver, the work area is frozen automatically.

Naming your work areas

When TeamConnection automatically creates a work area, the work area is given the same name as the defect or feature it was created for plus the initial version number, :1. When you create a work area, you can also give it the same name as the defect or feature, or you could give it any other name. Where possible, we recommend that you name it after a defect or feature, or relate the name to the change that is being made.

Here are some things you should know before you name a work area:

- Work area names must be unique within the context of a release.
- After you create a work area, you cannot delete it. You can, however, cancel the work area in the following situations:
 - No part changes were made.
 - You undo the changes you made.

- With the proper authority, other users in your organization will be able to access your work area and make changes to the parts. This means that you need to make it easy for them to locate the work area. Following your local naming conventions will help.
- After the work area is integrated with the release, you cannot reuse the work area. If the defect is still in the working state, you can create another work area with a different name after the initial work area is integrated with the release.

Creating parts

A TeamConnection part is controlled by the TeamConnection server. A TeamConnection part is uniquely identified by the *path name* of the part, the part *type*, and the name of the release in which it is contained. You must specify both the release name and the path name whenever you perform a TeamConnection action on a part. Multiple releases can share the same part.

When you create a part, you do one of the following:

- Take an existing text or binary file that is on your workstation and place it into TeamConnection.
- Create an empty part that has no content. Empty parts are used as place holders until an application is built. For example, you can create a place holder for an executable part that will be created by a build. See “Creating the build tree for the application” on page 143 for an example of creating a place holder.

After you put a part under TeamConnection control, the official copy of the part resides in the database. The copy on your workstation is changed to read-only mode. You can then change the part by checking it out to your workstation or editing it within the GUI.

Use the online help facility if you need assistance when creating parts.

Naming your parts

If your organization has a naming convention, be sure to follow it when naming your parts. When the naming convention is not followed, everybody in your organization can have trouble locating parts. Part names created on the server are case-sensitive; they must be retrieved using the same case in which they were created.

When you name TeamConnection parts, you can specify only the base name, such as `hand.c`, or you can specify the directory path in addition to the base name, such as `robot\hand\hand.c`. Specifying the path name as part of the name lets you have several identical base name parts included in the same release—for example, `robot\hand\msg.h` and `robot\optics\msg.h`.

You can also have identical part names within the context of a release as long as their part types are different, such as `TCPart` and `vgdata`.

Note: It is recommended that you use lowercase letters to name your parts.

Preparing to build your parts

If you are going to use the TeamConnection build function, you must provide certain information about each part that participates in a build. You can provide this information when you create the parts or wait until later. You can also change the information at any time.

To associate a part with a build, you must specify the following information:

- The parent part that you want to associate the part with.
- The type of relationship the part has to the parent, such as:

Input The part will be used as input to building its parent. An example of an input part is a C language source file, `x.c`, which is compiled to create its parent, `x.obj`.

Output The part will be a generated output from the same build that creates its parent part. In other words, both the parent part and this child part are outputs when the parent part is built.

Dependent The part will be needed for the build operation of its parent to complete, but it will not be passed directly to the builder. An example of this is an include file.

If you do not provide this information when you create the part, you can provide it later using the `connect` function.

You can also specify the builder or parser that a part will use, as well as any build parameters.

“Part 3. Using TeamConnection to build applications” on page 91 explains the build function in more detail.

Working with parts

After the parts are created in TeamConnection, you will be working with these parts—getting them to your workstation so you can change them and then getting them back in to TeamConnection. This section gives a brief overview of these tasks. “Chapter 5. Working with no component or release processes” on page 47 and “Chapter 6. Working with component and release processes” on page 69 go into more detail about these and other TeamConnection tasks.

Working in serial or concurrent development mode

A release is set up for either serial development or concurrent development mode. Once the development mode is set you can change from serial mode to concurrent mode, but not from concurrent mode to serial mode. In serial development, a part is locked when a user checks it out, and no one else can update the part as long as it is checked out. In concurrent development, more than one user can simultaneously have the same part checked out.

When two users have the same part checked out in concurrent development, both can change it. The first user integrates the work area, which contains the changed

part, with the release. When the second user does the same, TeamConnection recognizes that the parts differ and notifies the user. It is up to this user to resolve the differences, using the TeamConnection merge program or some other merge program.

Before getting parts from TeamConnection, you might want to find out if the development mode for the release is concurrent or serial. To determine the mode, view the information about the specific release. To do this, select **View** from the Selected pull-down menu on the Releases window.

Working with common parts

A *common* part is a part with identical content that is shared by two or more releases or two or more work areas. For example, when an identical part is needed in two separate releases, you can link the part from one release to the other (if you have the proper authority). Both releases would then have a link to the current version of that part.

When a common part is checked out of a release, TeamConnection locks the current version of the part in all releases if one of them uses serial development. When putting the part back into the release, one of the following actions reflects the change in all releases in which the part is common:

- You integrate the work area when the driver subprocess is not followed, or
- You commit the driver when the driver subprocess is followed.

You can break the common link if you make changes to a common part and you do not want these changes reflected in other releases or work areas that link to the part. You can break the common link when you check out, check in, rename, delete, re-create, connect, or *disconnect* parts. When a part is common to more than two releases, you can maintain the common link with some of the releases while breaking the link with other releases. When a link is broken, the parts still share the same name, but the information contained in the parts is different.

Parts can also be linked between two or more work areas in the same or different releases, making the parts common to those work areas. For example, a user working in one work area can link to the latest version of a part in another work area of the same release (the part has yet to be integrated with the release). The part is then common to the two work areas within the same release. If you want to maintain the common link to all work areas, you must specify the names of the common work areas when you check in, rename, delete, or re-create the parts. As with common parts in releases, you can break the common link.

You can also link all the parts within a release to another release. This function is especially helpful when development begins on a new release of a product, and you want the parts in the new release to initially be the same as the parts in the current release. As development of the two releases continues, the common link between the parts can be broken to separate development of the new release from maintenance of the current release.

For more information about how to link parts, refer to the *Commands Reference* and online help.

Getting parts from TeamConnection

Checking out a part implies that you intend to modify it; extracting a part merely gives you a copy of the part. Normally, when you extract a part, you do not plan to change the current version in TeamConnection.

You must have the necessary authority to a component before you can check out or extract parts from that component. You need *PartExtract* authority to extract a part from TeamConnection; you need *PartCheckOut* authority to check a part out. See “Appendix H. Authority and notification for TeamConnection actions” on page 243 for a listing of all the TeamConnection actions and their authority requirements.

Parts are checked out to work areas. The work area is where you store updated parts and do builds without affecting the version of the parts in the release. When a part is checked out of the release to the specified work area, TeamConnection locks the part in the release if you are working in serial development. If you are working in concurrent development, the part is never locked. TeamConnection also puts a copy of the part on your workstation. It is here where you update the part. If a read-only copy of the part exists on your workstation, the first character of its file extension changes as follows: It becomes \$ for OS/2 and Windows platforms. It becomes a . for AIX and HP-UX platforms. This copy is a backup copy. If a backup copy already exists, it is deleted. When you are finished updating the part, you check it back in to the work area. A work area is optional when extracting a part.

When you extract a part, TeamConnection copies the part to your workstation, and the part is not locked. In other words, other users can still check out the same version of the part and make changes to it, even in serial development mode. By default, TeamConnection sets the extracted part to read-only access. This is done to keep you from inadvertently changing the part on your workstation when the part in TeamConnection is not locked. You can, however, change this in the Settings window or when you are extracting the part. When you do this, be aware that someone else can change the official part in TeamConnection, making your workstation copy back level.

Where TeamConnection places a checked-out or extracted part on your workstation depends on the following:

- Your workstation’s *current working directory*
- Whether you use the -relative flag on the command line or the **Destination directory** field on the GUI
- Whether the TC_TOP environment variable is set

For more information about how these interact, refer to the part command examples in the *Commands Reference*

When you want to make changes to a part, you can do one of the following:

- Check out one or more parts and edit the parts on your workstation. When you finish making changes to the parts, you check them back in.
- Edit a part from within the TeamConnection GUI using a specified editor. When you exit the editor, the Check In Parts window appears and you can check the part back in to TeamConnection.

In either case, if you are working in concurrent development and someone else changed a part while you had it checked out, you are asked to resolve the differences when you try to integrate your work area.

Checking parts in to TeamConnection

After you have verified the accuracy of your part changes, you are ready to check them in to TeamConnection. Any parts that you have checked out, you have the authority to check back in.

As mentioned earlier, you check parts out to a work area so you can work on them. Therefore, when you check in a part, you must specify the work area where that part is checked out. In other words, you check the part back in to the same work area. When the part is checked in, the copy on your workstation is flagged read-only.

At this time, the changed part is visible in only the named work area; it is not visible at the release or to any other work area. This lets you test your changes by building the version of the code that is in your work area.

When you are satisfied with your changes, you can integrate the parts into the release by integrating your work area. This action makes the work area visible to all the users in the release.

If you are working in concurrent development mode, TeamConnection generates a *collision record* when a changed part conflicts with a previously committed part. For example, both you and Keith have `hand.c` checked out. Keith makes changes to the part and then integrates the work area that contains that part. (Depending on the process being followed, Keith might have to commit the work area rather than integrate it.) Later, after making changes to `hand.c`, you attempt to integrate the work area that contains the part. Because the part was already integrated by Keith, you are notified of a collision and asked to refresh your work area. After the refresh, you can view the collision record and decide how you want to resolve the conflicts. “Reconciling differences” on page 65 explains in more detail how this works.

Finding different versions of TeamConnection objects

TeamConnection version control maintains different versions of the following objects:

- Releases
- Work areas (and driver members)
- Drivers
- Parts

When you want to find and retrieve previous versions of these objects, it is helpful to know how TeamConnection creates and deletes previous versions of each object.

Some basics of TeamConnection versioning will help you understand how TeamConnection identifies unique versions of objects:

- When you first create an object, the initial version name is the object name suffixed with `:1`. When you create a new work area called *myWorkArea*, for example, its version is *myWorkArea:1*. Subsequent versions are identified in numerical order: *myWorkArea:2*, *myWorkArea:3*, *myWorkArea:4*, and so on. Versions of releases and drivers are identified similarly: *myRelease:1*, *myRelease:2*, *myRelease:3*; *myDriver:1*, *myDriver:2*, *myDriver:3*; and so on.
- Unique versions of parts are identified by association with a specific version of a release, work area, or driver. Your TeamConnection family may have three

different versions of a part called *myPart*, for example: one associated with release *myRelease:2*, one associated with work area *myWorkArea:1*, and one associated with work area *myWorkArea:2*.

Versioning releases

TeamConnection creates new versions of releases whenever you do the following:

- Create a release

This is the initial version of a release and contains no parts. When you create *myRelease*, for example, its version name is *myRelease:1* and it contains no parts.

- Commit a work area to the release

Committing a work area to a new release creates a new version of the release and adds the parts in the work area to the release. When you commit work area *myWorkArea:1*, for example, to *myRelease:1*, TeamConnection creates a version of *myRelease* called *myRelease:2*. It also associates the parts in *myWorkArea:1* with *myRelease:2*.

- Commit a driver to a release

Because drivers are simply collections of work areas, committing a driver to a release has the same effect as committing a work area: TeamConnection creates a new version of the release. When you commit *myDriver:2* to *myRelease:2*, for example, TeamConnection creates a version of *myRelease* called *myRelease:3*.

TeamConnection deletes versions of releases whenever you prune the release. Refer to the *Administrator's Guide* for an explanation of pruning.

Versioning work areas

TeamConnection creates new versions of work areas whenever you do the following:

- Create a work area

This is the initial version of a work area. When you create *myWorkArea*, for example, its version name is *myWorkArea:1*.

- Refresh a work area

Refreshing a work area updates it with any new versions of parts that have been integrated with the release. When a workarea is refreshed, two versions of the workarea are created. One of the contents before the refresh and one with the contents after the refresh.

- Freeze a work area

Freezing a work area is like taking a snapshot of the work area. It preserves the parts as they are at a given point in time. If you create work area *myWorkArea:1*, add three new parts to it—called *part1*, *part2*, and *part3*—and then freeze it, your family contains a work area called *myWorkArea:2*, with *part1*, *part2*, and *part3*. The version name of each of these parts is *myWorkArea:1*. If you then alter *part2* and freeze the work area again, your family contains the following:

- *myWorkArea:1*, with nothing in it
- *myWorkArea:2* contains *part1*, *part2*, and *part3* at version *myWorkArea:1*
- *myWorkArea:3* contains *part1* and *part3* at version *myWorkArea:1*, and *part2* at version *myWorkArea:2*

- Commit a work area

Committing a work area adds the parts in the latest version of the work area to the release. It also does the following:

- Creates a new version of the release
- Creates new versions of the parts in the release
- Deletes any intermediate versions of the work area

Using the previous example, if you commit myWorkArea:3 to myRelease:1, the following happens:

- TeamConnection creates a new version of myRelease called myRelease:2.
- TeamConnection creates new versions of the parts in myRelease:2.
- TeamConnection deletes myWorkArea:1, myWorkArea:2, and myWorkArea:3.

TeamConnection deletes versions of work areas whenever you commit them to a release. Once a work area has been committed, you can no longer use it for making part changes and you cannot create a new work area with the same name.

Deleting work area versions is controlled by the autopruning option of the release associated with the work area. By default, TeamConnection always deletes work area versions on commit, but you can change this option. Refer to the *Administrator's Guide* for an explanation of autopruning.

Versioning drivers

TeamConnection creates new versions of drivers whenever you do the following:

- Create a driver

When you create a new driver, TeamConnection makes two versions of it: myDriver:1, for example and myDriver:2.

- Add a work area (driver member) to a driver

If you add myWorkArea:1 to myDriver:2, for example, TeamConnection creates a new version of myDriver called myDriver:3.

- Freeze a driver

Freezing a driver is like taking a snapshot of the driver. It preserves the parts as they are when the driver is frozen. If you freeze myDriver:3, for example, TeamConnection creates a new version called myDriver:4.

- Refresh a driver

Refreshing a driver updates the driver with all changes that have been made in all of its driver members. Refreshing a driver actually creates two new versions of the driver, as follows:

1. Freezes the driver (so that TeamConnection can have a point to roll back to if an error occurs during the refresh operation).
2. Updates the driver with any changes from the driver members
3. Freezes the driver again, thus preserving a copy of the updated driver.

If the current version of myDriver is myDriver:2, for example, and the parts in its driver members have been changed, then TeamConnection does the following when it refreshes the driver:

1. Freezes myDriver, creating myDriver:3.
2. Updates myDriver with changes from its driver members.
3. Freezes myDriver again, creating myDriver:4.

The result of refreshing myDriver (version myDriver:2) is two new versions: myDriver:3, containing a snapshot of the driver before the refresh, and myDriver:4, containing a snapshot of the driver after the refresh.

TeamConnection deletes versions of drivers whenever you remove driver members or commit a driver to a release.

- If you have a driver version myDriver:4 with driver members myWorkArea, yourWorkArea, and ourWorkArea, and you remove myWorkArea, then TeamConnection deletes driver versions myDriver:2, myDriver:3, and myDriver:4 and creates a new driver version called myDriver:5 containing members yourWorkArea and ourWorkArea. As a result, the family contains two versions of the driver, myDriver:1 and myDriver:5.
- When you commit a driver to a release, all intermediate versions of the driver (resulting from driver member add, driver freeze, driver refresh, or driver member remove operations) are deleted.

Versioning parts

TeamConnection versions parts in association with other TeamConnection objects, such as work areas. If, for example, you create part1 in myWorkArea:1, the current version of part1 is myWorkArea:1. If part1 is in release myRelease:2 and work area myWorkArea:2, then you can view the version of the part for either the release or the work area. The version label for part1 in myRelease:2 is **myRelease:2** and in myWorkArea:2 is **myWorkArea:2**.

TeamConnection deletes part versions whenever it deletes versions of the object that the part is associated with. In addition to versioning in association with other TeamConnection objects, TeamConnection maintains versions of build output parts (parts that are created as the result of a build, such as an .exe file or a .hlp file). When you create a release, you can set the maximum number of versions of build output parts to maintain. If you set this maximum to 10, for example, then TeamConnection saves only 10 versions of build output parts and discards the oldest version each time a new version is created.

Working with defects and features

Defects are used to report problem information; features are used to record information about proposed design changes. After a defect or feature is opened, TeamConnection tracks the progress of the defect or feature through various states. To what degree defects and features are tracked depends on the processes followed by the release and component to which they are assigned. The following describes actions that your defined processes might require:

Analyzing defects and features

The owner is responsible for analyzing a defect or feature after it is opened. The owner can then return it if it is not valid or feasible, reassign it to another user or component, or accept it for resolution.

Designing the resolution

After a defect or feature has been accepted, the actual resolution needs to be designed so that an informed evaluation can be made. This resolution needs to be designed by users who are familiar with the product or area affected by the defect or feature.

Identifying the required resources

Sizing records are created by the owner to identify the components and releases that might be affected by the defect or feature. Each owner of a component that is referenced in a sizing record needs to evaluate the impact of the defect or feature on the parts managed by the component. If

the defect or feature requires changes to parts, the sizing record is accepted and sizing information is added.

When sizing records exist and the associated defect or feature is accepted, TeamConnection automatically creates a work area.

Reviewing the design and resource estimates

After the resolution has been designed and the resources have been identified, the proposal needs to be reviewed. If the review indicates that work should continue on the defect or feature, it is accepted.

Resolving defects and implementing features

Resolving one defect or implementing one feature in one release can involve one or more users changing many parts. To change a part, a user must check out the part, make the changes required to resolve the problem or implement the design change, and check the part back in. If the release follows a tracking process, all defects or features must be associated with a work area. Parts that are checked out refer to the work areas that are monitoring the defect or feature.

Resolving a defect or implementing a feature also involves integrating the changed parts with changes made for other defects and features in that release. All changed parts are eventually integrated with the unchanged parts within the release.

Verifying the resolution of the defect or feature

The originator uses a verification record to acknowledge that the defect or feature was satisfactorily resolved or not. Accepting a verification record formally closes the defect or feature. Rejecting a verification returns the defect or feature to working state.

“Chapter 4. The states of TeamConnection objects” on page 37 explains in more detail the various states that different TeamConnection objects can go through depending on the process that is being followed. A diagram in this chapter shows the flow of these states. You might want to study this information before you start to work with defects and features.

Testing and verifying part changes

You can use TeamConnection’s build function to build your program. Before you check in updated parts, you will probably want to verify the accuracy of your changes.

The scenarios in “Chapter 5. Working with no component or release processes” on page 47 and “Chapter 6. Working with component and release processes” on page 69 include information about testing and verifying part changes. “Part 3. Using TeamConnection to build applications” on page 91 provides detailed information about the build function.

Chapter 4. The states of TeamConnection objects

The actions that you can perform on certain TeamConnection objects are controlled by two factors:

- The process followed by the component and by the release
- The current state of the object

Certain TeamConnection objects can follow certain states through their life cycle. An instance of an object might not go through all the possible states for that object—it moves through the states that are defined in the process followed by the component and by the release. The following table briefly lists the component and release subprocesses. For more information on component and release subprocesses, refer to the *Administrator's Guide*

Component subprocesses

dsrDefect

Design, size, and review fixes to be made for defects

verifyDefect

Verify that defect fixes work

dsrFeature

Design, size, and review changes to be made for features

verifyFeature

Verify that feature changes work

Release subprocesses

track Relate all part changes to a specific defect or feature and a specific release

approval

Require all changes to be approved before incorporating them into a release

fix Use fix records to ensure that all required changes are made

driver Use drivers to integrate changes into a release

test Require all changes to be tested before they are integrated into the release

This chapter explains the possible states of certain TeamConnection objects and how objects are moved from one state to the next. It also explains how component and release subprocesses affect the flow of states. For a diagram showing the flow of states, refer to the poster *Staying on Track with TeamConnection Processes*.

Defects and features

Use defects to report problem information; use features to record information about proposed design changes. After you open defect or feature, TeamConnection tracks the progress of the defect or feature through various states. Defects and features are tracked according to the processes followed by the release and component that they are assigned to. The possible states for defects and features are:

Open state

When you open a defect or feature, it is in the open state and you are considered the originator.

You assign the defect or feature to a component. The owner of this component becomes the feature or defect owner and is responsible for managing its resolution. The component you open a defect or feature against should be one that manages the parts affected by the enhancement or problem. Use the component descriptions and the structure of your family's hierarchy to find the most appropriate component. If you open a defect or feature in an inappropriate component, the component owner can reassign it.

While the defect or feature owner is responsible for implementation, the originator is responsible for verifying that the defect or feature is resolved correctly.

Returned state

A defect or feature owner can return a defect or feature to its originator. You can return a feature or defect from the open, design, size, or review state if you decide that the defect or feature is not feasible or not valid. You can return a defect or feature back to the working state only if it has no associated work areas. If there are associated work areas, you must cancel or undo them before you can return the defect or feature. When you return a defect or feature, add your reason for returning it so that the originator and any other users can evaluate why you believe it is not feasible or not valid.

Canceled state

A feature or defect in the open or returned state can be canceled only by its originator or by a superuser. A canceled defect or feature remains inactive unless it is reopened by the originator.

Design state

If the component to which a defect or feature is assigned includes the `dsrDefect` or `dsrFeature` subprocess, you move defects or features in the open or returned state to the design state.

In this state, the proposed change is designed, and a description of the design change is entered. The owner must describe the design change before the defect or feature can move to the next state.

If the release includes the fix subprocess, fix records are automatically created when a defect or feature is designed.

Size state

Defects or features move to this state after the owner enters design information.

In this state, users can create a *sizing record* for each release that contains parts affected by the enhancement or problem. A sizing record identifies the work that is required for and the resources affected by the defect or feature. The owner of the component that is referenced in the sizing record is the owner of the sizing record. The owner is responsible for entering information about the amount of work that is required to implement the feature or resolve the problem.

The sizing record owner can reject the sizing record if it does not affect the specified component. After all sizing records are either accepted or rejected, the defect or feature moves to the review state or returns to the design state if more design information is needed.

Review state

Defects or features move to this state after they have been sized. In this

state, the design text and sizing records are reviewed to determine the feasibility of the proposal. The owner can do one of the following:

- Accept the defect or feature if all design and sizing records are acceptable. This moves the defect or feature to the working state.
- Return the defect or feature to the originator if all design and sizing records are not acceptable. If necessary, the originator can reopen a defect or feature.
- Move the defect or feature back to the design state if design modifications are needed.

Working state

Defects or features move to this state when the owner accepts the defect or feature when it is in the:

- Review state, if the component includes the `dsrDefect` or `dsrFeature` subprocess
- Open state, if the component does not include the `dsrDefect` or `dsrFeature` subprocess

When you accept a defect or feature, you accept the responsibility of resolving it. A defect or feature might require changes in more than one release.

What happens after a defect or feature is accepted varies according to the subprocesses in effect:

Component subprocesses

dsrDefect or dsrFeature

TeamConnection creates a work area in the approve state for each release identified in the accepted sizing records for the defect or feature.

verifyDefect or verifyFeature

TeamConnection creates verification records in the notReady state.

Release subprocesses

fix TeamConnection creates fix records in the notReady state based on the sizing records.

approval

TeamConnection creates approval records for each user on the release's approver list.

If the component does not include the `dsrDefect` or `dsrFeature` subprocess, then you must manually create a work area before you can check out or create parts to address the defect or feature.

Verify state

Defects and features go through the verify state only if their component includes the `verifyDefect` or `verifyFeature` subprocess. Defects and features are automatically moved to this state when one of the following happens:

- All work areas (there can be multiple work areas for the defect or feature) for the release are integrated, if a release is specified when the defect or feature is created

When a defect or feature is accepted, TeamConnection creates a *verification record*. This record lets the originator:

- Accept the fix if the resolution was satisfactory
- Reject the fix if not satisfied with the resolution
- Abstain if unable to assess the resolution

Once all verification records have been accepted or abstained, the defect or feature moves to the closed state. If a verification record is rejected, the defect or feature returns to the working state. The defect or feature cannot be closed until the verification records are accepted.

A defect or feature can have more than one verification record. For example, if defect 123 is returned because it is a duplicate of defect 122, a second verification record is created for defect 122. The originator of defect 123 is the owner of the second verification record for defect 122. If the originator is the same for both defects, only one verification record is created.

Note: For a discussion of verification records and test records, see “Verification and test records” on page 44.

Closed state

The closed state is the final state of a defect or feature.

If the defect is associated with multiple work areas, the defect will remain in the working state until all of the work areas are integrated.

If the component includes the `verifyDefect` or `verifyFeature` subprocess, the defect or feature automatically moves to the closed state after all verification records are in the accept or abstain state and all work areas are in the complete state. If a verification record is rejected, the defect or feature moves back to the working state. Otherwise, the defect or feature moves directly from the working state to the closed state when the first work area moves to the complete state.

You cannot re-open a defect or feature that is in the closed state. If the defect or feature was not resolved correctly, you must open a new defect or feature to address the necessary changes.

The states of work areas

A work area is a storage area where you can work on the parts in a release without affecting the “official” versions of those parts. A work area can be associated with a specific defect or feature, but it does not have to be.

Approve state

When a work area is created, it goes to this state if the release includes the approval subprocess. TeamConnection creates an approval record for each user on the release’s *approver list*. Each approver indicates their evaluation of the changes in their approval record:

- Accept that work should continue
- Abstain if unable to assess if work should continue
- Reject if work should not continue

When all approval records are marked as abstain or accept, the work area goes automatically to the fix state. If any approval record is marked as reject, the state of the work area remains at approve. You can change rejected approval records to accept or abstain.

Fix state

If the release does not include the approval subprocess, work areas for the release begin in the fix state.

While the work area is in this state, parts are checked out to the work area, changes are made to these parts, and builds are done to verify the accuracy of the changes.

If the release includes the fix subprocess, any fix records created for a defect or feature move to the active state when a part change is associated with the work area for the defect or feature. A fix record monitors the part changes within a single component. Fix records provide a mechanism for reviewing all part changes that apply to components before integrating those changes with changes made for other defects and features.

How fix records are handled varies according to the subprocesses in effect:

Component subprocesses

dsrDefect or dsrFeature

TeamConnection creates fix records for features or defects when existing sizing records are accepted.

Release subprocesses

fix If a fix record does not already exist for the component, TeamConnection creates one when a part managed by that component is checked in to the database.

If neither of these subprocesses are in place and a defect or feature owner needs to create a work area manually, he or she can create fix records at the same time. Existing fix records go to the active state when a part is checked in to the work area.

Fix records provide a way of ensuring that all necessary part changes within the specified component have been made and are reviewed or inspected. The fix record owner is responsible for this review. When the fix record owner is satisfied that the part changes made within that component are complete and ready for integration with other parts in the release, the owner marks the fix record as complete. When all existing fix records for a work area are complete, the work area automatically moves to the integrate state.

Integrate state

Work areas can be moved to the integrate state as follows. If the release includes the fix and driver subprocesses, the work area automatically moves to the integrate state when all fix records are complete. If all fix records are not complete, you can force a work area to the integrate state, provided that no part changes are associated with the work area. If the release does not include the fix and driver subprocesses, you must move the work area to the integrate state manually.

While a work area is in integrate state, you must add it to an existing driver as a driver member if the release includes the driver subprocess. All work areas in the integrate state do not have to be added to the same driver. The work area stays in the integrate state until the driver in which it is a member is committed.

You can move work areas from the integrate to the following states, according the subprocesses in effect:

Release subprocesses

driver A work area moves to the commit state when the driver it is a member of is committed, or to the restrict state when the driver is restricted. You can also force a work area to the commit state, provided that no part changes are associated with the work area.

test A work area moves to the test state so that test records can be approved or rejected.

If the release does not include these subprocesses, you can manually complete a work area in the integrate state.

Restrict state

Work areas can be moved to the restrict state only when the release includes the driver subprocess. The work area moves automatically to the restrict state when the driver to which it belongs is restricted. If a work area in this state is removed from the driver, it returns to the integrate state. Otherwise, the work area remains in the restrict state until the driver to which it belongs is committed.

Commit state

Work areas can be moved to the commit state only when the release includes the driver subprocess. The work area moves automatically to the commit state when the driver to which it belongs is committed. At this point, all parts that were changed in this release to resolve the feature or defect are committed. The work area remains in the commit state until the driver to which it belongs is completed.

Test state

Work areas can be moved to the test state only when the release includes the test subprocess. When the associated driver moves to the complete state or when a work area is committed without a driver, the work area moves to the test state. The driver is then ready for formal testing in the specified environments. Test records for the work area are created in the ready state when the work area moves to the test state. The work area stays in the test state until all test records are accepted, rejected, or abstained.

Complete state

The complete state is the final state of a work area; the work area can no longer be used. If the test subprocess is not included in the release process, the work area moves directly to this state when the associated driver is completed or when the work area is explicitly integrated.

When a work area is completed, the feature or defect associated with that work area automatically moves to the verify or complete state. The defect does not leave the working state until the work area for that release is completed.

The states of drivers

Drivers monitor and implement the integration of part changes within a release. Those part changes are included in a driver by adding the work areas containing the changed parts to the driver as driver members.

Working state

The working state is the initial state of a driver. While the driver is in this state, it is not associated with any work areas and, therefore, contains no part changes.

If the release includes the driver subprocess, drivers can be explicitly created at any time.

Integrate state

Each driver automatically moves to the integrate state when the first work area is added to it as a driver member. If all work areas are removed from the driver, the driver automatically returns to the working state.

Work areas can be added to drivers as driver members when the driver is in the working, integrate, or restrict state and the work area is in the fix state. Adding driver members to a driver in restrict state requires proper authority.

You can extract the driver when it is in the integrate state; however, only those parts that were changed in reference to driver members are extracted. This is referred to as extracting a *delta part tree*.

Restrict state

Before a driver is committed, you can move it to the restrict state. While a driver is in this state, work areas in the integrate state can be created for or deleted from the driver by only a superuser or an individual with the special authority of memberCreateR or memberDeleteR. This allows a build administrator to have better control over what is being built. The build administrator can delete driver members that are causing build errors or add driver members to fix build errors. You can then commit an error-free driver.

When a driver moves to the restrict state, all work areas that are included as driver members also move to the restrict state.

Commit state

Committing a driver commits all work areas included as driver members and all parts that were changed in reference to those work areas.

TeamConnection commits only a successfully built driver. Committing a driver changes it to the commit state. You can, however, manually commit the driver. You can also commit an unsuccessful driver by using the force option.

When a driver moves to the commit state, all work areas that are included as driver members also move to the commit state. When a work area is in the commit state, all part changes associated with the work area become the "official" versions of the parts in the release and are visible to all users of the release.

A committed driver can be extracted as a full part tree as well as a delta part tree. A full part tree is the part structure of all the parts within the release.

Complete state

The complete state is the final state of a driver. In this state, the driver is ready for formal testing in the specified environments.

If the release includes the test subprocess, the work areas that are included as driver members move to the test state. Any existing test records for the

work area move to the ready state when the work area moves to the test state. The work area stays in the test state until all test records are accepted, rejected, or abstained.

Test records are used to record the outcome of environment tests for changes implemented in a driver. This record lets the owner:

- Accept the record if the test was satisfactory
- Reject the record if not satisfied with the test results
- Abstain if unable to assess the results

Once all test records have been accepted or abstained, the states of other objects change as follows:

Work areas

Go to complete state

Defects and features

Go to verify state if the component includes the verifyDefect or verifyFeature subprocess; otherwise they go to the closed state.

Verification records

Go to ready state and are sent to the defect or feature originators

If the test subprocess is not configured, then work areas move to the complete state and any defects or features move to the verify state.

If the component includes a verifyFeature or verifyDefect subprocess, verification records move to the ready state and notification is sent to the originators of any defects or features that were addressed in the completed driver.

The commit and complete states of drivers differ as follows:

- When a driver is committed, all work areas are committed, but no changes occur in the states of defects or features associated with the work areas.
- When a driver is completed, then the states of other associated objects (such as test records, work areas, verification records, defects, and features) change according to the other subprocesses in effect:

test Work areas go to the test state and test records are created in the ready state for each environment in the release's environment list.

verify Verification records go to the ready state.

If the release includes neither of these subprocesses, then the work area goes to the complete state and all features and defects associated with the work area are closed.

Verification and test records

If you use both the verify component subprocess (verifyDefect or verifyFeature) and the test release subprocess, then TeamConnection creates both verification records for features or defects and test records for each environment defined in the release's environment list. These records serve different purposes:

- Verification records provide a means of accepting or rejecting the product changes made in response to defects or features and are thus specific in nature.

- Test records provide a means of accepting or rejecting the results of a build and are more global in nature.

These records are handled by different people and enable you to monitor your development progress in different ways. The sequence of creating and handling verification and test records is as follows:

1. Verification records are created in the notReady state when a defect or feature is accepted. This indicates that someone on the development team has begun implementing the changes warranted by the defect or feature, but the changes are not yet ready to be verified. A work area is also created for the part changes.
2. When a driver is committed all part changes associated with the driver members are integrated into the release.
3. To create test records, the driver is completed. This action creates one test record for each environment on the release's environment list. The testers on your development team use the test records to accept or reject the results of their tests on the part changes.
4. After all test records have been accepted or abstained, the verification records are moved to the ready state. This indicates that the part changes have been tested in the context of the build and each individual defect or feature is ready to be accepted or rejected by the person who opened it.
5. The defect or feature originator accepts or abstains the verification record to close the defect or feature. The originator can also reject the verification record to move the defect or feature back to working state.

Chapter 5. Working with no component or release processes

To illustrate how to work with objects in a release that does not follow a tracking process, this chapter follows an example of a programming team that is developing the control systems for a robot. They are working in a family called robot.

Instructions for performing the task are given for both the graphical user interface (GUI) and the command line interface (Command).

This chapter illustrates two scenarios: working in serial development and working in concurrent development. Working in *serial development* means that after you check out a part, TeamConnection locks the part so that it cannot be updated by anyone else. Compare this to *concurrent development*, in which more than one person can simultaneously update the same part.

The following table directs you to the scenario you need:

For this scenario,	Go to this page.
Working in serial development	47
Working in concurrent development	63

Working in serial development

Alex is one of the programmers working on the robot application within a release called robot_control. The release does not follow a tracking process, and the release supports serial development. Even though the release does not follow a tracking process, defects are opened when problems are found.

This example assumes that the parts that Alex will work with have already been created in the release, and the build tree has been established. The build tree shows the hierarchy of objects that take part in the build of an application. It identifies parts as inputs, outputs, and dependencies of a build. For more information about build trees, see “Working with a build tree” on page 98 or “Creating the build tree for the application” on page 143.

This example also assumes that the family named robot has been defined in the TC_FAMILY environment variable. Because Alex accesses information in several releases, he has not defined the release named robot_control. Therefore, he must explicitly identify the release when performing TeamConnection actions, but not the family.

A fellow team member, Carol, has discovered that the robot’s aperture is not working correctly. To address this problem, she opens a defect. To fix the problem, Alex needs to make some modifications to the parts in this release. This fix will require the tasks noted in the following table:

For information about this task,	Go to this page.
Accepting the defect	48

For information about this task,	Go to this page.
Creating a work area	49
Checking out a part	50
Searching for a part	51
Checking in a part	53
Verifying and testing part updates	54
Freezing the work area	58
Refreshing the work area	59
Building the application	60
Integrating the work area	61
Closing the defect	62

Accepting a defect

Alex is notified via electronic mail that defect 310 has been opened against the robot component. After some research, he agrees that there is a problem with the aperture of the robot's on-board camera, so he accepts the defect. Alex does one of the following:

GUI

From the GUI, he:

1. Selects **Defects** → **Accept** from the Actions pull-down menu on the Tasks window. The Accept Defects window appears.
2. Types 310 in the **Defects** field and selects **program_defect** from the **Answer** list.
3. Selects **OK**.

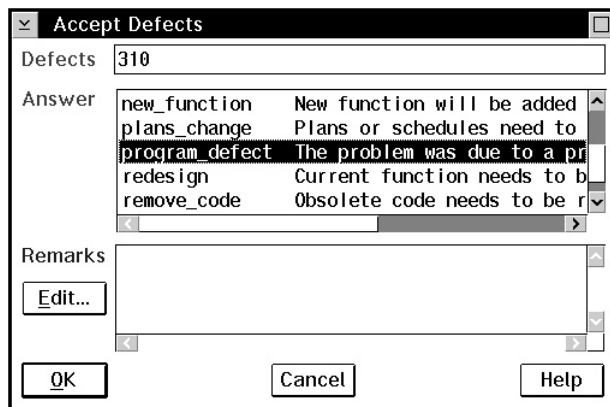


Figure 6. Accept Defects window

Command

From a command line, he issues the following command:

```
teamc defect -accept 310 -answer program_defect
```

Result

The defect goes to the working state.

Creating a work area

Because the component is not following a design, size, and review process, Alex needs to manually create a work area in which to modify and build his parts. (If the component follows a design, size, and review process, a work area is automatically created when the defect moves to the working state, provided that sizing records have been accepted for the defect.)

Before Alex checks out any parts, he creates a work area that will contain the latest view of the parts in the release by doing one of the following:

GUI

From the GUI, he:

1. Selects **Work areas** → **Create** from the Actions pull-down menu on the Tasks window.
2. Types 310 in the **Work areas** field and robot_control in the **Releases** field and selects **OK**.

Note: 310 is the name of the defect that was opened for the problem, so this is how Alex wants to identify the work area.

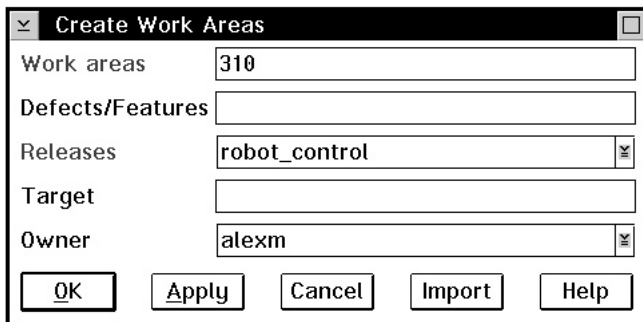


Figure 7. Create Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -create -name 310 -release robot_control
```

Result

TeamConnection creates a work area named 310 associated with release robot_control. The following parts are currently available in the latest view of release robot_control:

brain.c	leg.c
brain.obj	leg.obj
brain.exe	foot.c
arm.c	foot.obj
arm.obj	optics.c
hand.c	optics.obj
hand.obj	

These parts are also visible in the work area 310 because the work area is associated with the release upon creation, and it contains the latest view of the entire release.

Checking out a part

Alex wants to update a subroutine within optics.c, which controls the aperture of the robot's on-board camera. He checks the part out to start the modifications. Because Alex knows the exact name of the part, he does one of the following:

GUI

nt.ide

From the GUI, he:

1. Selects **Parts** → **Check out** from the Actions pull-down menu on the Tasks window.
2. Types the following:
 - optics.c in the **Path names** field
 - robot_control in the **Release** field
 - 310 in the **Work area** field
3. Selects **OK**.

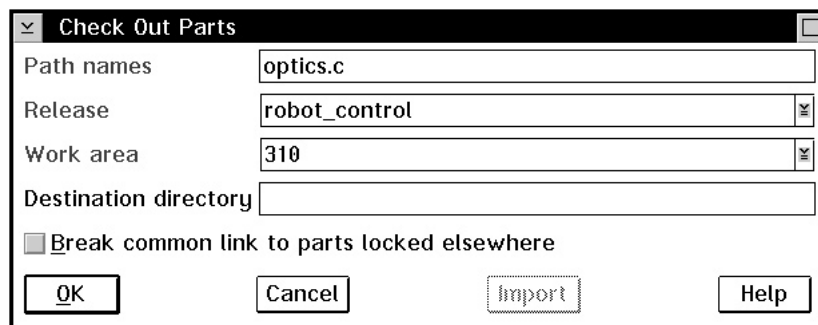


Figure 8. Check Out Parts window

Command

From a command line, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Result

A copy of the part optics.c is checked out from TeamConnection and placed in the directory specified on the Environment page of the Settings notebook of Alex's

TeamConnection client. The part, optics.c, is locked. No other user can update the part until Alex integrates his work area with the release.

Searching for a part

Because Alex knows exactly what part he wants to check out, he specifies the name of the part. If he does not know the name, Alex can use the Parts Filter window or the report command to search for the name. He can do one of the following:

GUI

From the GUI, he:

1. Selects **Parts** → **Parts** from the Objects pull-down menu on the Tasks window.
He does not select the **PartFull** choice because he wants to limit his search to a particular release and work area. He uses **PartFull** when he wants to search for parts across releases, components, or work areas.
2. Types the following in the Parts Filter window:
 - robot_control in the **Release** field
 - 310 in the **Work area** field
 - % in the **Base names** field and selects **like**
3. Selects **Save to Task List**.

Release	Work area	Version
robot_control	310	

Path names	Base names	Components	Current versions	Committed versions
in	like	in	in	in

History	Query	Release	Work Area	Version
	Show all Parts			

Query

OK Apply Clear Save to Task List... Generate Query Cancel Help

Figure 9. Part Filter window

Alex does this because he realizes that he is going to use this query many times, so he wants to add the query to the Tasks window.

4. Adds the necessary information to the Edit Task List window, and selects **Add/Change**.
5. Closes the Edit Task List window. The Tasks window appears.

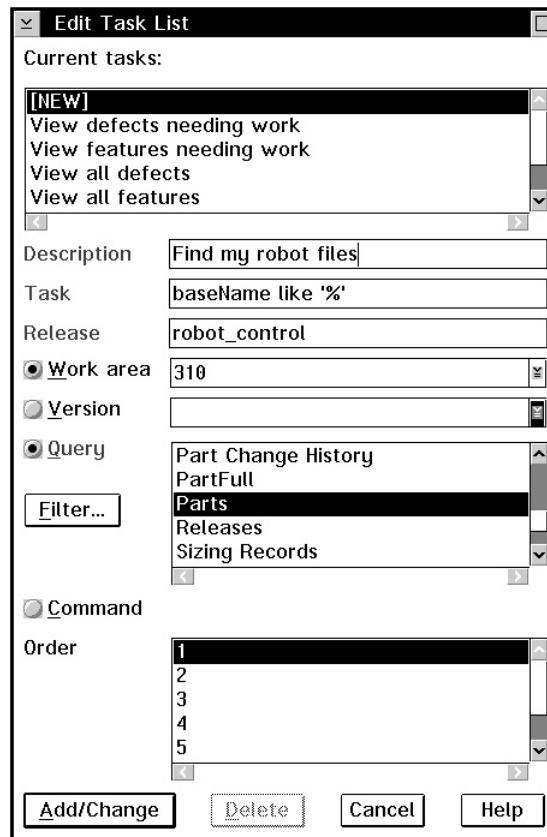


Figure 10. Edit Task List window

6. Double-clicks on the task entry he just created. The Parts window appears. Hereafter, to display the list of parts in his work area, he merely double-clicks on the task entry.
7. Places the mouse pointer over the part name optics.c and presses mouse button 2 to display the pop-up menu.
8. Selects **Check out**. The Check Out Parts window appears with the required fields pre-filled. If Alex provided directory information on the Environment page of the Settings notebook, the **Destination directory** field is pre-filled also.
9. Selects **OK** to check out the part.

Command

From a command line, he issues the following command:

```
teamc report -view partView -where "baseName like '%.c'" -release robot_control
-workArea 310
```

This command returns a list of all the parts that match the query. After Alex determines which part he wants to check out, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Result

A copy of the part optics.c is checked out from TeamConnection and placed in the appropriate directory. The part optics.c is locked. No other user can update the part until Alex integrates the work area with the release.

Checking in a part

Alex edits the part, making the modifications he thinks necessary. Now, he wants to test the modifications. First, he checks the changed part back into his work area.

GUI

From the GUI, he:

1. Selects **Parts** → **Check in** from the Actions pull-down menu on the Tasks window.
2. Types the following in the Check In Parts window, and then selects **OK**:
 - optics.c in the **Path names** field
 - robot_control in the **Release** field
 - 310 in the **Work area** field

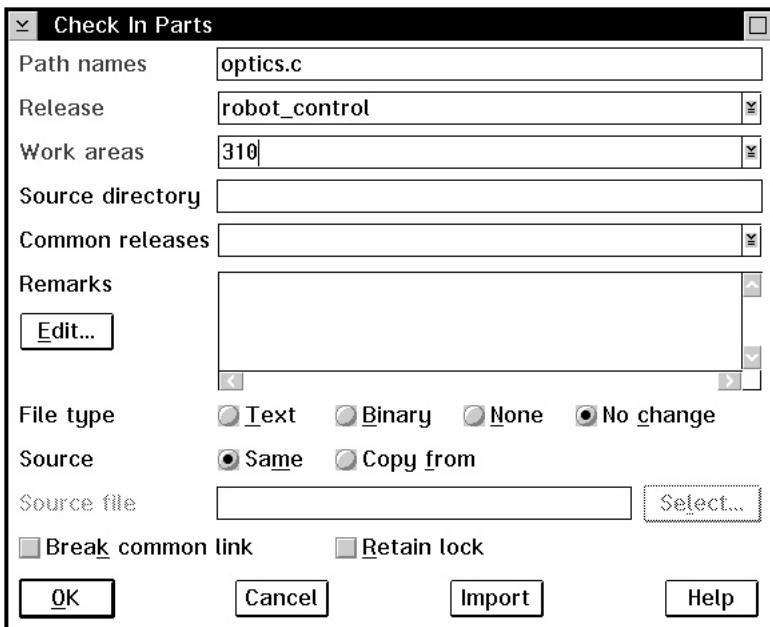


Figure 11. Check In Parts window

Note: Alex follows these steps because he knows the exact name of the part that he is checking in. If he does not know the name, or if he is checking in many parts, he can instead do one of the following to display a list of parts:

- Select the entry on his Tasks window that displays the list of parts.
- Re-open the Parts window if it was previously minimized.
- Add an entry to his Tasks window that lists all of his checked-out parts.

He then selects the parts that he wants to check in.

Command

From a command line, he issues the following command:

```
teamc part -checkin optics.c -release robot_control -workarea 310
```

Result

At this point, it is important to note that the part is checked in to work area 310 and is visible in work area 310 only. The change to optics.c is not visible at the release level or to any other work area. Only the 310 work area contains the change, which is why Alex must specify the work area on the check-out command. Because changes to parts are isolated within work areas, the check-out command must specify which work area to use so that the correct copy of the part is retrieved.

Thus, work area 310 contains the following parts:

brain.c	leg.c
brain.obj	leg.obj
brain.exe	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 1)
hand.c	optics.obj
hand.obj	

Work area 310 continues to contain the unchanged parts from the requested release view, but now the work area is overlaid with changes local to the work area—optics.c in this case. Alex has his own copy of the application that he can modify without impacting other developers. Alex has checked in optics.c; however, the modified part remains locked until the work area is integrated with the release.

Verifying and testing part updates

Alex now requests a build of brain.exe, the high-level program for the robot control application.

GUI

From the GUI, he:

1. Selects **Parts** → **Build** from the Actions pull-down menu. The Build Parts window appears.
2. Types the following, and then selects **OK** to start the build:
 - brain.exe in the **Path name** field
 - robot_control in the **Release** field
 - 310 in the **Work area** field
 - normal in the **Pool** field

The **Pool** field tells TeamConnection which set of build agents will handle this build. Alex got the name of the pool from his build administrator.

Alex could have selected brain.exe from a list of parts on the Parts window, and then selected **Build** from the Selected pull-down menu. This action would have placed some information in the fields, such as the path name and release name.

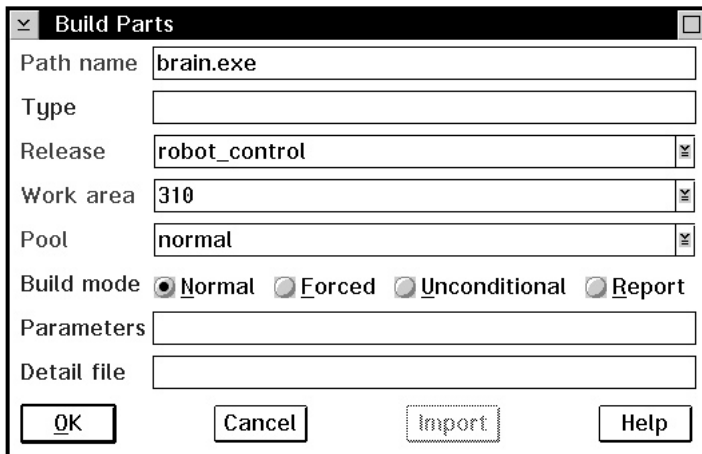


Figure 12. Build Parts window

Command

From a command line, he issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 310 -pool normal
```

Result

TeamConnection determines the parts that are needed for the build from the set of all the part versions that are currently visible from work area 310. The following part versions are selected for build:

brain.c	leg.c
brain.obj	leg.obj
brain.exe	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 1)
hand.c	optics.obj
hand.obj	

After the build is complete, TeamConnection stores the resulting outputs of the build in the work area 310. After the build, the work area contains these parts:

brain.c	leg.c
brain.obj	leg.obj
brain.exe (contains modification 1)	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 1)
hand.c	optics.obj (modification 1)
hand.obj	

Note: For a detailed build example, see “Chapter 12. Building an application: an example” on page 141.

Extracting a part

Next, Alex tests his modifications in the robot prototype in his office. He extracts the executable part from the work area 310.

GUI

From the GUI, he:

1. Selects **Parts → Extract** from the Actions pull-down menu on the Tasks window.
2. Types the following in the Extract Parts window, and then selects **OK**:
 - brain.exe in the **Path names** field
 - robot_control in the **Release** field
 - 310 in the **Work area** field

Alex does this because he wants to extract the .exe part that is in his work area. If he leaves the **Work area** field blank, he gets the latest committed version of the .exe part from the release.

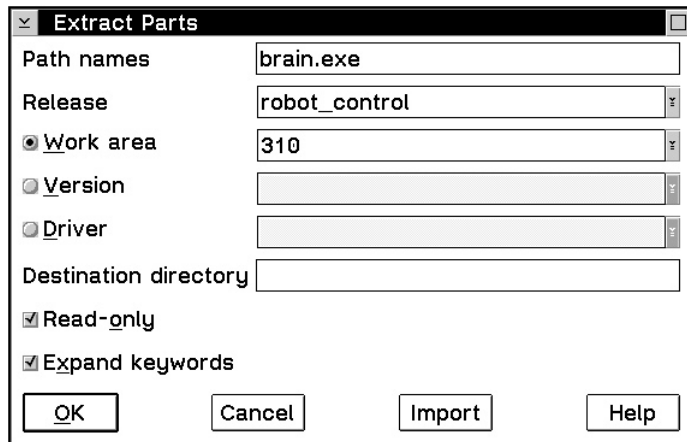


Figure 13. Extract Parts window

Command

From a command line, he issues the following command:

```
teamc part -extract brain.exe -release robot_control -workarea 310
```

Result

This action places a copy of the part brain.exe in the current directory.

Checking out the part one more time

Alex then downloads brain.exe to his robot, runs his test, and determines that the modification did not work: the robot slams into the wall. However, Alex thinks he knows what the problem is, so he needs optics.c for further modifications. First, he checks out the part.

GUI

From the GUI, he:

1. Does one of the following to display the Check Out Parts window:
 - Selects **Parts → Check out** from the Actions pull-down menu on the Tasks window.
 - Selects the entry on his Tasks window that displays the list of parts, and then selects the part.
 - Re-opens the Parts window if it was minimized, and then selects the part.

2. When the Check Out Parts window appears, he types the necessary information and selects **OK**.

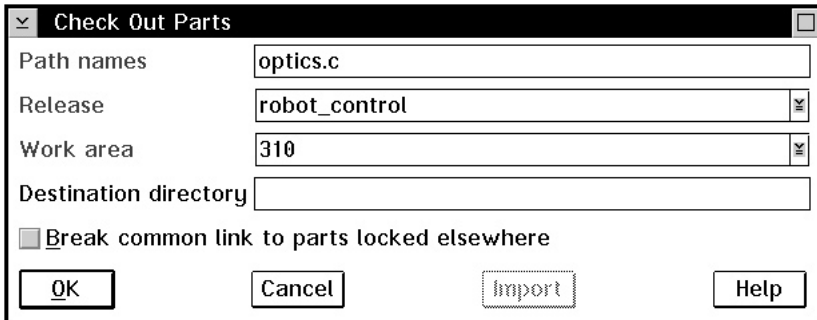


Figure 14. Check Out Parts

Command

From a command line, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Result

A copy of the previously modified optics.c from work area 310 is checked out and placed in the current directory.

Checking the part back in

Alex makes his modification and checks the part in.

GUI

From the GUI, he:

1. Does one of the following to display the Check In Parts window:
 - Selects **Parts** → **Check in** from the Actions pull-down menu on the Tasks window.
 - Selects the entry on his Tasks window that displays all the parts he has checked out, and then selects the part.
 - Re-opens the Parts window if it was minimized, and then selects the part.
2. When the Check In Parts window appears, he types the necessary information and selects **OK**.

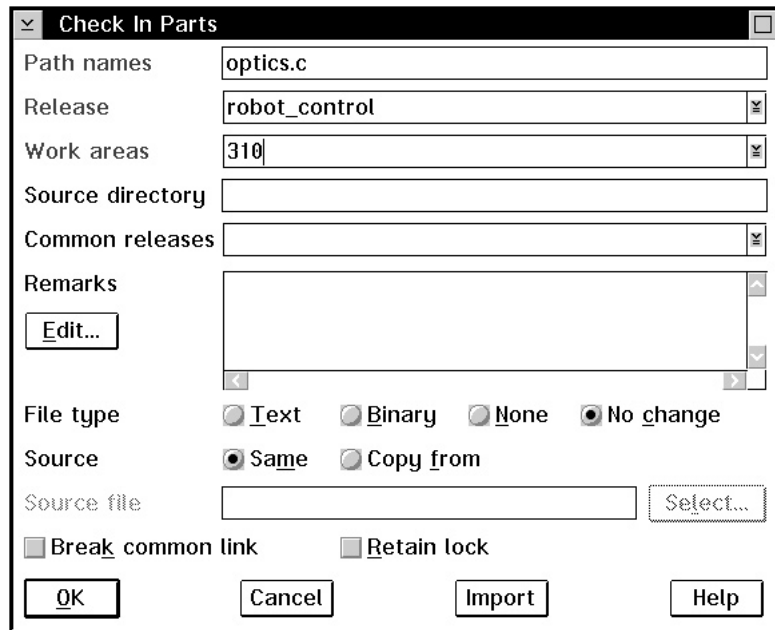


Figure 15. Check In Parts window

Command

From a command line, he issues the following command:

```
teamc part -checkin optics.c -release robot_control -workarea 310
```

Result

Now the work area contains the following parts:

brain.c	leg.c
brain.obj	leg.obj
brain.exe (contains modification 1)	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 2)
hand.c	optics.obj (modification 1)
hand.obj	

Because Alex did not specify that he wanted to save a copy of the work area by freezing it, optics.c (modification 1) was overwritten.

Freezing the work area

Alex builds the application again, extracts the executable part, and runs his test. This time, everything works, and the robot successfully finds its way to the snack machine down the hall without hitting anything. Alex is very pleased, but he notices an unrelated problem in the robot's autofocus system. Before Alex begins repairing the autofocus subroutine, he wants to save a copy of the application as it exists now in his work area. So, Alex does one of the following to freeze the work area:

GUI

From the GUI, he:

1. Displays the Freeze Work Areas window in one of the following ways:

- Selects **Work areas** → **Freeze** on the Actions pull-down menu from the Tasks window.
 - Selects 310 from the list of work areas on the Work Areas window, then selects **Freeze** from the Selected pull-down menu.
2. Types 310 in the **Work areas** field and robot_control in the **Releases** field if the data is not already in the fields.
 3. Selects **OK**.

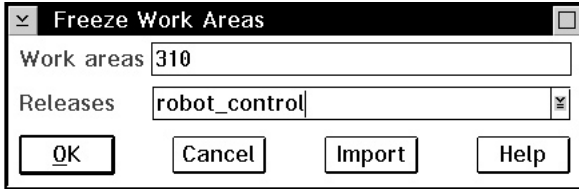


Figure 16. Freeze Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -freeze 310 -release robot_control
```

Result

The freeze command saves the work area 310. Thus, TeamConnection takes a snapshot of the work area, with all its parts and their visible versions, and saves it. Alex can come back to this stage of development in the work area if he wants. Note, however, that a freeze action does not make the changes visible to the other people working in the release, nor does it unlock the parts.

Refreshing the work area

Alex finally finishes his work on the robot's optical systems after making three additional attempts at modifying optics.c and rebuilding the application. Alex modified and rebuilt the application a total of five times in the work area. Now, he wants to share his work with the rest of the team. His work area currently contains the following parts:

brain.c	leg.c
brain.obj	leg.obj
brain.exe (contains modification 5)	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 5)
hand.c	optics.obj (modification 5)
hand.obj	

While Alex worked in his work area, other members of the team were working on their own modifications. Some of these modifications have been integrated with the release, so the copy of the release that Alex has is probably stale. If he were to integrate his changes at this time with the release, he might cause the application to break.

Alex first refreshes his work area with parts from the release by doing one of the following:

GUI

From the GUI, he:

1. Selects **Work areas** → **Refresh** from the Actions pull-down menu on the Tasks window.
2. Types 310 in the **Work areas** field and robot_control in the **Releases** field. Alex wants to refresh from the release, so he does not specify a source.
3. Selects **OK**.

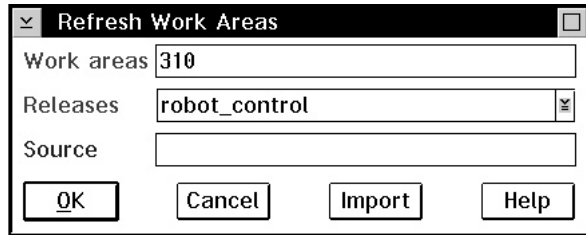


Figure 17. Refresh Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -refresh 310 -release robot_control
```

Result

This action updates work area 310 with any changes from the release, and it also freezes work area 310, if it is not already frozen. Now Alex's work area contains the following versions of parts:

brain.c (Jenny's modification)	leg.c
brain.obj (from Alex's build after refresh)	leg.obj
brain.exe (contains modification 5)	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 5)
hand.c (Joy's and Ken's modification)	optics.obj (modification 5)
hand.obj (from Alex's build after refresh)	

None of the objects that Alex modified and none of the objects built as a result of Alex's modifications is overwritten by the refresh.

Building the application

Alex again builds the application brain.exe within his work area to determine whether his changes integrate with Jenny's, Joy's, and Ken's modifications.

GUI

Alex has a Parts window open with a list of all the parts that exist in work area 310. He highlights the part brain.exe, and then does the following:

1. Selects **Build** from the Selected pull-down menu.
2. Types normal in the **Pool** field. The other required fields have the correct information.

3. Selects **OK** to start the build.

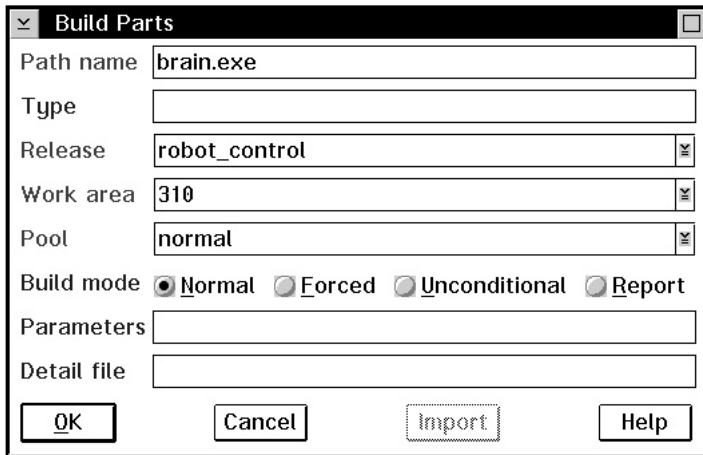


Figure 18. Build Parts window

Command

From a command line, he issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 310 -pool normal
```

Result

Fortunately, nothing breaks, so Alex is ready to integrate his changes with the release.

Integrating the work area

To integrate his changes with the release, Alex must integrate the work area he has been using with the release. This will make the work area visible to all the users in the release. He does one of the following:

GUI

From the GUI, he:

1. Selects **Work areas** → **Integrate** from the Actions pull-down menu on the Tasks window. The Integrate Work Areas window appears.
2. Types 310 in the **Work areas** field and robot_control in the **Releases** field.
3. Selects **OK**.

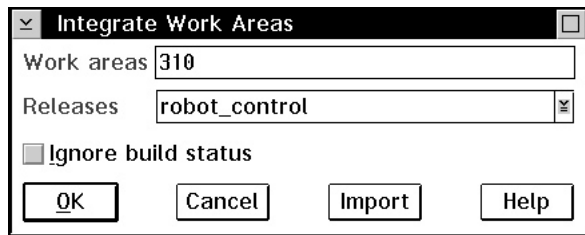


Figure 19. Integrate Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -integrate 310 -release robot_control
```

Result

TeamConnection first determines that Alex's changes were built against the latest version of the release. Then TeamConnection makes Alex's changes visible at the release level so that the other team members can see and use them. The following part versions are now visible from the release:

```
brain.c (Jenny's modification)
brain.obj (from Jenny's build)
brain.exe (from Alex's build)
arm.c
arm.obj
hand.c (Joy's modification, Ken's modification)
hand.obj (from Ken's build)
leg.c
leg.obj
foot.c
foot.obj
optics.c (Alex's modification 5)
optics.obj (from Alex's build)
```

TeamConnection also makes a copy of the release before integrating Alex's changes. If something doesn't work, the users or the administrator can go back to the release prior to Alex's integration. The part, optics.c, is now unlocked in the release. The work area is now in the complete state and can no longer be used.

Closing a defect

Now that Alex is finished making changes to fix the problem reported in defect 310, he is ready to close the defect. He does one of the following:

GUI

From the GUI, he:

1. Selects **Defects** → **Verify** from the Actions pull-down menu on the Tasks window. The Verify Defects window appears.
2. Types 310 in the **Defects** field.
3. Selects **OK**.

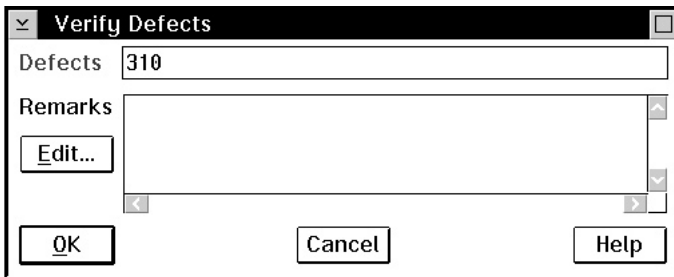


Figure 20. Verify Defects window

Command

From a command line, he issues the following command:

```
teamc defect -verify 310 -release robot_control
```

Result

Because the component does not include the verifyDefect subprocess in its process, the defect moves directly to the closed state.

Working in concurrent development

The previous section discussed working in a serial development environment. While Alex had optics.c in his work area, no one else on the team could check out the part. TeamConnection allows you to hold the part until you are sure that it integrates with the rest of the application. Therefore, the lock is not released until the work area as a whole is integrated with the release.

The scenario changes slightly for concurrent development. In this case, several users can work on the same part at the same time. These users must reconcile their changes as they integrate their work areas with the release.

The following tasks are required:

For information about this task,	Go to this page.
Refreshing the work area	63
Integrating the work area	64
Resolving differences	65

Refreshing the work area

If Alex and Jenny are working on optics.c at the same time, they must resolve their part differences at some point, because both want to make their changes visible to the release. If Alex and Jenny were not required to do this before committing their work areas, the last developer to commit would always overlay the other's changes. For this scenario, assume that Jenny finishes her changes first. The first thing she does is refresh her work area.

GUI

From the GUI, she:

1. Selects **Work areas** → **Refresh** from the Actions pull-down menu on the Tasks window.
2. Types 415 in the **Work areas** field.
3. Types robot_control in the **Releases** field. Jenny wants to refresh from the release, so she does not specify a source.
4. Selects **OK**.

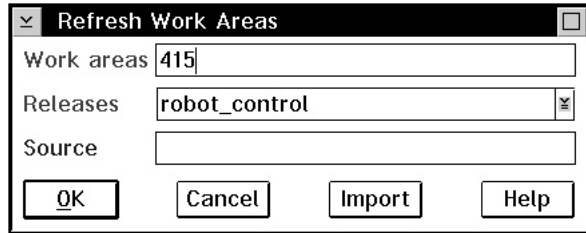


Figure 21. Refresh Work Areas window

Command

From a command line, she issues the following command:

```
teamc workarea -refresh 415 -release robot_control
```

Result

This command refreshes her work area with the latest view of the release. Her work area now contains the following part versions:

```
brain.c (Jenny's modification 3)
brain.obj (Jenny's modification 3)
brain.exe (has Jenny's brain.c modification 3 and optics.c modification 4)
arm.c
arm.obj
hand.c (Joy's modification, Ken's modification)
hand.obj (Joy's modification, Ken's modification)
leg.c
leg.obj
foot.c
foot.obj
optics.c (Jenny's modification 4)
optics.obj (Jenny's modification 4)
```

Integrating the work area

The refresh shows Jenny only the parts integrated with the release. She does not see Alex's work because he has not integrated his work area yet. Jenny rebuilds the application, tests it, and decides she is ready to integrate her changes. She does one of the following:

GUI

From the GUI, she:

1. Selects **Work areas** → **Integrate** from the Actions pull-down menu on the Tasks window. The Integrate Work Areas window appears.

2. Types 415 in the **Work areas** field and robot_control in the **Releases** field.
3. Selects **OK**.

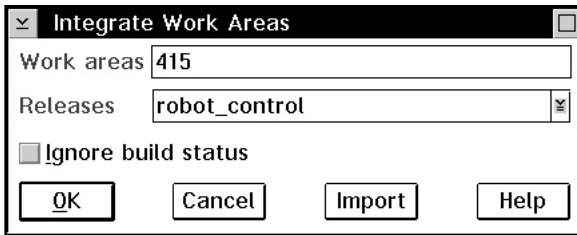


Figure 22. Integrate Work Areas window

Command

From a command line, she issues the following command:

```
teamc workarea -integrate 415 -release robot_control
```

Result

Because Jenny is up-to-date with the latest view of the release, her changes are integrated after TeamConnection preserves a copy of the previous version of the release.

Reconciling differences

Later, Alex is ready to integrate his modifications. Alex issues a refresh command, as Jenny did (see page 63 for instructions).

This time, Alex receives a message that collision records were generated, because both he and Jenny have updated the same parts. At this time he does not know which parts collided. TeamConnection refreshes work area 310 with the exception of the part optics.c, which had the collision. Alex's work area shows the following parts:

```
brain.c (Jenny's modification 3)
brain.obj (Jenny's modification 3)
brain.exe (Contains Alex's modification 5)
arm.c
arm.obj
hand.c (Joy's modification, Ken's modification)
hand.obj (Joy's modification, Ken's modification)
leg.c
leg.obj
foot.c
foot.obj
optics.c (Alex's modification 5)
optics.obj (Alex's modification 5)
```

Alex can use either the GUI or the command line to reconcile the differences. Four steps are required from the command line:

1. Check out the latest uncommitted version.
2. Extract the latest committed version.
3. Run the merge program against the two parts.
4. Check in the resultant part.

However, on the GUI the reconcile action automatically does the preceding steps for you, which can save you a considerable amount of work if several parts require reconciliation.

GUI

From the GUI, he:

1. Selects **Parts** → **Collision Records** from the Objects pull-down menu. The Collision Record Filter window appears.
2. Types 310 in the **Work areas** field and selects **OK**. The Collision Records window appears with optics.c listed as the part having the collision.
3. Highlights the optics.c entry and selects **Reconcile** from the pop-up menu. The Reconcile Collision Record window appears with the required information pre-filled.

Alex does not have to reconcile every part for which a collision record is created. He can choose either his copy or the copy at the release rather than combining the two. For example, if Alex wants to use his copy of optics.c without merging with the copy at the release level, he selects the reject action (of course, he would not do that without first talking with Jenny). If he wants to use the copy of optics.c at the release level without merging any of his changes into the copy at the release level, he selects the accept action.

4. Because Alex wants to combine the two sets of changes, he selects **Merge** to start the TeamConnection merge program, or any merge program of his choice. Alex merges the changes and then exits from the merge program.

The online help provides information on how to use the merge program.

5. Selects **OK** from the Reconcile Collision Record window. TeamConnection checks the resultant part back in.

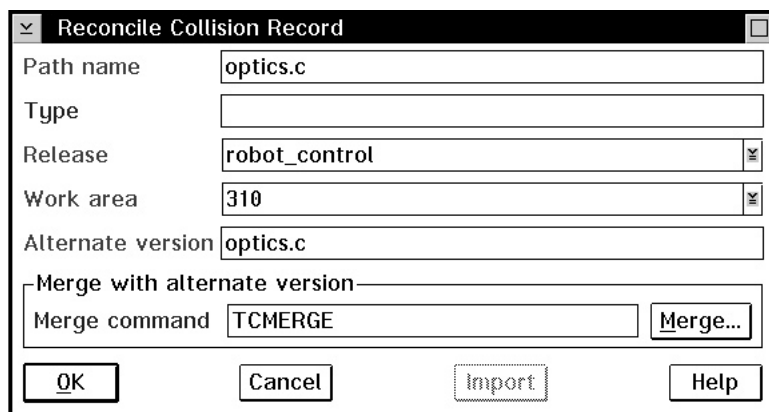


Figure 23. Reconcile Collision Record window

Command

From a command line, he does the following steps:

- Issues a report command to determine which parts are in conflict:

```
teamc report -view collisionView -workarea 310
```

This report tells him that `optics.c` is the part that collided and gives the *alternate version ID* of the part that caused the collision. Alex makes note of the alternate version ID, `robot_control:2`, because he needs to specify that in a later step.

- Extracts a copy of `optics.c` from the release:

```
teamc part -extract optics.c -release robot_control -relative d:\temp
```

By not specifying a work area on the `part -extract` command, Alex ensures that he receives the last committed copy of the part at the release. Also, Alex specifies a *relative* path for the part extract. By specifying the relative directory, he prevents TeamConnection from placing the part in his default directory, where he normally works on checked-out parts. For more information about the `-relative` flag, refer to the *Commands Reference*

- Checks out his copy of `optics.c` from his work area:

```
teamc part -checkout optics.c -release robot_control -workarea 310
```

Because he did not specify a relative path, this part is checked out to his working directory `d:\robot`.

- Uses the merge program to reconcile the two copies of `optics.c`:

```
tcmerge d:\temp\optics.c d:\robot\optics.c -out d:\robot\optics.c
```

If Alex decides not to merge the two parts, but instead wants to use his copy of `optics.c`, he uses the collision `-reject` command. Or, if he wants to use the copy of `optics.c` at the release level, he uses the collision `-accept` command.

- Checks the resultant copy of `optics.c` into his work area and builds it against the rest of the system.
- After he is satisfied with the reconciled changes, he lets TeamConnection know that the previously discovered conflict is reconciled. Alex does this by completing the collision record that TeamConnection created when Alex attempted to integrate his copy of `optics.c`. He does the following:

```
teamc collision -reconcile -path optics.c -release robot_control  
-workarea 310 -altversion robot_control:2
```

Result

Alex is now ready to make his changes visible to the release. He can use either the GUI or the command line to integrate the work area.

The integrate is permitted because a completed collision record exists for the conflict between the two versions of `optics.c`. However, if Ken or Joy had integrated a new version of `optics.c` while Alex was busy resolving the last collision, Alex's integrate would fail. He would have to repeat the collision resolution process.

Chapter 6. Working with component and release processes

The previous chapter described how to work with parts when the release does not follow a tracking process. This chapter describes how to work with parts when a tracking process *is* followed.

When tracking is part of the process, users must associate any changes to their parts with the defects or features active for the release. This association is made through a work area. The work area is the object that ties a defect or feature with a specific release. When checking out a part, the user must specify the work area with which the modification is associated. For any release and defect or feature pair, there can be multiple work area objects.

Aside from their association with a defect or feature, the work areas for a full-tracking process environment are identical to those defined for working in a no-tracking process environment. Work areas maintain a separate view for the user working on the modifications associated with a defect or feature without affecting the release. This view can be integrated with the release at some point. A work area is implicitly created when a defect or feature is accepted if the managing component follows a design, size, and review process for defects and features and if a sizing record is created. The work area that TeamConnection creates is based on the sizing record and has the same name as the defect or feature. If sizing records were not created, you must explicitly create the work area.

As an example of how this all works, suppose that the robot project from the previous chapter is entering system test. The administrator decides to turn on a full-tracking process for the release, such as `track_full`. This process includes the track, approval, fix, driver, and test subprocesses. The release follows concurrent development, and the component follows a design, size, and review process for both defects and features.

On a weekly basis the project leader, Carol, creates a driver. A *driver* monitors and implements the integration of part changes within a release. These part changes are included in a driver by adding the work areas referenced by the changed parts to the driver as *driver members*.

One of the testers for the robot project discovers that the autofocus mechanism in the robot's eye fails when the robot is placed in front of striped wallpaper. The tester must open a defect against the component optics, which is owned by Carol. Carol verifies that the problem does exist, accepts the defect, and assigns it to Alex. This fix will require the tasks noted in the following table:

For information about this task,	Go to this page.
Changing the defect owner	70
Accepting the defect	71
Approving the fix	72
Checking out a part	73
Verifying the changes	74
Freezing the work area	75

For information about this task,	Go to this page.
Building the application	76
Accepting fix records	77
Adding a driver member	78
Returning the work area to the fix state	79
Reactivating the fix record	80
Refreshing the work area	81
Refreshing the driver	81
Building the driver	82
Restricting the driver	83
Integrating the parts	84
Completing the driver	85
Testing the built application	85

Moving through design, size, and review

Because the defect was created against a component that follows the design, size, and review process for defects, Carol must move the defect through this process before the defect can be accepted and parts can be checked out. As the names imply, the process requires that the following be done:

- Design what needs to be done in order to resolve the problem. She must enter design text before the defect can move to the size state.
- Size the amount of work that is required to resolve the problem. At this time, Carol creates a sizing record and specifies robot_control as the release that contains the parts that require changing. If parts in other releases require changing because of the defect, a sizing record is created for each release. A sizing record assures that a work area is created when the defect is accepted. It identifies the work that is required for and the resources affected by the defect or feature. The owner of the component that is referenced in the sizing record is the owner of the sizing record. The owner is responsible for entering information about the amount of work that is required to implement the feature or resolve the problem.
- Review all design text and sizing records and determine if work should continue on the defect.

Changing defect ownership

Because Carol is the component owner, she is currently defined as the owner of defect 456. But the problem is in Alex's code, so she wants him to own the defect. To reassign ownership, she does one of the following:

GUI

From the GUI, she:

1. Selects **Defects** → **Modify** → **Owner** from the Actions pull-down menu on the Tasks window. The Modify Defect Owner window appears.

2. Types 456 in the **Defects** field and types Alex's user ID, alexm, in the **New owner** field.
3. Selects **OK**.

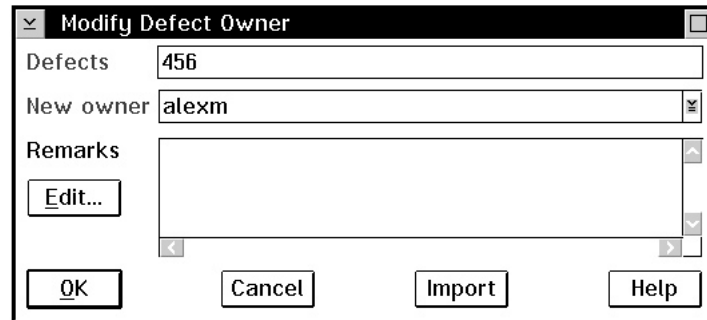


Figure 24. Modify Defect Owner window

Command

From a command line, she issues the following command:

```
teamc defect -assign 456 -owner alexm
```

Results

Alex is now the owner of defect 456. He is responsible for fixing the problem and moving the defect through its various states.

Accepting a defect

When you accept a defect or feature, you accept the responsibility of resolving it. A defect or feature might require changes in more than one release. If the component includes the design, size, and review process, these releases were identified during the size state, and TeamConnection created a work area for each identified release. If the component does not include the design, size, and review process, you will need to create a work area manually.

When the first work area moves to the complete state, the defect or feature automatically moves to the verify state or closed state.

Alex, now the owner of the defect, accepts the defect by doing one of the following:

GUI

From the GUI, he:

1. Selects **Defects** → **Accept** from the Actions pull-down menu on the Tasks window. The Accept Defects window appears.
2. Types 456 in the **Defects** field and selects **program_defect** from the **Answer** list.
3. Selects **OK**.

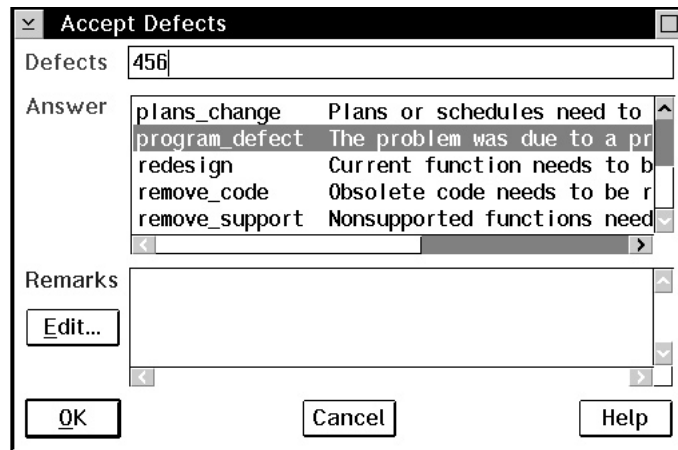


Figure 25. Accept Defects window

Command

From a command line, he issues the following command:

```
teamc defect -accept 456 -answer program_defect
```

Results

Defect 456 moves to working state, and TeamConnection creates a work area called 456. The work area is associated with the release specified on the sizing record, which in this example is robot_control. When the work area is created, a fix record is also created based on the sizing record. Because the approval subprocess is included in the release's process, the work area is created in the approve state and the fix record is created in the notReady state.

Just as with a work area that is explicitly created, the defect work area contains a view of the current versions visible to the release. In this case, the contents of the work area are:

brain.c	leg.c
brain.obj	leg.obj
brain.exe	foot.c
arm.c	foot.obj
arm.obj	optics.c
hand.c	optics.obj
hand.obj	

Approving the fix

Because the full-tracking process includes the approval subprocess, each person identified on the approval list must approve the proposed changes before Alex can begin work on the defect.

Linda and Sam are both listed as approvers. They have been notified by TeamConnection that they have approval records. After reviewing the defect, they do one of the following to indicate their approval:

GUI

From the GUI, they:

1. Select **Records** → **Approval records** → **Accept** from the Actions pull-down menu.
2. Type 456 in the **Work areas** field and robot_control in the **Release** field.
3. Select **OK**.

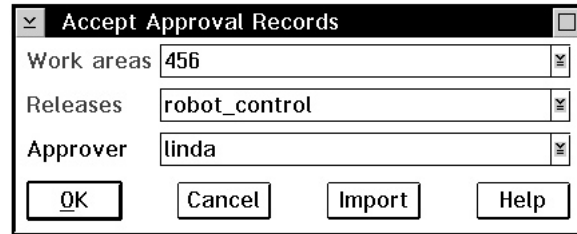


Figure 26. Accept Approval Records window

Command

From a command line, they both issue the following command for the approval record that they have:

```
teamc approval -accept -workarea 456 -release robot_control
```

Results

After both Linda and Sam accept the approval records, TeamConnection moves the work area to the fix state.

Checking out a part

Now that the approval records have been accepted, Alex can check out the necessary parts. He decides that modifications are again required to the part optics.c. So, that is the part he checks out.

Alex must specify the work area on the check-out command so that the part is obtained from the defect's work area. He does one of the following:

GUI

From the GUI, he:

1. Selects **Parts** → **Check out** from the Actions pull-down menu on the Tasks window.
2. Types the following:
 - optics.c in the **Path names** field
 - robot_control in the **Release** field
 - 456 in the **Work area** field
 - d:\robot\src in the **Destination directory** field
3. Selects **OK**.

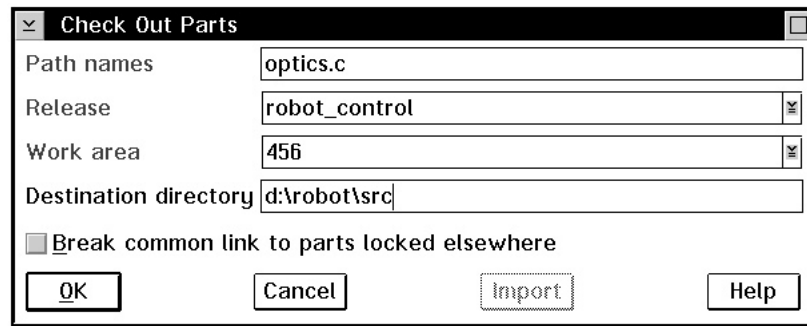


Figure 27. Check Out Parts window

Command

From a command line, he issues the following command:

```
teamc part -checkout optics.c -release robot_control -workarea 456
          -relative d:\robot\src
```

Results

A copy of the part optics.c is checked out from TeamConnection and placed in the directory d:\robot\src. If the directory name is not specified in the command, TeamConnection uses the directory specified in the TC_RELATIVE environment variable. Because the release is following concurrent development mode, other users can also check out and change this part while Alex has it checked out.

Verifying the changes

Alex makes his modifications and wants to test his corrections. First, he must check the part into the work area. He does one of the following:

GUI

From the GUI, he:

1. Selects **Parts** → **Check in** from the Actions pull-down menu on the Tasks window.
2. Types the following in the Check In Parts window, and then selects **OK**:
 - optics.c in the **Path names** field
 - robot_control in the **Release** field
 - 456 in the **Work areas** field
 - d:\robot\src in the **Source directory** field

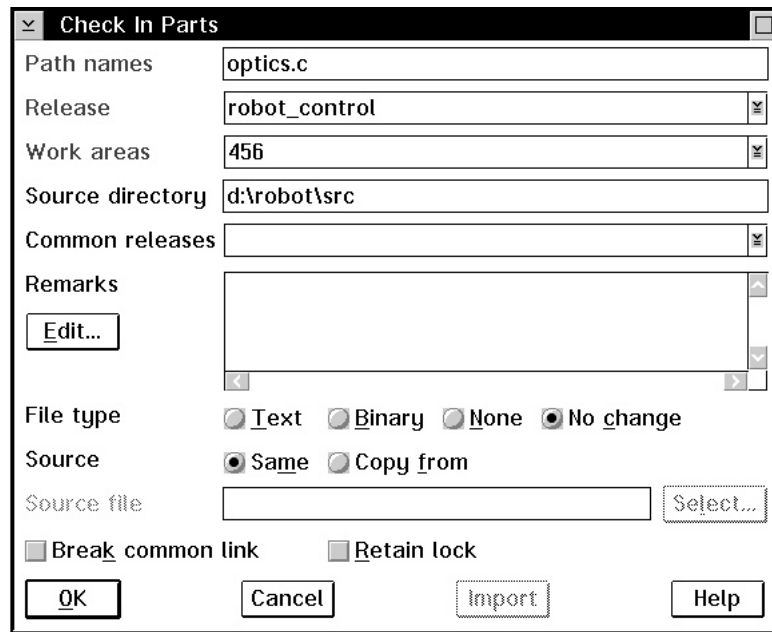


Figure 28. Check In Parts window

Note: Alex follows these steps because he knows the exact name of the part that he is checking in. If he does not know the name, or if he is checking in many parts, he can instead do one of the following to display a list of parts:

- Select the entry on his Tasks window that displays the list of parts.
- Re-open the Parts window if it was previously minimized.
- Add an entry to his Tasks window that lists all of his checked-out parts.

He then selects the parts that he wants to check in.

Command

From a command line, he issues the following command:

```
teamc part -checkin optics.c -release robot_control -workarea 456
```

Results

Now the work area contains the following part versions:

brain.c	leg.c
brain.obj	leg.obj
brain.exe	foot.c
arm.c	foot.obj
arm.obj	optics.c (Alex's modification 1)
hand.c	optics.obj
hand.obj	

Freezing the work area

Alex now wants to save, or freeze, the working system. He does one of the following:

GUI

From the GUI, he:

1. Displays the Freeze Work Areas window in one of the following ways:
 - Selects **Work areas** → **Freeze** from the Actions pull-down menu on the Tasks window.
 - Selects **Work areas** from the Objects pull-down menu on the Tasks window. Types the appropriate search information on the Work Area Filter window to get a list of work areas. Selects 456 from the list of work areas on the Work Areas window, and then selects **Freeze** from the Selected pull-down menu. This method is useful when you are going to be working with several work areas or you are unsure of the work area name.
2. Types 456 in the **Work areas** field and robot_control in the **Releases** field if the information is not already there.
3. Selects **OK**.

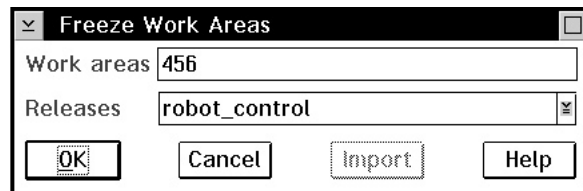


Figure 29. Freeze Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -freeze 456 -release robot_control
```

Results

The freeze command saves the work area 456. Thus, TeamConnection takes a snapshot of the work area, with all its parts and their visible versions, and saves it. Note, however, that a freeze action does not make the changes visible to the other people working in the release. This does not occur until the work area is integrated.

Building the application

Alex now builds the application to verify that the changes he has made have fixed the problem. He does one of the following:

GUI

From the GUI:

Alex has a Parts window open with a list of all the parts that exist in work area 456. He highlights the part brain.exe and then does the following:

1. Selects **Build** from the Selected pull-down menu.
2. Types normal in the **Pool** field. The other required fields are pre-filled with the correct information.
3. Selects **OK** to start the build.

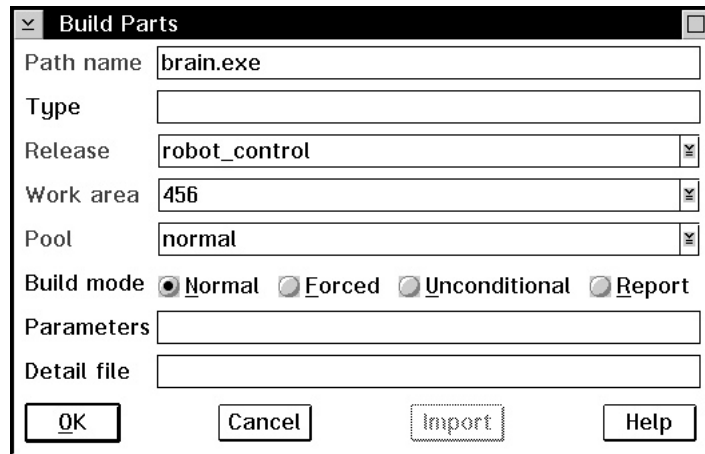


Figure 30. Build Parts window

Command

From a command line, he issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 456 -pool normal
```

Results

Alex builds the application and tests the results. The modification seems to solve the problem.

Note: For a detailed build example, see “Chapter 12. Building an application: an example” on page 141.

Accepting fix records

Alex is satisfied that the changes are complete and the part is ready to be integrated with other parts in the release. He does one of the following:

GUI

From the GUI, he:

1. Selects **Records** → **Fix records** → **Complete** from the Actions pull-down menu on the Tasks window.
2. Types the following in the Complete Fix Records window, and then selects **OK**:
 - 456 in the **Work areas** field
 - robot_control in the **Releases** field
 - optics in the **Component** field

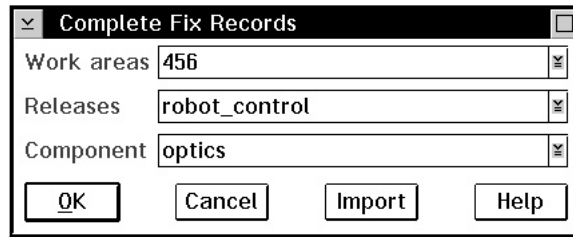


Figure 31. Complete Fix Records window

Command

From a command line, he issues the following command:

```
teamc fix -complete -workarea 456 -component optics -release robot_control
```

Results

The fix record moves to the complete state. Because only one fix record was generated for this defect, the work area moves to the integrate state at the same time. When more than one fix record exists, they all must be completed before the work area moves to the integrate state.

Integrating changed parts into a release

The changes that Alex has made are now ready to be put into the next set of changes scheduled to be integrated with the release. This set of changes is known as a *driver*.

A driver named 0105 currently exists, and several driver members have already been added to the driver. Therefore, the driver is in the integrate state.

Adding a driver member

Carol, the project lead, adds work area 456 as a *driver member* of driver 0105:

GUI

From the GUI, she:

1. Selects **Drivers** → **Add driver members** from the Actions pull-down menu on the Tasks window.
2. Types the following:
 - 0105 in the **Driver** field
 - robot_control in the **Release** field
 - 456 in the **Work areas** field
3. Selects **OK**.

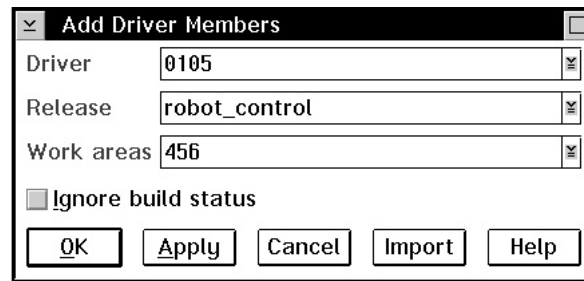


Figure 32. Add Driver Members window

Command

From a command line, she issues the following command:

```
teamc driverMember -create -driver 0105 -workarea 456 -release robot_control
```

Results

Carol previously created a driver member for driver 0105 that included changes to optics.c, so Carol is notified that collisions were detected. (Remember, the release is in concurrent development mode.)

Carol deletes the driver member for work area 456. She then asks Alex to reconcile the collisions.

Reconciling the differences

Before Alex can reconcile the differences, he needs to do the following:

1. Return the work area to the fix state
2. Reactivate the fix record
3. Refresh his work area

Returning the work area to the fix state

The first step in reconciling the differences is for Alex to return work area 456 to the fix state. He does one of the following:

GUI

From the GUI, he:

1. Selects **Work area** → **Fix** from the Actions pull-down menu on the Tasks window.
2. Types 456 in the **Work areas** field and robot_control in the **Releases** field.
3. Selects **OK**.

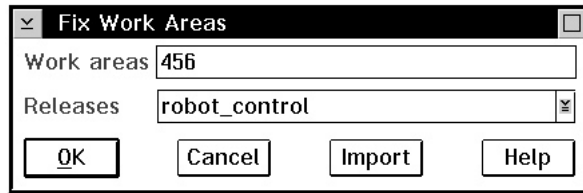


Figure 33. Fix Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -fix 456 -release robot_control
```

Results

Work area 456 is in the fix state. After the fix record is reactivated, Alex will check out optics.c from this work area to reconcile the differences.

Reactivating the fix record

Currently, the fix record for work area 456 is in the complete state. Alex must reactivate the fix record to move it back to the active state so that he can make the necessary changes to optics.c. He does one of the following:

GUI

From the GUI, he:

1. Selects **Records** → **Fix records** → **Activate** from the Actions pull-down menu on the Tasks window.
2. Types 456 in the **Work areas** field and selects robot_control from the **Releases** field and optics from the **Component** field.
3. Selects **OK**.

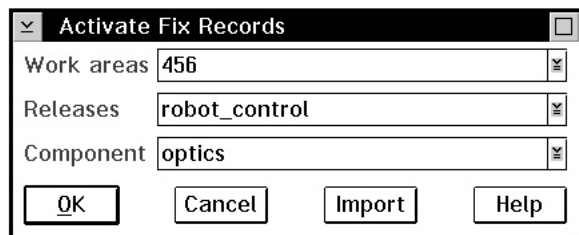


Figure 34. Activate Fix Records window

Command

From a command line, he issues the following command:

```
teamc fix -activate 456 -release robot_control -component optics
```

Results

The fix record returns to the active state.

Refreshing the work area

Alex now needs to refresh his work area with the parts that are already in driver 0105. He does one of the following:

GUI

From the GUI, he:

1. Selects **Work areas** → **Refresh** from the Actions pull-down menu on the Tasks window.
2. Types the following in the Refresh Work Areas window and selects **OK**:
 - 456 in the **Work areas** field
 - robot_control in the **Releases** field
 - 0105 in the **Source** field

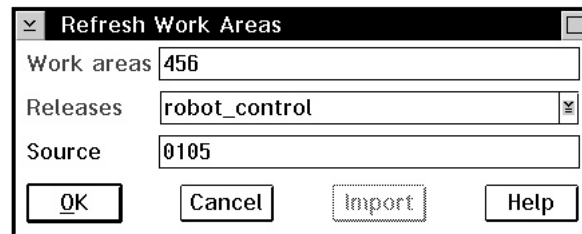


Figure 35. Refresh Work Areas window

Command

From a command line, he issues the following command:

```
teamc workarea -refresh 456 -release robot_control -source 0105
```

Results

TeamConnection notifies Alex of the collision, so his next step is to reconcile the differences. He follows the same procedure that is described on page 65.

Alex completes the fix record and then tells Carol that he has reconciled the part differences and that she can now create the driver member. She creates the driver member without any collisions this time.

Refreshing the driver

Carol is ready to integrate the changes in driver 0105 with the release. Because other team leads have integrated changes as well, she wants to build her driver with the most current release part versions. She does one of the following:

GUI

From the GUI, she:

1. Selects **Drivers** → **Refresh** from the Actions pull-down menu on the Tasks window.

2. Types 0105 in the **Drivers** field and robot_control in the **Release** field.
3. Selects **OK**.

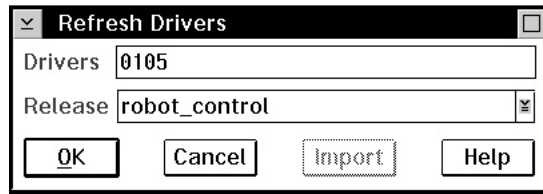


Figure 36. Refresh Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -refresh 0105 -release robot_control
```

Results

This command refreshes driver 0105 with any committed updates to the release.

Building the driver

Carol builds the application using the parts current to driver 0105. She does one of the following:

GUI

From the GUI, she:

1. Selects **Build** from the Action pull-down menu on the Tasks window.
2. Types the following in the Build Parts window:
 - brain.exe in the **Path name** field.
 - robot_control in the **Release** field.
 - 0105 in the **Work area** field.
 - normal in the **Pool** field.
3. Selects **OK** to start the build.

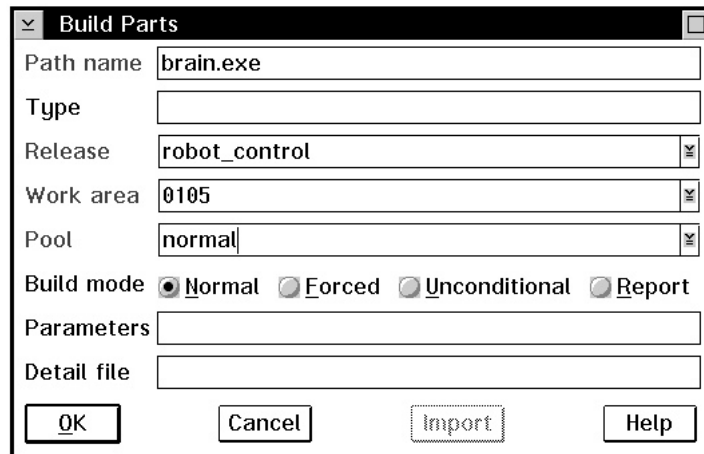


Figure 37. Build Parts window

Command

From a command line, she issues the following command:

```
teamc part -build brain.exe -release robot_control -workarea 0105 -pool normal
```

Results

Carol runs some simple regression tests to verify that the application built properly. She is satisfied with the results, and is ready for the next step—committing the driver changes to the release.

Restricting the driver

After all changes have been integrated with the release, Carol needs to make some final changes before building the driver. To enable her to make these changes while protecting the driver from access by anyone else, she needs to restrict access to it. She does one of the following:

GUI

From the GUI, she:

1. Selects **Drivers** → **Restrict** from the Actions pull-down menu on the Tasks window.
2. Types 0105 in the **Drivers** field and robot_control in the **Release** field.
3. Selects **OK**.

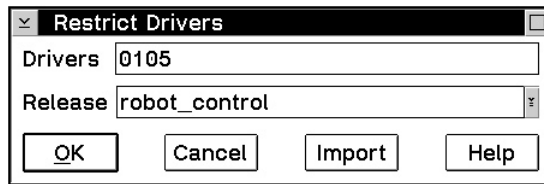


Figure 38. Restrict Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -restrict 0105 -release robot_control
```

Results

This command restricts driver 0105 so that only Carol is able to make changes to it. Carol is now ready to build the application.

Integrating the parts

Carol commits the changes in the driver to the release by doing one of the following:

GUI

From the GUI, she:

1. Selects **Drivers** → **Commit** from the Actions pull-down menu on the Tasks window.
2. Types 0105 in the **Drivers** field and robot_control in the **Release** field.
3. Selects **OK**.

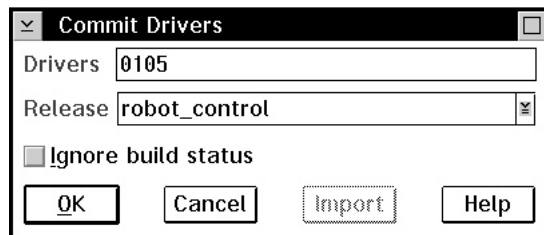


Figure 39. Commit Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -commit 0105 -release robot_control
```

Results

TeamConnection moves the part versions associated with driver 0105 into the release. Other members of the team can now view the changes. Committing a driver commits all work areas designated as driver members and all parts changed in reference to those work areas.

Completing the driver

The driver is ready for formal testing in the specified release's environment list. Testing is tracked using test records for each environment in which testing is to be done. To create the test records, Carol must complete the driver.

GUI

From the GUI, she:

1. Selects **Drivers** → **Complete** from the Actions pull-down menu on the Tasks window.
2. Types 0105 in the **Drivers** field, and selects robot_control from the **Release** field.
3. Selects **OK**.

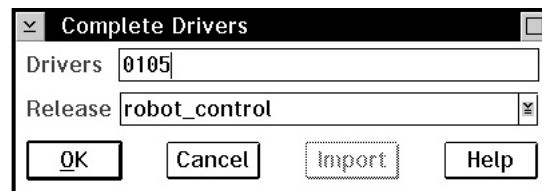


Figure 40. Complete Drivers window

Command

From a command line, she issues the following command:

```
teamc driver -complete 0105 -release robot_control
```

Results

All the work areas in the driver are changed to the test state, and test records are created.

Testing the built application

Annmarie is the tester for the MVS version of the robot application. When she receives notification that the test record is in the ready state, she tests the part changes that were made within the release by Alex and several of his team members. The tests complete successfully, so she accepts the test record by doing one of the following:

GUI

From the GUI, she:

1. Selects **Records** → **Test records** → **Accept** from the Actions pull-down menu on the Tasks window.

2. Types 456 in the **Work areas** field, and selects robot_control from the **Releases** field and MVS from the **Environments** field.
3. Selects **OK**.

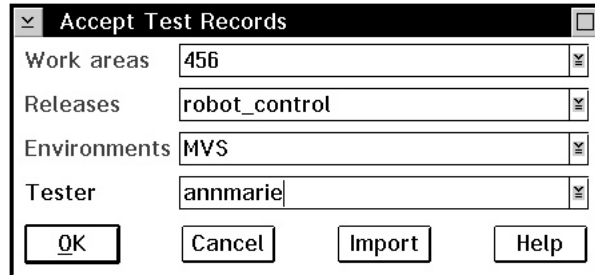
A screenshot of a software dialog box titled "Accept Test Records". It contains four labeled text input fields: "Work areas" with the value "456", "Releases" with the value "robot_control", "Environments" with the value "MVS", and "Tester" with the value "annmarie". Each field has a small dropdown arrow on its right side. At the bottom of the dialog are four buttons: "OK", "Cancel", "Import", and "Help".

Figure 41. Accept Test Records window

Command

From a command line, she issues the following command:

```
teamc test -accept -workarea 456 -release robot_control -env mvs
```

Results

Annmarie's test record moves to the accept state. However, work area 456 will not go to the complete state until Tim, who is the tester for the OS/2 environment, marks his test record.

After all test records are moved from the ready state, the work area moves to the complete state. Because the component process includes the verifyDefect subprocess, defect 456 moves to the verify state. A verification record for the defect is created in the ready state.

Using a configured process

The scenarios in this chapter and the preceding chapter illustrate one release with no process management enabled and another release with full process management enabled. However, administrators can define a release that requires users to work with some intermediate level of process management. That is, the administrator can remove some of the subprocesses from the full-tracking scenario.

For example, the administrator might want to eliminate the driver subprocess. If the driver subprocess is eliminated, the user cannot create driver members to associate the changes in a work area with a driver. Likewise, users cannot commit drivers to integrate several work areas with the release. Instead, users integrate the changes for each work area by integrating the work area with the release.

To demonstrate how this works, assume that Carol and Alex are trying to fix the robot's dislike of striped wallpaper using a release without the driver subprocess enabled. Initially, the scenario is not affected by the absence of the driver subprocess. The defect is opened, and a work area is created. Alex, after receiving notice that he needs to solve the problem, goes through the process of checking

out the faulty part, making fixes, checking the fixes into the work area, and rebuilding. He can still freeze the work area whenever he wants to save its current content.

The difference occurs when Alex is ready to integrate his changes with the release. When the driver subprocess is not enabled, Alex issues the following command:

```
teamc workarea -integrate 456 -release robot_control
```

This command moves the part versions associated with work area 456 into the release so they are visible to other developers. However, if collision records are created, TeamConnection flags the concurrent changes and stops the integration until the changes are reconciled and the corresponding collision records are completed.

Retrieving a past version of a part

Versioning is an insurance policy for the developer. By freezing the work area, the developer knows that the parts currently visible in the work area will be retained in their current form.

For this example, assume that Alex just updated the optics.c module to add support for a new zoom lens. Alex did a considerable amount of work on this task, and it required a dozen check-out, check-in, and build cycles before he finished. Alex's work area now contains the following:

brain.c	leg.c
brain.obj	leg.obj
brain.exe (from Alex's build 12)	foot.c
arm.c	foot.obj
arm.obj	optics.c (modification 12)
hand.c	optics.obj (from Alex's build 12)
hand.obj	

Next Alex must update the brain.c part to set the appropriate conditions for activating the new zoom capability. He does not yet want to integrate his changes to optics.c for the zoom lens with the release because they are of little value without his changes to brain.c. Also, he is not certain that he is completely done with optics.c until he completes the modifications to brain.c. Rather than integrate an incomplete change, he freezes his work area by issuing the following command:

```
teamc workarea -freeze 1208 -release robot_control
```

This command takes a snapshot of the work area and its parts in their current state.

As Alex works on the brain.c module, he makes sweeping modifications to optics.c to simplify the interface between brain.c and optics.c. Unfortunately, he realizes too late that the simplification he is pursuing will not work. Rather than spend several hours removing his updates to optics.c, he wants to start fresh from a copy of optics.c that does not contain the changes for the simplification.

Alex has frozen his work area three times since beginning work on the zoom lens integration. Also, he has done additional check-ins to his work area since his last freeze. He cannot remember the particular version of his work area that contains the copy of optics.c that he wants. So, he wants to see all the versions of his work area that he has saved. He issues the following report command:

```
teamc report -view versionView -where "workAreaName='1208' and  
releaseName='robot_control'" -stanza
```

This command returns a list of the versions frozen from work area 1208. The report looks like this:

```
name          1208:1
workAreaName  1208
releaseName   robot_control
predecessor   robot_control:5
hasSuccessor  yes
releaseVersion no
addDate       1995/01/11 14:30:26
freezeDate    1995/01/11 15:00:00

name          1208:2
workAreaName  1208
releaseName   robot_control
predecessor   1208:1
hasSuccessor  yes
releaseVersion no
addDate       1995/01/12 09:25:13
freezeDate    1995/01/12 17:15:58

name          1208:3
workAreaName  1208
releaseName   robot_control
predecessor   1208:2
hasSuccessor  yes
releaseVersion no
addDate       1995/01/14 11:13:25
freezeDate    1995/01/15 09:01:35

name          1208:4
workAreaName  1208
releaseName   robot_control
predecessor   1208:3
hasSuccessor  no
releaseVersion no
addDate       1995/01/16 08:10:15
freezeDate    1995/01/16 10:05:11
```

So what does it all mean?

- name is the name of the version in the work area.
- workAreaName is the name of the work area that owns the version.
- ReleaseName is the name of the release that owns the version.
- Predecessor is the name of the version that precedes, or is the parent of, this version.
- hasSuccessor has a value of *yes* if the version has a successor, *no* if it does not.
- releaseVersion has a value of *yes* if the version is part of the release's main version history; the value is *no* if the version belongs to a work area.
- addDate is the date and time the version was created.
- freezeDate is the date the version was frozen.

This report seems erroneous. TeamConnection returned four versions in the report even though Alex has executed the freeze command against his work area only three times. The fourth version, 1208:4, is the unfrozen version in which Alex is currently making his changes.

Another concern might be the predecessor of the first version returned in the report. Why is its predecessor robot_control:5? At some point Alex began his work by making modifications to the latest code in the release. The first version of Alex's changes is based on the release version robot_control:5.

After reviewing the report, Alex thinks that his last working copy of `optics.c` was saved when he created version 1208:2. However, to make sure, he wants to see the parts modified in version 1208:2. He issues the following command:

```
teamc report -view partView -version 1208:2 -release robot_control
           -where "currentVersion='1208:2'" -stanza
```

This report returns a list of parts visible to version 1208:2 that have a `currentVersion` (or version ID) of 1208:2. If a part has such a version ID, the part was modified in the version 1208:2.

Note: If the `-where` clause were not specified, the report would return all of the parts visible from version 1208:2.

The TeamConnection system returns the following report:

<code>baseName</code>	<code>optics.c</code>
<code>releaseName</code>	<code>robot_control</code>
<code>compName</code>	<code>robot_dev</code>
<code>versionSID</code>	<code>1208:2</code>
<code>addDate</code>	<code>02/02/94</code>
<code>lastUpdate</code>	<code>04/15/94</code>
<code>pathName</code>	<code>smarts\eyes\optics.c</code>
<code>nuVersionSID</code>	<code>1208:2</code>
<code>nuAddDate</code>	
<code>nuDropDate</code>	
<code>nuPathName</code>	
<code>userLogin</code>	<code>alexm</code>
<code>fmode</code>	<code>0640</code>

Because `optics.c` is the only part modified in version 1208:2, Alex assumes it is the copy he wants. He extracts the part by issuing the following command:

```
teamc part -extract optics.c -version 1208:2 -workarea 1208 -release robot_control
```

This command extracts the desired copy of `optics.c` from the frozen version 1208:2. Alex can then overlay the corrupted copy of `optics.c` that he has checked out with the copy he just extracted, and he can start over fresh. He can also check in the overlaid `optics.c` to his work area.

This method works only for parts with a file type of `TCPart`. If your part has a type of something other than `TCPart`, you can do one of the following to restore the part:

- Use the undo action if restoring to the previous version.
- Use the link action to link to a previous version.

In addition to the reporting features mentioned above, Alex can also obtain a list of work areas by issuing the following command:

```
teamc report -view WorkAreaView -where "releaseName='robot_control'" -stanza
```

The report that is returned lists the work areas in the release `robot_control`. A user can also see the parts changed for each work area by specifying the `-long` parameter on this command.

Part 3. Using TeamConnection to build applications

Chapter 7. Basic build concepts	93
The physical structure of the build function	93
The build object model.	95
Parent-child relationships in a build tree	96
Working with a build tree	98
Putting the pieces together	99
 Chapter 8. Starting and stopping the servers	101
Setting up the mail facility	101
Starting the servers	101
Starting servers from the Family Administrator GUI	101
Starting build servers using teamcbld	102
Caching and the build directories	103
An MVS build server	104
The build cache data sets	104
Customizing the cache data set space attribute	105
Starting a build agent for an MVS build server	105
Creating build startup files	107
Startup file for build servers	107
Startup file for build agents	108
Stopping the servers	108
A build server	108
An MVS build server	109
 Chapter 9. Working with build scripts and builders	111
Creating a builder	111
Writing a build script	115
Passing parameters to a build script.	115
Writing a simple build script	116
Writing an executable file for a build script	117
Testing a build script	118
Modifying the contents of a build script.	118
Putting a builder to work	119
Removing a builder from a part	120
Working with VisualAge C++ and Templates	120
 Chapter 10. Working with MVS build scripts and builders	121
Creating a builder for MVS builds.	121
Writing an MVS build script	125
File name conversions for MVS	125
Passing parameters to an MVS build script	126
TeamConnection syntax for MVS build scripts	127
Supported JCL syntax	128
EXEC statement	128
DD STATEMENT	128
Example of a build script for a C compile	129
Example of a build script for a COBOL compile	131
Example of a build script for a link	132
 Chapter 11. Working with parsers	135
Creating a parser	135
Putting a parser to work	137
Removing a parser from a part	138

Writing a parser command file	139
---	-----

Chapter 12. Building an application: an example	141
Starting the build processors and build agents	142
Creating builders and parsers	143
Creating the build tree for the application	143
Starting the build on the client	147
Determining the build scope.	149
Adding the job to the job queue	151
Picking up the work orders	151
Putting the build processors to work.	151
Putting the build scripts to work	151
Finishing the job and reporting the results to the user	152
Monitoring the progress of a build	152
Running a build in spite of errors	153
Building all parts, regardless of build times	153
Finding out which parts will be built	154
Canceling a build.	154
More sample build trees	155
Defining multiple outputs from a single build event	155
Synchronizing the build of unrelated parts	156

This section tells how to install and use the TeamConnection build function.

Though build administrators will be most interested in this section, anyone who builds an application using TeamConnection will find the first and last chapters helpful.

Chapter 7. Basic build concepts

This chapter defines terms and briefly describes the TeamConnection pieces that work together in building an application. For more details, continue to the other chapters in this section.

The TeamConnection build function has numerous features:

- It builds applications for platforms in addition to those it runs on. Currently you can build applications using TeamConnection on the following platforms: AIX, HP-UX, MVS, OS/2, Windows NT, and Windows 95. For more information on installing, setting up, and using the AIX and HP-UX build servers, refer to the readme file on the installation CD.
- Its graphical representation of the structure of an application makes it easier to visualize and change.
- It lets you build an application using any number of machines working in parallel.
- Because it is fully integrated with TeamConnection's version control system, it ensures that the correct versions of parts are used in a build.
- It can work not only with parts that represent files, such as C source files, but also with parts that represent objects, such as VisualAge Generator applications.
- It can manage other steps related to software packaging and distribution.

For more information, see "Part 4. Using TeamConnection to package products" on page 157 .

The physical structure of the build function

Figure 42 on page 94 shows the structure of TeamConnection:

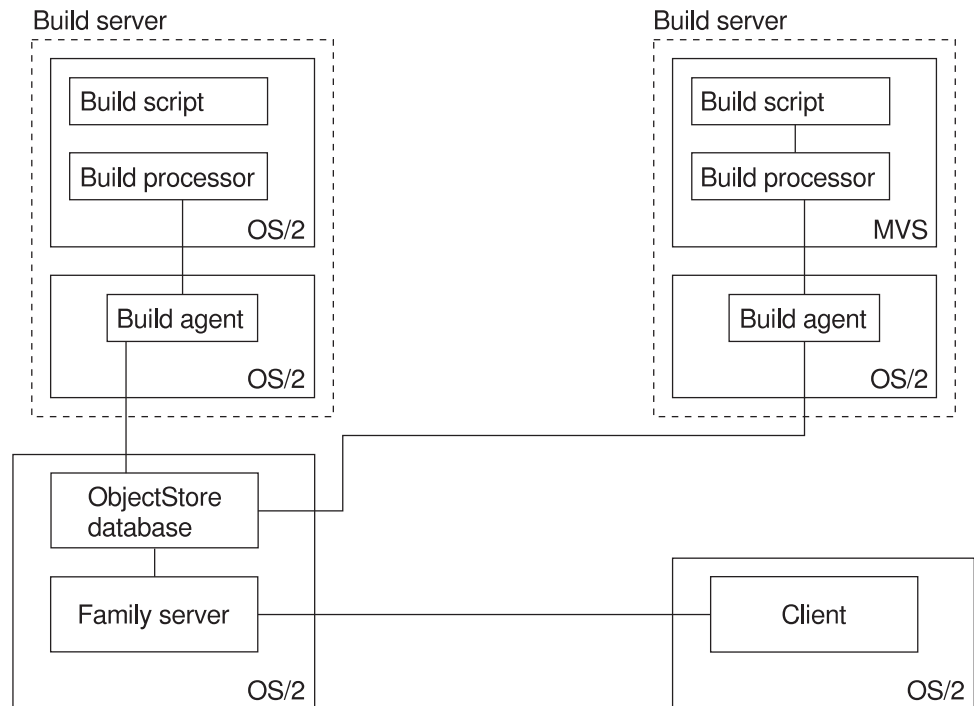


Figure 42. The physical structure of TeamConnection

Of special note to the build function are the build agents and build processors:

Build agent

This program handles access to parts data on behalf of the build processor. Like the family server, the build agent uses an ObjectStore database client.

There can be any number of build agents. Each is connected to one and only one build processor. TeamConnection provides build agents for the following platforms: AIX, HP-UX, OS/2, and Windows NT.

Build processor

This program invokes the tools, such as compilers and linkers, that construct an application. The build processor uses a build script to invoke the tools. It maintains a file cache to reduce file transfer overhead.

There can be any number of build processors. Each is connected to one and only one build agent. TeamConnection provides build processors on the following platforms: AIX, HP-UX, OS/2, Windows NT, Windows 95, and MVS. The term *build server* refers to this combination of build agent and build processor.

Build processors and agents are started by a TeamConnection administrator. For more information, see “Chapter 8. Starting and stopping the servers” on page 101.

Figure 42 shows each part of the build function on a separate machine. However, you could install them in other combinations. For more information, refer to the *Administrator's Guide*

The build object model

Figure 43 on page 97 shows the TeamConnection objects and events that constitute the build function, as illustrated in a sample application named msgcat.exe. This build object model is a conceptual model of the build function. When you use TeamConnection to define a build, you work with a *build tree* (a simplified graphical illustration of the build object model), which you can access through the TeamConnection GUI. “Working with a build tree” on page 98 explains build trees. This section explains the build objects and events represented in a build tree.

In TeamConnection, the build function is always described and discussed in terms of the final output of the build: the product or executable file that the build produces. For the sample application shown in this illustration, msgcat.exe is the build output and appears at the top of the build object model and as the top branch of the build tree illustrated on page 97. When you want to actually build the product, you request a build of msgcat.exe. TeamConnection uses the build tree that you define for this product to determine which objects and build events it needs to generate the final output. The objects and events that TeamConnection uses for a build include the following:

TeamConnection part

An object produced or used during a build, containing any data produced or used by the build. For example, a part called hello.c contains the source code for the application called msgcat. A part might be a text or binary file, or an object such as a VisualAge Generator generic collector.

Build event

An individual step in the build of an application, such as the compiling of hello.c into hello.obj.

A *build scope* is a collection of build events that implement a specific build request. For example, if you start a build of an entire application, TeamConnection creates a build scope containing many build events such as compiles and links.

A *job queue* is a queue of build scopes. One job queue exists for each TeamConnection family. When a developer starts a build, the resulting build scope is added to the job queue for the family. The build agents then process the build events in the build scope, on more than one machine at a time if possible.

Build events, build scopes, and job queues are internal to TeamConnection; you cannot interact with them directly.

Builder

An object that can transform a build event's input parts into output parts by calling tools such as linkers or compilers. For example, one builder might know how to transform the input part hello.c into the output part hello.obj. A different builder might know how to transform hello.obj into msgcat.exe. Builders are associated with the parent, or output part, rather than the child, or input.

Build script

An object that a builder uses in transforming inputs to outputs; it is essentially a binding between TeamConnection and a transformation tool, such as a linker or compiler. In OS/2, Windows, or UNIX environments, a build script is usually a command file, but it can be a string that calls the tool. In MVS, it is a file containing JCL.

Parser

A tool that can read a source file and report back a list of dependencies of that source file. It frees a developer from knowing the dependencies one part has on other parts to ensure a complete build is performed. For example, a C parser can read a C source code file and report back a list of the files included by the source file or by the included files.

Parent-child relationships in a build tree

One relationship that is important to understand and distinguish is the relationship between parent and child parts in a build tree.

Though parent-child relationships usually imply that the parent part generates the child part, in a TeamConnection build it is the opposite. Because TeamConnection places the build output at the top of the tree, it refers to the build output as the parent and to the build input as the child.

A child part can be related to a parent part one of three ways: it can be an input part, an output part, or a dependent part.

Input parts

A part used as direct input to your build. An example of this is a C language source part. If you start a build and this part has changed, the changed part will be part of the new build.

Output parts

A generated output from a build, such as an OBJ or EXE part, or a part with no contents that serves as an organizer object. If you start a build and this part has changed, the changed part will be included in the new build.

Dependent parts

A part needed for the build operation to complete but that is not passed directly to the compiler. An example of this is an include part. If you start a build and this part has changed, the changed part will be included in the new build.

Though parent-child relationships usually imply that the parent part generates the child part, in a TeamConnection build it is the opposite. Because TeamConnection places the build output at the top of the tree, it refers to the build output as the parent and to the build input as the child.

To understand how build output is generated, it may be easier to start at the bottom of the build object model and work your way up. In Figure 43 on page 97, `hello.h` and `bye.h` are C source files that are embedded in `hello.c` and `bye.c`, respectively. The parser, `parser1`, is able to read `hello.c` and `bye.c` to determine files they embed. This build object model contains three build events:

- The builder `compiler1` compiles `hello.c` into `hello.obj`.
- The builder `compiler1` compiles `bye.c` into `bye.obj`.
- The builder `linker1` links `hello.obj` and `bye.obj` into `msgcat.exe`

This build object model contains the following parent-child relationships:

- `msgcat.exe` is the parent of `hello.obj` and `bye.obj`.
- `hello.obj` is the parent of `hello.c`
- `bye.obj` is the parent of `bye.c`

You establish these parent-child relationships between parts when you create the parts in TeamConnection.

Before you can build msgcat.exe, for example, you need to create a place-holder part for it and designate linker1 as its builder. You then create place-holder parts for hello.obj and bye.obj and designate compiler1 as their builder and msgcat.exe as their parent.

“Creating the build tree for the application” on page 143 walks you through an example of creating the build tree for this object model.

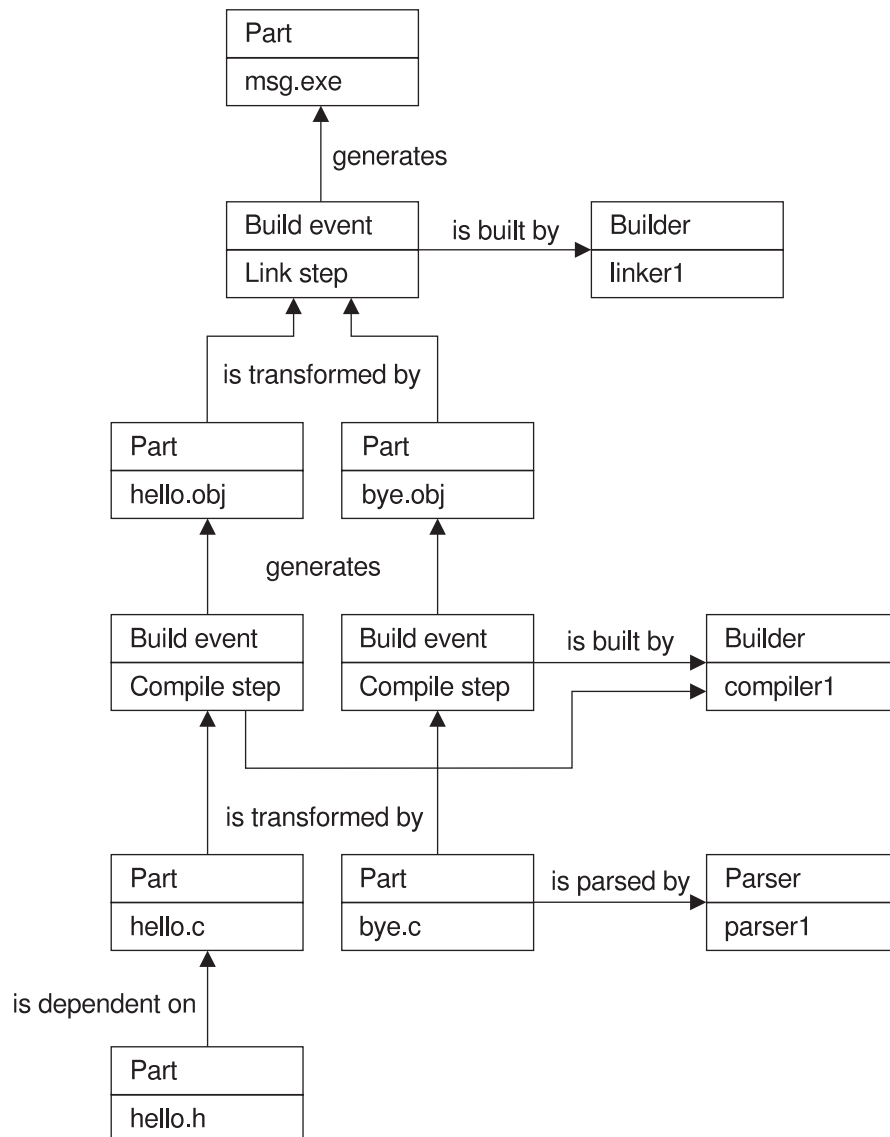


Figure 43. Sample build object model for msgcat.exe

Working with a build tree

Software developers must provide the information by which TeamConnection determines the build events that make up a given build scope. An application's build tree shows this information graphically.

A build tree is a simplified version of the build object model, showing the dependencies that the parts in an application have on one another. If you change the relationship of one part to another, the build tree changes accordingly. Figure 44 shows a build tree for the hello application:

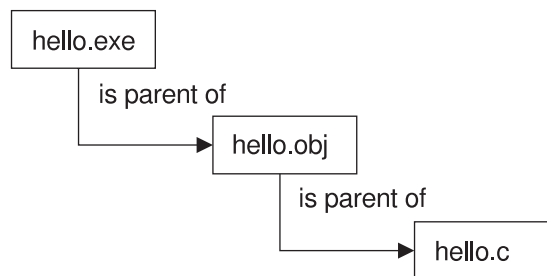
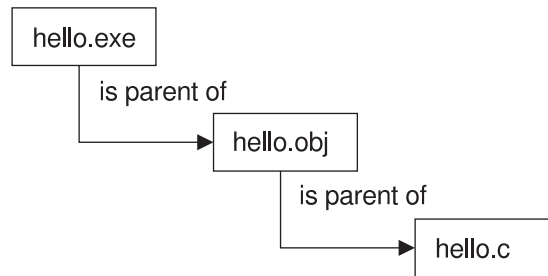


Figure 44. The build tree for the hello application

In this simple application, `hello.c` is compiled to create `hello.obj`, which in turn is linked to create `hello.exe`. The build tree shows that `hello.exe` is the parent of `hello.obj`, which in turn is the parent of `hello.c`. To build the entire application, you request to build `hello.exe`.

Just as the parts that make up an application are versioned, the relationships between these parts are versioned. Thus, more than one version of the build tree can exist. For example, Figure 45 on page 99 shows two different versions of the same build tree:

Work area 817



Work area 915

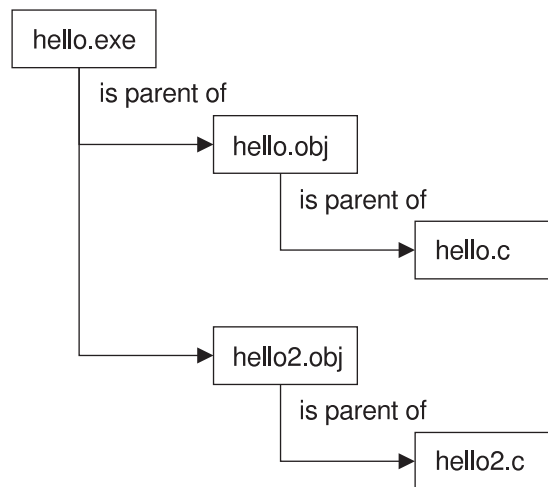


Figure 45. Two versions of a build tree

Putting the pieces together

The table that follows lists the tasks involved in preparing for building an application and in actually building it. Usually an administrator does the preparation steps, but anyone with the proper authority can do so.

For more information about this task,	Go to this page.
Creating build startup files	"Creating build startup files" on page 107
Starting build servers	"Starting the servers" on page 101
Stopping build servers	"Stopping the servers" on page 108
Writing a build script	115
Creating a builder	111
Creating a parser	135
Defining a build tree	143
Starting a build	147
Stopping a build	154

For more information about this task,	Go to this page.
Verifying the parts to be built	154

Chapter 8. Starting and stopping the servers

This chapter explains how to start and stop a build server.

Setting up the mail facility

TeamConnection users can receive notification when certain events occur within TeamConnection. A user's mail address is specified when a TeamConnection user ID is created. TeamConnection uses this mail address to notify users when certain actions occur.

In order for users to receive notification, the notification server must be running. When you start the notification server, you specify an executable or command file that specifies the mail exit routine that processes mail requests.

Two mail exit routine samples are shipped with TeamConnection: `mailexit.cmd` and `mailexit.exe`. These samples are located in the directory where TeamConnection is installed. Both of these samples use the `sendmail` command. You can either use one of these sample mail exits or you can use a different mail facility and write your own routine.

The `sendmail` command is part of TCP/IP, and is installed when TCP/IP is installed. If you use the `sendmail` function to send notification messages, you must configure it on your network in order for TeamConnection client workstations to receive notification messages from the server. Refer to your TCP/IP documentation for more information.

If you use a different mail facility, refer to the shipped mail exit routine sample, `mailexit.c`, to see how you can tailor TeamConnection to support your mail facility.

In order not to lose messages when the mail exit routine fails, you can have the exit routine return a code of 1041. This causes the notification daemon to exit and the mail that was being processed is not deleted. If the exit routine returns any other code, the mail that is being processed is deleted.

Starting the servers

This section explains how to start the TeamConnection family server, the notification server, and the build server. These processes can be started together when you start the family server, or individually.

Starting servers from the Family Administrator GUI

You can follow these steps to start both the family and notification servers from the Family Administrator GUI:

1. Do one of the following to display the TeamConnection Family Administrator window:
 - From the TeamConnection Group folder on the desktop, double-click on the **Family Administrator** icon.
 - Type `tcadmin` from a prompt.
2. Double-click the family icon for the family you want to start. The Server window appears.

3. When starting the family server, specify in the **Daemons** field the number of daemons you want started.
4. To start only one server, select the appropriate **Start** push button. To start both the family and notification servers, select the **Start Both** pushbutton.
When a server starts successfully, the message "Press CTRL-C to stop" appears in the list box and the **Start** push button changes to **Stop**.
5. Minimize the Server window.

Note: Do not close the Server window. Closing the Server window stops the family server.

Starting build servers using teamcbld

You can also start the build servers using the following line command:

```
teamcbld [-e environment] [-p pool] [-f family] [-c] [-s] [-l]
```

Where:

- *environment* specifies the environment that you are building for, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive. You can also set this value using the TC_BUILDENVIRONMENT environment variable.
- *pool* is the name of the build pool. You can also set this value using the TC_BUILDPOOL environment variable.
- *family* is the name of the TeamConnection family. You can also set this value using the TC_FAMILY environment variable.

If the agent is installed on a different machine than the family, this is the fully qualified name: *hostname:dbpath\family_name.tcd*, where

- *hostname* is the name of the machine where the family resides.
- *dbpath* is the drive and path for the family.
- *family_name* is the name of the TeamConnection family.

Note: You can also access the database remotely by issuing a mount command. For example, assume the build server is installed on a machine named bldsrv1, the database is installed on a machine named teamc in the root directory of the d: drive, and your family is named testfam. From bldsrv1, issue the following mount command:

```
mount x: teamc:d:\
```

Also from bldsrv1, set the TC_DBPATH environment variable to x:\. When you issue the following command from the bldsrv1 machine, you will access the family database testfam.tcd on the d: drive of the teamc machine:

```
teamcbld -f testfam -e os2 -p pool1 -s bldsock
```

- **-c** turns caching off. For more information, see "Caching and the build directories" on page 103.
- **-s** sends log file messages to the screen. The build server generates a log file called teamcbld.log. Build server log messages can be routed to the screen using the **-s** parameter.
- **-l** increases the level of messages written to the log. This option is equivalent to the verbose option on TeamConnection teamc commands.



You can also set the **-c**, **-s**, and **-l** build options using the `TC_BUILDOPTS` environment variable. See “Appendix A. Environment Variables” on page 203 for a list of TeamConnection environment variables.

Caching and the build directories

The build server uses two special directories: the build directory and the build cache directory.

When you start a build server, a build directory is created. By default, the name of this directory is `cwd\fhbbuild`, where *cwd* is the name of the current working directory in which the build server was started. For example, if you start the build server in the directory `x:\teamcInstallPath\build` (where `x:\teamcInstallPath` is the drive and directory in which TeamConnection is installed), the build directory is `x:\teamcInstallPath\build\fhbbuild`. You can specify a different cache location by using the `-c` parameter on the `teamcbld` command. In this directory the actual compiles, links, and other data transformations invoked from build scripts are performed. The build server changes its current working directory to this directory before it invokes a build script. The name of the cache directory that you specify on the `-c` parameter must be unique.

All files needed by the build script are extracted from the TeamConnection database into the build directory or some directory subordinate to it. Because a build extracts parts from TeamConnection, anyone requesting a build needs to have PartExtract authority to all parts involved in the build. When a build event is complete, the files in the build directory are moved to the cache directory so they can be used again later. The next time a build event is processed that uses some of the same files as some previous build event, the necessary files are moved into the build directory from the cache directory rather than extracted from the TeamConnection database.

The name of this cache directory is `fhbcache`. You can specify a different name for the cache directory when you start the build server. If you are starting more than one server on the same machine, you must specify a different cache directory for each. How you do that is described in “Starting build servers using `teamcbld`” on page 102 .

A side information file called `fhbhag.***` is maintained in the cache directory of the build server. This file contains information about the files in the cache directory. Do not delete this file, or else the cache directory will be emptied the next time the build server is started.

To control the size of the cache directory, you can set two environment variables:

```
TC_CACHESIZELIMIT=n
TC_CACHEPRUNEMETHOD=value
```

Where:

- *n* is the maximum number of bytes to allow for the build server’s cache directory.
- *value* defines the order in which parts are pruned if the cache directory reaches the value of `TC_CACHESIZELIMIT`. Valid values are the following:

SIZE Largest parts are pruned first.

DATE Least recently used parts are pruned first.

If you specify TC_CACHESIZELIMIT but not TC_CACHEPRUNEMETHOD, the default pruning method is by size.

An MVS build server

To start an MVS build server, do the following to modify the RUNPGM JCL:

1. Add a job card.
2. Modify the STEPLIB DD statement to point to the data set that contains the load module teamcbld.
3. Modify the TEAMCBLD DD statement to point to the data set that will contain all your MVS build scripts.
4. Modify the EDCENV DD statement to point to the data set that contains the environment variables for the fhbmenv.var file.
5. Modify the following statement:

```
//RUNPGM EXEC PGM=TEAMCBLD,PARM='-S @nnnn -K IBM-1047 [-U unit_name] [-T]'
```

Where the:

- -S parameter indicates the port address. The address itself is preceded by an at sign (@). Ensure that the address matches the address specified in the teamagt command for the matching build agent (or the logical address maps to the real address in your workstation TCP/IP hosts and services files).
 - -K parameter indicates the code set that text data is converted to for Because the MVS C compiler defaults to the code set IBM-1047, you need to specify the IBM-1047 code set. When starting its matching build agent, you need to specify the IBM-850 code set.
 - -U parameter indicates the default unit type for dynamic data set allocations. VIO is the default if this parameter is not specified.
 - -T parameter turns trace on.
6. Submit the job.

The build cache data sets

As an MVS build server runs a build script, it creates a cache data set for each unique file extension type, associated with a TCEXT tag, that it encounters. When the build completes successfully, the data sets created for the build are deleted, and their contents are moved to the cache data sets.

Each cache data set is created using the attributes specified in the DD statement associated with the TCEXT attribute. (See “TeamConnection syntax for MVS build scripts” on page 127 for more information about the TCEXT attribute.) If attributes are not specified, these are the defaults used for creating a cache data set:

```
DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
UNIT=VIO
SPACE=(CYL,(30,30,1000))
```

For example, when the build server reads a DD statement that contains a TCEXT=C attribute, it allocates a partitioned data set for caching parts based on a file extension of C. If a DD statement contains the attribute TCEXT=(H,HPP), it allocates a partitioned data set for caching parts based on file extensions of H and

HPP. If, in a later build event, the build server encounters another DD statement with the attribute TCEXT=C, it does not create a new cache data set. Instead, it uses the one created previously.

These cache data sets are associated with the TEAMCBLD address space. They are deleted only when the build server is stopped.

Fragmentation can occur in the cache data sets. Because TeamConnection does not provide a compression tool to manage fragmentation in these data sets, we recommend that you stop and then restart the build server when this situation occurs.

Customizing the cache data set space attribute

The default space for cache data sets is always allocated using cylinders. The primary and secondary space allocations in the space attribute are calculated from the TC_CACHESIZELIMIT environment variable.

To specify the size of the cache directory, you can set two environment variables:

```
TC_CACHESIZELIMIT=n
TC_CACHEPRUNEMETHOD=value
```

Where:

- *n* is the maximum number of bytes to allow for the build server's cache directory.
- *value* defines the order in which parts are pruned if the cache directory reaches the value of TC_CACHESIZELIMIT. Valid values are the following:

SIZE Largest parts are pruned first.

DATE Least recently used parts are pruned first.

If you specify TC_CACHESIZELIMIT but not TC_CACHEPRUNEMETHOD, the default pruning method is by size.

The number of cylinders allocated as primary and secondary is calculated using the following formula:

$$\text{secondary} = \text{primary} = \text{TC_CACHESIZELIMIT} / 716400$$

This number is rounded up to the nearest whole number.

A minimum of 1 cylinder is allocated.

All cache data sets are allocated as partitioned data sets. The number of directory blocks allocated are based on the following formula:

$$\text{directory blocks} = \text{TC_CACHESIZELIMIT} / 22000$$

This number is rounded up to the nearest whole number.

A minimum of 15 directory blocks are allocated.

Starting a build agent for an MVS build server

A build agent handles access to parts data on behalf of an MVS build server. Like the family server, the build agent uses an ObjectStore database client.

There can be any number of build agents. Each is connected to one and only one build server. TeamConnection provides build agents for the following platforms: AIX, HP-UX, OS/2, and Windows NT.

There can be any number of build agents. As you start the build agents, you assign them to build pools. The environment specified in the builder for a particular build event determines which agents in the pool are available to it.

A pool is formed when you start the first build agent that specifies its name. There is no limit to the number of agents that you assign to a build pool.

If you are responsible for starting the build agents, be sure to let others who will be using them know the names of the pools and the kinds of machines assigned to them.

To start an build agent, do one of the following:

- Double-click the **TeamConnection Build Agent** icon. On the pop-up window, type the name of the startup file for the build agent you want to start. For a sample startup file, see the file `teamagnt.fil` in the `TeamConnection \bin` subdirectory. See “Creating build startup files” on page 107 for more information about creating startup files.

To start the build agent without seeing the pop-up or to change the value for the build pool, display the agent's Settings notebook. Type the appropriate values in the **Parameters** field of the Program page. Next time you double-click on the icon, the agent will start without asking for this information.

- From the command line, type the following and press Enter:

```
teamagnt -s socket_port -f familyname -p poolname -e environment
[-k local_codeset]
```

Specify the following attributes in the `teamagnt` command:

- *socket_port* is the TCP/IP socket port. The socket port must match the socket port specified in the `teamcbld` command used to start the corresponding build server. The socket port value can have one of two formats:

- If you have added an alias name for the build socket in your hosts file, you can use that value here. For example, assume your hosts file has the following line:

```
9.12.987.65 tcserv.company.com tcfam bldsock
```

And your services file has the following line:

```
bldsock 7890/tcp # build agent
```

You can specify the socket port like this:

```
teamagnt -s bldsock -f tcfam -p pool1 -e os2
```

- If you have not specified an alias name in the hosts file, you can specify `@hostname@address`, where *hostname* is the name of the host on which the build server to which it will connect is installed, and *address* is the port address.

For example, you can specify the socket port like this:

```
teamagnt -s @bldproc1@7890 -f testfam -p pool1 -e mvs
```

- *familyname* is the name of the TeamConnection family. If the agent is installed on a different machine than the family, this is the fully qualified name: `hostname:dbpath\family_name.tcd`, where
 - *hostname* is the name of the machine where the family resides.

- *dbpath* is the drive and path for the family.
- *family_name* is the name of the TeamConnection family.

Note: You can also access the database remotely by issuing a mount command. For example, assume the build agent is installed on a machine named hag, the database is installed on a machine named teamc in the root directory of the d: drive, and your family is named testfam. From hag, issue the following mount command:

```
mount x: teamc:d:\
```

Also from hag, set the TC_DBPATH environment variable to x:\. When you issue the following command from the hag machine, you will access the family database testfam.tcd on the d: drive of the teamc machine:

```
teamagt -f testfam -e os2 -p pool1 -s bldsock
```

- *poolname* is the name of the build agent pool.
- *environment* specifies the environment that you are building for, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive.
- *local_codeset* is the code set for data stored in TeamConnection. When starting a build agent that connects to an MVS build server, you need to specify the IBM-850 code set for this parameter. Do not use the -k flag for Windows NT build agents. Windows does not support code sets as implemented by TeamConnection and the build agent is not enabled for translation.

This command starts a process that waits for work from the TeamConnection family server, which in turn waits for work from TeamConnection clients.

Creating build startup files

You can create startup files for concurrently starting build servers with the family server using the teamcd command. This is the preferred method for starting build servers. When starting the build servers in this manner, you need to create a startup file.

Information about the build servers is put in a startup file and the name of the startup file is specified in one of two ways:

- In the teamcd command using the -p or -a parameters. See page for more information about these parameters.
- In the TC_BUILD_PROCESSORS_FILE or TC_BUILD_AGENTS_FILE environment variables.

You can store the build startup files wherever you like, provided that you give the full file path names for them in the -p or -a parameters, or in the TC_BUILD_PROCESSORS_FILE or TC_BUILD_AGENTS_FILE environment variables.

Startup file for build servers

Describe each build server on a separate line and specify the following:

```
-s socket_port [-c cache_location -k local_codeset -n]
```

where

- *socket_port* is the TCP/IP socket port. See “Starting build servers using teamcbd” on page 102 for more about this parameter and its format.
- *cache_location* is the fully qualified name of the directory in which caching will take place during builds. For more information, see “Caching and the build directories” on page 103.
- *local_codeset* is the code set that text data is converted to for the build.
- *-n* specifies to erase the contents of the cache directory before starting this build server.

Lines beginning with # are considered comments. Blank lines are allowed. The following is an example of a build server startup file:

```
#-----
# My build server startup file
#-----

-s 9001 -c g:\mycache -n
```

Startup file for build agents

Describe each build agent on a separate line and specify the following:

```
-s socket_port -e environment -p poolname [-k local_codeset]
```

where

- *socket_port* is the TCP/IP socket port. The socket port must match the socket port specified in the build server startup file used to start the corresponding build server. See “Starting a build agent for an MVS build server” on page 105 for more about this parameter and its format.
- *poolname* is the name of the build agent pool.
- *environment* specifies the environment that you are building for, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive.
- *local_codeset* is the code set for data stored in TeamConnection.

Lines beginning with # are considered comments. Blank lines are allowed. The following is an example of a build agent startup file:

```
#-----
# My build agents file
#-----

-s 9002 -e OS2 -p pool1
```

Stopping the servers

A build server

To stop a build server, do one of the following:

- Close the window in which the build server is running.
- Press Ctrl+C when the build server window has focus.
- Close the window in which the family server was started if the build server was started with the teamcd command.

An MVS build server

To stop an MVS build server, cancel the RUNPGM job that was used to start it.

Chapter 9. Working with build scripts and builders

A builder is an object that can transform one set of TeamConnection parts into another by invoking tools such as compilers and linkers. For example, one builder might transform a COBOL source file into an object file. Another might transform a set of object files into an executable file. Builders use build scripts to invoke the tools that actually transform TeamConnection parts.

Usually a build administrator creates build scripts and builders, but anyone with the proper authority can do so. For more information about the required authority, see Appendix H. Authority and notification for TeamConnection actions.

This chapter tells how to create and maintain build scripts and builders. It assumes that you have read “Chapter 7. Basic build concepts” on page 93. The following table directs you to the tasks to be done:

For more information about this task,	Go to this page.
Creating a builder	111
Writing a build script	115
Testing a build script	118
Updating a builder	118
Putting a builder to work	119
Removing a builder from a part	120

Creating a builder

As with most other TeamConnection operations, there are two ways you can create a builder: using the graphical user interface (GUI) or the command line interface.

To create a builder using the GUI:

1. From the Actions pull-down menu on the Tasks window, select **Builders → Create**.
2. On the Create Builder window, type the requested information.

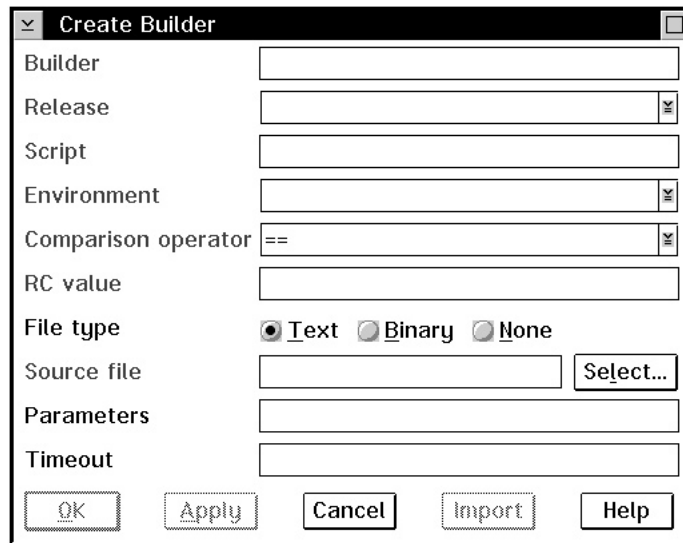


Figure 46. Create Builder window

To create a builder using the command line:

From a command line, type the `teamc builder -create` command and press Enter. The complete command syntax is the following:

```
teamc builder -create name -condition RC_expression
               -environment name
               -from script_filespec
               -script name
               -value RC_value -release name
               -family name
               [-text | -binary | -none]
               [-parameters Parameters]
               [-timeout number] [-become user_name]
               [-verbose]
```

No matter which way you create a builder, you must specify a number of attributes for it. Together with the contents of the build script and the tools you use (the compilers, linkers, and so on), the following attributes define how a transformation takes place.

Builder

The name of the builder must be unique within a release. It can be anything you want; we recommend you establish and follow a meaningful naming convention. An example of a builder name is `c_set_2`.

Release

This is the name of the release that contains the builder. Builders are release-specific objects. They are not versioned within a release; therefore you can have only one version of a builder at any time in a release.

To use the builder from a previous release, you can link to a part that uses it in that release. This action copies the builder to the new release. Otherwise, you must create the builder again in the new release.

Script, File type, and Source file

These fields work together to define the build script that the builder invokes to accomplish the transformation. (The **File type** field on the GUI corresponds to `-text`, `-binary`, and `-none` in the command. The **Source file** field on the GUI corresponds to the `-from` attribute.)

- If the build script is simple enough to be expressed in one line, you can specify it in the **Script** attribute when you create the builder, and specify a file type of none. At minimum, the script must specify the name of the transformation tool. For example, to invoke the C Set/2 compiler, you might specify these values:

File type
none

Script `icc`

See “Writing a simple build script” on page 116 for more information.

- If the build script is more complex, you must first create a separate file containing it; see “Writing an executable file for a build script” on page 117 for more information about how to write it. Specify the fully qualified path name of your file as the source file, and specify the file type as text or binary. TeamConnection can also detect the file type and store it in the proper format.

When the builder is created, this source file is stored as part of the builder in the TeamConnection database; during a build, the build processor creates and runs a local version of this file. Specify the name you want for this local file in the **Script** field. For example, you might specify these values:

File type
text

Script `c_compile.cmd`

Source file
`c:\src\c_compile.cmd`

When this builder is created, the contents of `c:\src\c_compile.cmd` are stored in the builder. When this builder is invoked, TeamConnection creates a file named `c_compile.cmd`, writes the build script into this file, and then runs it.

- If the builder is being used to only collect a set of build objects (for example, a VisualAge Generator collector part), specify these values:

File type
none

Script `null`

This prevents the build agent from extracting input and output parts to send them to the build processor. See “Synchronizing the build of unrelated parts” on page 156 for an example.

Environment

This is the name of the environment supported by the builder, such as OS/2, Windows, AIX, HP-UX, or MVS. The value that you specify here can be anything you like, but it must exactly match the environment value specified in a corresponding build agent. (See “Starting build servers using `teamcbld`” on page 102 for more information.) Again, we recommend you follow a naming convention for this attribute, using values such as `os2` and `mvs`.

Figure 47 shows how the value for environment must match the environment specified in a build agent in order for a build to take place:

teamagnt	-e environment
teamc builder	-environment Name

Figure 47. Matching environment values

Comparison operator and RC value

Together, these two attributes make up a Boolean expression that defines the criteria used to decide whether a specific build event was successfully accomplished, when evaluated against the value returned by the build script. (The **Comparison operator** and **RC value** fields on the GUI correspond to the `-condition` and `-value` attributes in the command.)

The values allowed for **Comparison operator** are as follows:

EQ or **==**

Equals

LT or **<**

Less than

LE or **<=**

Less than or equals

GT or **>**

Greater than

GE or **>=**

Greater than or equals

NE or **!=**

Not equal to

RC value can be any positive integer. An example of a Boolean expression formed from these two attributes is `return_value LE 4`, meaning that the build event is considered a success if the build script returns a value less than or equal to four.

Parameters

This is a string passed to the build script as its argument. If the string includes blanks, enclose the string in double quotes. For example, for a builder used for VisualAge C++ compiling, you might specify a parameter string of `"/Ss /Ge-"`. If the string includes a double quote, precede the double quote with a backslash (`\`). If the string includes a dash (`-`), precede the dash with a blank space, otherwise the string is interpreted as the start of a TeamConnection action flag.

Timeout

This attribute specifies the number of minutes that a build server will wait for an invoked build script to return before concluding an error has occurred and stopping the build event. If this occurs, the build event fails and the build agent will make itself available to process another build event.

Writing a build script

When you create a builder, you must specify a build script. The build script actually invokes the transformation tool and passes it parameters defined in the **Parameters** attribute of the builder.

Passing parameters to a build script

There are three places where parameters can be specified that affect the outcome of a build.

As attributes of a builder

Builder parameters are passed to the build script, after variable substitution is performed. Variables are substituted based upon the following syntax:

`$(variable_name)`

To pass parameters to your build script, specify them in the **Parameters** attribute of the builder. TeamConnection sets these variables before invoking the build script.

In UNIX environments, you need to include an escape character before the `$`: `\$(variable_name)`. The following is an example: `\$(TC_INPUT)`.

You can use the following TeamConnection environment variables:

TC_FAMILY

The TeamConnection family.

TC_RELEASE

The release of the parts that are being built.

TC_LOCATION

The current directory where the build script runs.

TC_INPUT

A list of the TeamConnection parts that are input to the object being built.

TC_INPUTTYPE

Identifies each input type.

TC_OUTPUT

A list of the parts that are being built in this build event.

TC_OUTPUTTYPE

Identifies each output type. The default is file.

TC_WORKAREA

The name of the work area in which the build is being performed.

You can define other variables. These can be set when you start the build by specifying a value for **parameters** in the part -build command (from the command line or through the GUI). These variables are set in the parameters string passed to the build script.

These variables are also used to set OS/2 environment variables before the build script is invoked.

As attributes of a part in the build tree

Parameters that are unique to a particular part are specified on the part

-create and part -modify commands. Like the builder parameters, these parameters allow variable substitution.

When parameters are specified for a part, these parameters are used *in place* of the parameters specified for the builder. In other words, if both builder and part parameters are specified, the part parameters take precedence.

In addition, whenever parameters are specified for *any* part that is an output of a build event, they apply to *all* the outputs of that build event. For example, if a build event has two outputs, msg.exe and msg.map, then changing the part parameters to `"/Debug"` for either of the two parts has the same result. The next time the build event is processed, the `"/Debug"` parameter is used when invoking the build script that produces both msg.exe and msg.map.

You can also substitute the builder parameters into the file parameters by using the variable `$(BUILDERPARMS)`. For example, you might use the following command:

```
teamc part -build myfile.c -parameters "/Ti+ $(BUILDERPARMS)" ...
```

At build time, the parameters specified in the builder for myfile.c are substituted for `$(BUILDERPARMS)`.

As parameters of the part -build command

The part -build command parameters are not used the same way as the other two parameters. Instead, these parameters are used to set the values of environment variables that can be used for substitution into either the builder or part parameters. They are also set in the environment so they can be retrieved by the build script. In other words, they set up the environment used by the builder.

For example, if you issue a part -build command for msg.exe, you can specify `-parameters DEBUG=YES` and, inside of both the compile and link build scripts, retrieve the value of this variable from the environment, setting compiler or linker flags accordingly.

95

The Windows 95 build processor does not allow you to pass other variables to the build script.

Writing a simple build script

This kind of build script is written into the **Script** attribute of the builder. When you create or modify the builder, you specify in this attribute the name of the transformation tool to be invoked.

For example, suppose you want to create a builder that compiles a C source file into a .exe file using IBM's VisualAge C++ compiler. You specify the following attributes for the builder:

Build script
icc

Parameters

```
"$(TC_INPUT) /Fe$(TC_OUTPUT)"
```

You can create this builder using the following command:

```
teamc builder -create c_builder -script icc -parameters "$(TC_INPUT)
/Fe$(TC_OUTPUT)"
```

If you use this builder to create hello.exe from hello.c, the command actually issued by the build processor is the following:

```
"icc hello.c /fehello.exe"
```

Writing an executable file for a build script

Suppose you need to build a C application and you want to specify at build time whether to use debug information. To do this, you define in the builder parameters a variable called *debug* and set the variable when you start the build. In this case, you need a build script that is a separate executable file to pass the debug parameter after the variable substitution.

For a build script of this form, you first write a program or command file; this file is stored in the TeamConnection database when you create the builder. When a build is performed, this build script file is extracted from the database and run. It interprets the parameters passed to it and then invokes the actual transformation tool, such as the compiler.

Our earlier example describes a builder that compiles a C source file into a .obj file using IBM's VisualAge C++ compiler. Using this builder, you can specify at build time whether to use debug information. Here is the complete build script for such a builder, written in IBM's REXX language (it could just as easily have been written in C or COBOL).

```
/* sample C Build Script using debug flag */
parse arg parms

environ = 'OS2ENVIRONMENT'
input   = VALUE('TC_INPUT',,environ)
output  = VALUE('TC_OUTPUT',,environ)
debug   = VALUE('DEBUG',,environ)

if debug = 'YES' then
do
  parms = parms || '/Ti+'
end
icc parms '/Fo'||output input

exit result
```

95

NT

Put your text here.

Windows NT and 95 build scripts must be able to return a value for a return code. Because *.bat command files provide little support for programming logic and cannot return a value, use a compiled executable for your build script. TeamConnection provides two sample Windows build scripts and their source files. These samples, fhwcomp.exe and fhwlink.exe, are C programs for the Microsoft Visual C++ compiler and linker, respectively. Because these samples are C programs, they can also be used with the OS/2 build processor with only slight modifications.

You can create the builder that invokes this build script using the following command:

```
teamc builder -create c_builder2 -script c_compile.cmd -parameters "/c"
            -from d:\teamc\c_compile.cmd
```

Where d:\teamc\c_compile.cmd is the file to be stored in the TeamConnection database and c_compile.cmd is the name of the local file that the build processor creates and runs during a build.

To build hello.obj using the debug option, you use the following command:

```
teamc part -build hello.obj -parameters "debug=YES"
```

The command issued by the build server is the following:

```
c_compile.cmd /c
```

In turn, the build script inspects the contents of the parameters it received in its argument list and from the environment, and it forms this command:

```
"icc /c /Ti+ /fohello.obj hello.c"
```

Testing a build script

The easiest way to test a build script is to write a simple driver program that sets the environment variables that the build script will expect and then runs the script against local files.

For example, to test the example build script in "Writing an executable file for a build script" on page 117, write a program that sets the TC_INPUT, TC_OUTPUT, and DEBUG parameters, and then runs the command file against a local copy of hello.c. If the test is successful, the script correctly builds hello.obj in the current directory, and DEBUG is interpreted correctly.

Modifying the contents of a build script

Sometimes you need to modify the contents of a build script. Remember that a build script is stored as part of the builder itself. Because builders are not versioned, you do not check them out as you would most TeamConnection parts. Instead, follow these steps:

1. Extract the builder (in which the build script is stored) from the TeamConnection database.
2. Make your changes at your workstation.
3. Store the contents back into the TeamConnection database by using the builder -modify command.

For example, to modify the build script in “Writing an executable file for a build script” on page 117, you first issue the following command:

```
teamc builder -extract c_builder2 -to d:\build\c_builder2
```

Then, you use an editor to update d:\build\c_builder2. To move the updated build script back into TeamConnection, you issue the following command:

```
teamc builder -modify c_builder2 -from d:\build\c_builder2
```

The builder is an implied dependency for any part that uses it. Therefore, the next time you build the application that uses the modified builder, all the parts that use it get rebuilt.

Putting a builder to work

For an application to use a builder, the builder must be attached to the TeamConnection parts that it will build.

For an existing part, do one of the following:

- **GUI:** From the Actions menu of the TeamConnection Tasks Window, select **Parts → Modify → Properties**. On the Modify Part Properties window, type the name of the builder in the **Builder** field.

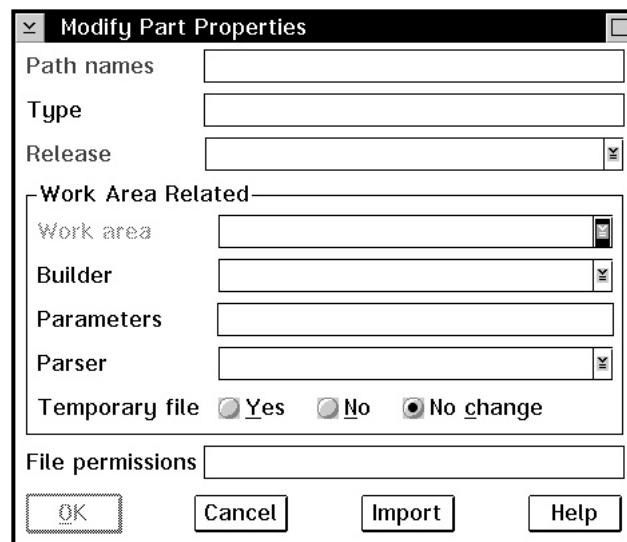


Figure 48. Modify Part Properties window

- From a command line, type the following and press Enter.

```
teamc part -modify name -Builder name
```

where the part *name* is the name of the output file to be created by this builder and the builder *name* is the name of the builder itself.

The complete syntax for this command is described in the *TeamConnection Commands Reference*.

You can also attach a builder to an output file when the part is created.

After you attach a builder to a part, the builder is ready for action. When the part is built, the builder invokes the build script, which in turn invokes a tool to transform the inputs of the part into the output.

For more information about attaching builders to the build tree, refer to “Creating the build tree for the application” on page 143.

Removing a builder from a part

If you no longer want to use a builder for a part, do one of the following:

- From the GUI, select **Parts** → **Modify** → **Properties** from the Actions menu of the TeamConnection Tasks window. On the Modify Part Properties window, type null in the **Builder** field.

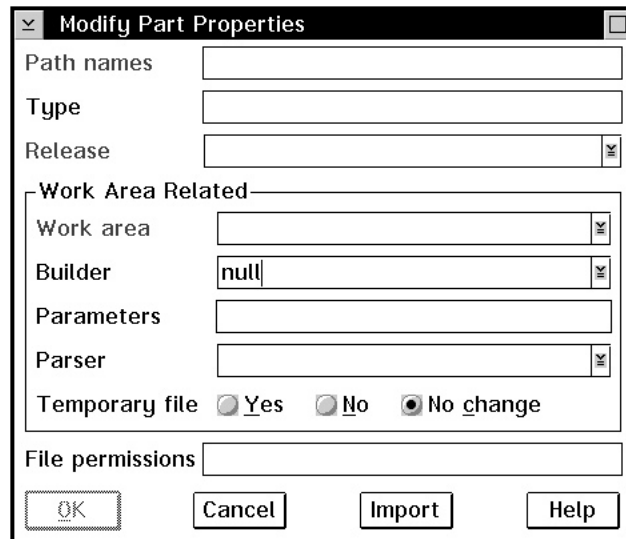


Figure 49. Modify Part Properties window

- From a command line, type the following:
`teamc part -modify name -builder null -release name -family name`

Working with VisualAge C++ and Templates

When using VisualAge C++ and templates, template-include objects are saved in a subdirectory of the current directory called TEMPINC, so that subsequent builds can use them. When you start a build from TeamConnection, you need to specify the /Ft(dir) parameter with your builder or use PRAGMA statements to update the template-include objects for subsequent builds. This parameter suppresses resolution of files and imbeds them within the object file.

You can specify the /Ft(dir) parameter with a builder as follows:

```
teamc builder -create c_builder -script icc -parameters "/FtE:\template"
```

Chapter 10. Working with MVS build scripts and builders

A builder is an object that can transform one set of TeamConnection parts into another by invoking tools such as compilers and linkers. For example, one builder might transform a COBOL source file into an object file. Another might transform a set of object files into an executable file. Builders use build scripts to invoke the tools that actually transform TeamConnection parts.

For MVS, a build script is a text file that contains JCL statements with additional TeamConnection syntax and substitutable variables. TeamConnection parses these statements and does the necessary allocations and program calls for a build.

Usually a build administrator creates build scripts and builders, but anyone with the proper authority can do so. For more information about the required authority, see Appendix H. Authority and notification for TeamConnection actions.

This chapter tells how to create MVS build scripts and builders. It assumes that you have read “Chapter 7. Basic build concepts” on page 93. The following table directs you to the tasks to be done. In some cases, if the instructions are the same for OS/2 and MVS, the table refers you to topics in “Chapter 9. Working with build scripts and builders” on page 111.

For more information about this task,	Go to this page.
Creating a builder for MVS builds	121
Writing an MVS build script	125
Testing a build script	118
Updating a builder	118
Putting a builder to work	119
Removing a builder from a part	120

Creating a builder for MVS builds

As with most other TeamConnection operations, there are two ways you can create a builder: using the graphical user interface (GUI) or the command line interface.

To create a builder using the GUI:

1. From the Actions pull-down menu on the Tasks window, select **Builders → Create**.
2. On the Create Builders window, type the requested information.

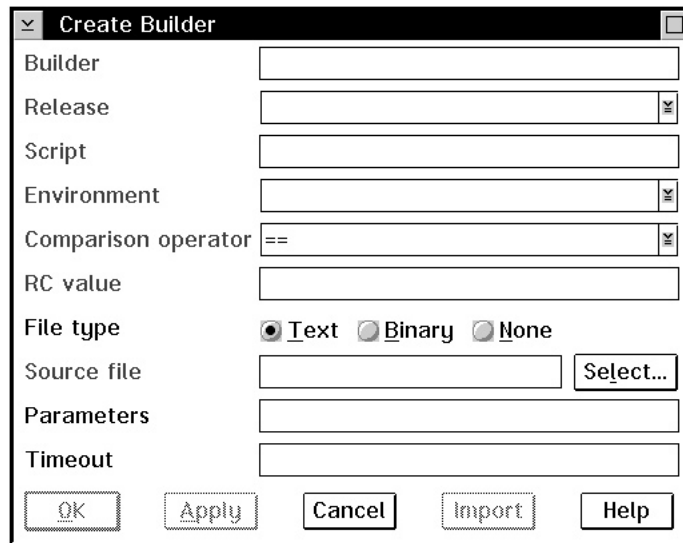


Figure 50. Create Builder window

To create a builder using the command line:

From an OS/2 command line, type the builder -create command and press Enter. The complete command syntax is the following:

```
teamc builder -create name -condition RC_expression
               -environment name
               -from script_filespec
               -script name
               -value RC_value -release name
               -family nName
               [-text | -binary | -none]
               [-parameters parameters]
               [-timeout number] [-become user_name]
               [-verbose]
```

No matter which way you create a builder, you must specify a number of attributes for it. Together with the contents of the build script and the tools you use (the compilers, linkers, and so on), the following attributes define how a transformation takes place.

Builder

The name of the builder must be unique within a release. It can be anything you want; we recommend you establish and follow a meaningful naming convention. An example of a builder name is c370.

Release

This is the name of the release that contains the builder. Builders are release-specific objects. They are not versioned within a release; therefore you can have only one version of a builder at any time in a release.

To use the builder from a previous release, you can link to a part that uses it in the previous release. This action copies the builder to the new release. Otherwise, you must create it again in the new release.

Script, File type, and Source file

These fields work together to define the build script that the builder invokes to accomplish the transformation. (The **File type** field on the GUI corresponds to -text, -binary, and -none in the command. The **Source file** field on the GUI corresponds to the -from attribute in the command.)

You must first create a separate OS/2 file containing the build script. All MVS build scripts must be written using JCL statements and the TeamConnection syntax described in "Writing an MVS build script" on page 125 . You can store the build script one of two ways:

- **To store the build script as part of the builder:** specify the fully qualified path name of your build script file as the source file, and specify the file type as text. When the builder is created, this source file is stored as part of it in the TeamConnection database.

During a build, the build processor creates and runs a local version of this file. Specify the name you want for this local file in the **Script** field. For example, you might specify these values:

File type

text

Script fhbc

Source file

C:\build\script\fhbc.jcl

When this builder is created, the contents of C:\build\script\fhbc.jcl are stored in the builder. When this builder is invoked, TeamConnection creates a file named FHBC in the data set referenced by the TEAMPROC ddname, writes the build script into this file, and then runs it.

- **To store the build script on MVS:** create the build script file and place it in the data set allocated to the TEAMPROC ddname. When you do this, specify the following attributes:

File type

none

Script link

Do not specify a source file.

- If the builder is being used to only collect a set of build objects (for example, a VisualAge Generator collector part), specify these values:

File type

none

Script null

See "Synchronizing the build of unrelated parts" on page 156 for an example.

Environment

This is the name of the environment supported by the builder, such as MVS. The value that you specify here can be anything you like, but it must exactly match the environment value specified in a corresponding build agent. (See "Starting build servers using teamcbld" on page 102 for more information.) Again, we recommend you follow a naming convention for this attribute, using values such as `os2` and `mvs`.

Figure 51 on page 124 shows how the value for environment must match the environment specified in a build agent and in the part -build command in order for a build to take place:

teamagnt	-e environment
teamc builder	-environment Name

Figure 51. Matching environment values

Comparison operator and RC value

Together, these two attributes make up a Boolean expression that defines the criteria used to decide whether a specific build event was successfully accomplished, when evaluated against the value returned by the build script. (The **Comparison operator** and **RC value** fields on the GUI correspond to the `-condition` and `-value` attributes in the command.)

The values allowed for **Comparison operator** are as follows:

- EQ** or **==**
Equals
- LT** or **<**
Less than
- LE** or **<=**
Less than or equals
- GT** or **>**
Greater than
- GE** or **>=**
Greater than or equals
- NE** or **!=**
Not equal to

RC value can be any positive integer. An example of a Boolean expression formed from these two attributes is `return_value LE 4`, meaning that the build event is considered a success if the build script returns a value less than or equal to four.

Parameters

This is a string passed to the build script as its argument. For example, for a builder used for linking load modules, you might specify a parameter string of `list,test`.

Timeout

This attribute specifies the number of minutes that a build agent will wait for an invoked build script to return before concluding an error has occurred and stopping the build event. If this occurs, the build event is queued again to be processed by the next available build agent.

Because MVS builds are processed in batch mode but the build is submitted to the build agent in real time, consider writing a user exit to check the time of day before allowing a build request to be submitted. Another approach to handling the timing of MVS builds is to start the MVS build agent only at night and ensure that the MVS builders do not have short timeout values.

Note: The ddname TEAMPROC must be defined to a shared data set when the MVS build processor is started; see “An MVS build server” on page 104 for more information. This data set is used as a cache for the build scripts of MVS builders.

Writing an MVS build script

The best starting point for an MVS build script is an existing JCL fragment that is used for transforming inputs into outputs. For example, suppose you want to create a builder that compiles a C source file into an OBJECT file using IBM's C/370 compiler. You probably already have JCL that can be submitted as a batch job that does this.

When you create a build script for the MVS environment, you specify JCL statements with additional TeamConnection syntax. This build script is parsed by the build processor. From the parsed results, TeamConnection allocates the specified ddnames and data sets; it then determines and executes the programs dynamically. The MVS build processor uses the specialized TeamConnection syntax in the JCL to determine where to store the parts involved in an MVS build.

All statements in the MVS build script except for comments and inline data stream must start with two forward slashes (/ /).

Before you start writing your build script, refer to the manuals for the compiler, linker, or other transformation program to determine the data set requirements. Pay particular attention to the DCB attributes for LRECL, BLKSIZE, and RECFM.

Sample build scripts shipped with TeamConnection can be installed on MVS. Page 262 lists the sample build scripts. For instructions on installing these samples, refer to the *Administrator's Guide*

If you are debugging a build script, these manuals are also the first place to look for problems.

For more information about JCL syntax, refer to the *JCL User's Guide* and *JCL Reference* for your version of MVS. (These are listed in the bibliography at the back of this book.)

The following sample MVS build scripts are shipped with TeamConnection:

fhbmasm.jcl

Calls the MVS assembler

fhbcobm.jcl

Calls the MVS COBOL compiler

fhbmpli.jcl

Calls the PL/1 MVS compiler

File name conversions for MVS

TeamConnection file names are modified by the MVS build server according to the following rules:

- The directory path of a file name is not used. All characters of a file name up to and including the rightmost slash (/ or \) are thrown away.

- Lowercase characters are converted to uppercase characters.
- The file extensions are stripped from the right, up to and including the leftmost period. The extension, minus the period, is used by the MVS build tool to direct the file to particular data sets according to user-specified syntax in the MVS build scripts.
- The remaining name is truncated from the left, to a maximum of 8 characters.
- Names must contain characters that are valid in MVS. MVS allows the following characters:

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ\$@#

However, the name must begin with an alphabetic character.

- Underscore characters (_) in a base name are converted to at signs (@).

The following are examples of how a TeamConnection name is converted:

- A TeamConnection file name of src\build\fhbldobj.C is converted to FHBLDOBJ on MVS.
- A TeamConnection file name of src/build/fhbtruncate.c is converted to FHBTRUNC on MVS.

In both of these examples, the .C or .c is split away. The MVS build processor uses the resulting C extension to resolve and possibly allocate the MVS data sets needed for the build process.

A TeamConnection file name of src\build\fhbtest.c.old is converted to FHBTEST, and c.old becomes the extension.

Passing parameters to an MVS build script

To pass parameters to your build script, specify them in the **Parameters** attribute of the builder. These are passed to MVS through the combination of the PARM keyword parameter on an EXEC card and the &TCPARM variable.

Note: Take extra care to use no single or double quotes in the **Parameters** attribute of the MVS builder definition. This rule follows standard JCL syntax for parameter substitution in the PARM keyword parameter of an EXEC statement.

You can use the **&TCPARM** variable in your MVS build scripts. This variable is substituted with any parameters that were specified using the -parameter attribute of the builder command or the **Parameters** field on the Create Builder window when the builder was created.

You can also use the following TeamConnection variables in writing MVS build scripts:

&TCINPUT

This variable is used for in-stream data. For each build input, the line where &TCINPUT appears is duplicated and the variable &TCINPUT substituted with the input name.

&TCOUTPUT

This variable is used for in-stream data. For each build output, the line where &TCOUTPUT appears is duplicated and the variable &TCOUTPUT substituted with the output name.

&TCWKAREA

The name of the work area in which the build is being performed.

&TCRELEAS

The name of the release in which the build is being performed.

Note: The &TCINPUT and &TCOUTPUT substitutable variables have limited scope in the MVS build scripts and should be used only within the in-stream data.

You can define other variables. You can set them by specifying a value for **Parameters** when you start a build. These variables are set in the parameters string passed to the build script.

Further, these variables can be used for variable substitution within MVS build scripts. Variable substitution works similarly to JCL variable substitution.

TeamConnection syntax for MVS build scripts

TeamConnection has extended the existing JCL syntax. The extended syntax tells the TeamConnection build processor where to put the inputs, where to get the outputs, and where to get messages from the translators after an MVS build.

To direct inputs, outputs, and messages, add TCEXT=xxx to the data set attributes defined to a ddname, where xxx is one of the following:

- The base name extension from the TeamConnection part—for example, TCEXT=H, where H is the extension from A.H.
- One or more base name extensions from TeamConnection parts, surrounded by parentheses—for example, TCEXT=(H,HPP), where H is an extension from A.H or HPP is an extension from A.HPP.
- The string TCOUT, which declares that the contents of the data set assigned to the ddname will be sent back to TeamConnection. Users can view this information in one of these ways:
 - On an OS/2 command line, typing `teamc part name -viewmsg` and pressing Enter
 - Selecting **Part → View → View build message** from the Actions pull-down menu on the Tasks window

When you add the TCEXT attribute for a ddname specification, you must also specify other attributes to allocate the data set through dynamic allocations:

- SPACE
- UNIT
- DCB, which includes the LRECL, BLKSIZE, and RECFM attributes

The UNIT attribute defaults to VIO unless the -U parameter is specified when the MVS build processor is started.

For translation messages, you can allocate a data set to the ddname TC\$LIST and specify the attributes yourself. Otherwise, the build processor allocates this data set with the following attributes by default:

```
//TC$LIST DD DCB=(RECFM=VB,LRECL=255,BLKSIZE=32640),  
// SPACE=(CYL,(2,1)),DISP=(NEW,DELETE),UNIT=VIO
```

Supported JCL syntax

The TeamConnection MVS build processor supports only a subset of the available JCL syntax.

The following are not supported:

- A JOBSTEP statement
- DISP=(...,PASS)...

JCL procedures can be used on an EXEC statement. However, you must verify that any procedure called by the build script uses syntax that TeamConnection supports.

The following list indicates the positional and keyword parameters that are supported. You can verify the syntax in the *JCL Reference*.

EXEC statement

```
//label EXEC positional_parameter,keyword_parameter
```

The following parameters are supported.

Positional parameters:

- PGM=*program_name*, where *program_name* is an executable load module
- PROC=*procedure_name*, where *procedure_name* is an existing JCL procedure
- *procedure_name*, where *procedure_name* is an existing JCL procedure

Keyword parameters:

- PARM='*information*', where *information* is the parameter string passed to the load module.
- COND=(*code*,*operator* [,*stepname*])
 - *code* is the value to test against the return code from a previous step
 - *operator* is the comparison to be made between the value for *code* and the return code
 - *stepname* is the step issuing the return code

All other keyword parameters are ignored and not used.

DD STATEMENT

```
//label DD keyword parameter
```

Positional parameters

The only supported positional parameter is [*], which begins an in-stream data set containing no JCL.

Keyword parameters

The following keywords are supported.

- DSN=*data_set_name* or DSNAME=*data_set_name*
- DISP=*status* or DISP=([*status*] [,*normal-termination-disp*] [,*abnormal-termination-disp*])
 - Valid values for *status* are NEW, OLD, SHR, or MOD.

- Valid values for *normal_termination_disp* or *abnormal_termination_disp* are DELETE, KEEP, CATLG or UNCATLG.
- UNIT=*unit_type*, where *unit_type* is any value allowed in JCL. The default is VIO unless a different default is set when the MVS build processor is started.
- SPACE=(*allocation_type*,(*primary*[, *secondary*] [,*directory*]))[,RLSE][,CONTIG])
 - Valid values for *allocation_type* are TRK, CYL, or the block size.
 - *primary* is the primary number of the allocation type.
 - *secondary* is the secondary number of the allocation type.
 - *directory* is the number of directory blocks for a partitioned data set.
- DCB=(LRECL=record_length,BLKSIZE=block_size,RECFM=record_format)
Valid values for *record_format* are F, FB, V, VB, or U).
- DSORG=*data_set_organization*
Valid values for *organization* are the following:
 - P0 for a partitioned data set
 - PS for a sequential data set
- DDNAME=*label*, where *label* is the later ddname label reference. This parameter is supported only for simple cases.
- SYSOUT=*class*
This will always be allocated as a DUMMY DSN.

All other keyword parameters are ignored and not used.

Example of a build script for a C compile

The following JCL can be submitted as a batch job to do the following:

- Compile the source file member in the data set WELLSK.TEAMC.C
- Produce an object file member in the data set WELLSK.TEAMC.OBJ
- Produce a listing of the source file in the file member in the data set WELLSK.TEAMC.LISTING
- List the compiler messages in the file member in the data set WELLSK.TEAMC.ERROR

```

//COMPILE EXEC PGM=EDCCOMP,
// PARM='LO,SSCOMM,NOSEQ,NOMAR,LIS,FL(I),SO,DECK,TEST',
//      REGION=1536K
//STEPLIB DD DSN=SYS1.EDC.SEDCCOMP,DISP=SHR
//      DD DSN=SYS1.EDC.SEDCLINK,DISP=SHR
//      DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYMSGSGS DD DSN=SYS1.EDC.SEDCDMSG(EDCMSGE),DISP=SHR
//SYSIN DD DSN=WELLSK.TEAMC.C(MEMBER),DISP=SHR
//USERLIB DD DSN=WELLSK.TEAMC.H,DISP=SHR
//SYSLIB DD DSN=SYS1.EDC.SEDCHDRS,DISP=SHR
//SYSPUNCH DD DSN=WELLSK.TEAMC.OBJ(MEMBER),DISP=SHR
//SYSLIN DD SYSOUT=*
//SYSPRINT DD DSN=WELLSK.TEAMC.ERROR(MEMBER),DISP=SHR
//SYSPRT DD DSN=WELLSK.TEAMC.LISTING(MEMBER),DISP=SHR
//SYSUT1 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10 DD SYSOUT=*
//

```

Figure 52. A JCL fragment for an MVS compile

The first step in converting the JCL fragment is to recognize the intent for each of the data sets and ddnames. For this C/370 compiler example, the SYSIN ddname needs to be associated with the source file, the SYSPUNCH ddname needs to be associated with the object file, and so on.

In each of these cases, the build script must tell the TeamConnection build processor where to put or pick up the parts before and after the execution of the specified program (PGM=EDCCOMP).

Assume that your source files in TeamConnection have the extension .c, your object files have .obj, and your include files .h or .hpp. You allocate a data set to the SYSIN ddname to contain a source file with a .c extension. You specify the DCB, UNIT, DISP, and SPACE attributes to dynamically create this data set every time this build script is invoked. Notice that the attribute SPACE=(TRK,(10,5)) indicates a sequential data set organization.

You specify the output messages that will be returned to TeamConnection by using the TCOUT attribute. This attribute tells the MVS build processor to return the information in the data set associated with the TCEXT=TCOUT attribute.

Note: The STEPLIB is renamed by the MVS build processor to STEPLIBB for data set lookup of the program specified by the PGM parameter on an EXEC statement.

The following MVS build script is the result of converting the JCL fragment by adding the TeamConnection MVS JCL syntax.

```
//COMPILE EXEC PGM=EDCCOMP,
// PARM='LO,SSCOMM,NOSEQ,NOMAR,LIS,FL(I),SO,DECK,&TCPARM',
// REGION=1536K
//STEPLIB DD DSN=SYS1.EDC.SEDCCOMP,DISP=SHR
// DD DSN=SYS1.EDC.SEDCLINK,DISP=SHR
// DD DSN=SYS1.PLI.SIBMLINK,DISP=SHR
//SYMSGSGS DD DSN=SYS1.EDC.SEDCDMSG(EDCMSGE),DISP=SHR
//SYSIN DD TCEXT=(C,CPP),DISP=(NEW,DELETE),
// UNIT=SYSDA,SPACE=(TRK,(10,5)),
// DCB=(RECFM=VB,LRECL=150,BLKSIZE=3200)
//USERLIB DD TCEXT=(H,HPP),DISP=(NEW,DELETE),
// UNIT=VIO,SPACE=(TRK,(5,10,10)),
// DCB=(RECFM=VB,LRECL=50,BLKSIZE=3200)
//SYSLIB DD DSN=SYS1.EDC.SEDCHDRS,DISP=SHR
//SYSPUNCH DD TCEXT=OBJ,DISP=(NEW,DELETE),
// UNIT=VIO,SPACE=(TRK,(10,5)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIN DD SYSOUT=*
//SYSPRINT DD TCEXT=TCOUT,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),UNIT=VIO,
// DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSPRT DD TCEXT=TCOUT,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),UNIT=VIO,
// DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT1 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9 DD UNIT=VIO,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10 DD SYSOUT=*
//
```

Figure 53. A JCL fragment converted to a build script

Example of a build script for a COBOL compile

TeamConnection provides a sample build script program for compiling MVS COBOL programs. This sample is called `fhbcobm.jcl`. It invokes a JCL procedure called `IGYWC`, which needs to be in the system `PROCLIB` concatenation or in the data set identified by the `TEAMPROC` DD statement in the MVS build processor job. You may need to adjust the default parameters for the system. The following JCL should work with any IBM COBOL/II type of compiler such as the IBM COBOL/II compiler `IGYCRCTL`:

```
/*-----
/* PROGRAM:  cobolcmp.jcl
/* IBM COBOL for MVS
/* Compile Only
/*
/*-----
```

```

//COBOLCMP EXEC PGM=IGYCRCTL,PARM='&TCPARM'
//*
//* INPUT FILES WITH EXTENSION CBL
//*
//SYSIN DD TCEXT=CBL,DISP=(NEW,DELETE),
// SPACE=(32000,(30,10)),UNIT=SYSDA,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//*
//* COPY FILES WITH EXTENSION CPY
//*
//SYSLIB DD TCEXT=CPY,DISP=(NEW,KEEP),
// SPACE=(32000,(30,30,30)),UNIT=SYSDA,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=6160)
//*
//SYSPRINT DD TCEXT=TCOUT,DISP=(NEW,DELETE),
// SPACE=(32000,(30,30)),UNIT=SYSDA,
// DCB=(RECFM=FBA,LRECL=133,BLKSIZE=3990)
//SYSLIN DD TCEXT=OBJ,UNIT=SYSDA,
// DISP=(NEW,DELETE),SPACE=(32000,(30,10)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//*
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//

```

Example of a build script for a link

Because MVS load modules are not easily transferable, TeamConnection provides a sample build script program that reads linkage editor SYSLIN control statements. This script produces a single file that can be returned from MVS and loaded into TeamConnection. You can later extract the file and transport it to MVS, where it can be link edited to produce an executable load module.

The next example shows this sample build script, named fhbtclnk.jcl, which is shipped with the TeamConnection client.

You can use either of the following for an INCLUDE control statement for the FHBTCCLNK program:

- INCLUDE DDNAME(MEMBER)
- INCLUDE DDNAME

This syntax is a subset of the linkage editor INCLUDE card.

If the card is an INCLUDE ddname(MEMBER) control statement, the object code is copied into a sequential data set associated with the SYSMOD ddname. Otherwise, the control card is embedded in the data set associated with the SYSMOD ddname. This data set can be returned as the output from this build script.

```

//FHBTCCLNK EXEC PGM=FHBTCCLNK,
// PARM='SIZE=(768K,192K),LIST,MAP,AMODE(31),RMODE(24),LET,XREF'
//STEPLIB DD DSN=userid.teamc.LOADLIB,DISP=SHR
//SYSMOD DD TCEXT=LOAD,DISP=(NEW,DELETE),
// SPACE=(32000,(30,10)),UNIT=VIO,
// DCB=(RECFM=U,LRE10CL=80,BLKSIZE=3200)
//OBJ DD TCEXT=(OBJ,PRE),DISP=(NEW,DELETE),
// UNIT=VIO,SPACE=(32000,(30,10,10)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)

```

```
//SYSPRINT DD TCEXT=TCOUT,DISP=(NEW,DELETE),
// UNIT=VIO,SPACE=(TRK,(30,10)),
// DCB=(RECFM=FB,LRECL=121,BLKSIZE=1210)
//SYSLIN DD *
INCLUDE OBJ(&TCINPUT)
ENTRY CEESTART
//
```

TCEXT attributes have been added to the following DD statements:

Data set

Purpose

SYSMOD

Return the output to check in to TeamConnection

OBJ Receive the object files transported to MVS from TeamConnection

SYSPRINT

Return any FHBTCLNK messages to TeamConnection

In the SYSLIN data stream, the statement INCLUDE OBJ(&TCINPUT) will be duplicated for all of the inputs to this build. The &TCINPUT variable will be replaced with the base name of the input without the extension.

To use the output of this build script as an MVS executable, do the following:

1. Extract the output from TeamConnection.
2. Transfer the output as a binary file from your workstation to MVS (for example, using FTP).
3. Link edit this output into a load module. Possible SYSLIN control statements for the link step include the following:

```
//SYSLIN DD *
INCLUDE OBJECT(OUTPUT)
NAME module(R)
//
```

The output specified in INCLUDE OBJECT(OUTPUT) contains embedded control statements specified from the build script FHBTCLNK. The linkage editor recognizes these embedded statements and produces an executable load module from the output file. The NAME control statement cannot be embedded in the output data set.

Chapter 11. Working with parsers

This chapter describes how to create a parser. It assumes that you have read "Chapter 7. Basic build concepts" on page 93.

Consider the task of defining and maintaining a build tree. One of the more time-consuming, and error-prone, portions of this task is defining the dependencies that one TeamConnection part has on others.

For example, if hello.c includes hello.h, you need to define hello.h as a dependency of hello.c in the build tree. That sounds simple enough, but imagine a real application in which there are hundreds of dependencies and the dependencies have dependencies. Defining such a tree becomes very difficult; maintaining it, even more so.

To solve this problem and automate some of the work of defining and maintaining a build tree, you can instead use a parser object. The task of a parser is to inspect source code to determine dependencies. In the previous example, a parser can inspect hello.c, recognize that it has a dependency on hello.h, and create that dependency in the TeamConnection build tree.

Because parsers are language-dependent, you probably need a different parser for each language you use in a particular release. For example, you might have both a COBOL parser and a C parser in a release. Many parts in the release can use the same parser.

Usually a TeamConnection administrator defines parsers, but anyone with the proper authority can do so. For more information about the required authority, see Appendix H. Authority and notification for TeamConnection actions.

Creating a parser

As with most other TeamConnection operations, there are two ways you can create a parser: using the graphical user interface (GUI) or the command line interface.

To create a parser using the GUI:

1. From the Actions pull-down menu on the Tasks window, select **Parsers → Create**.
2. On the Create Parser window, type the requested information.

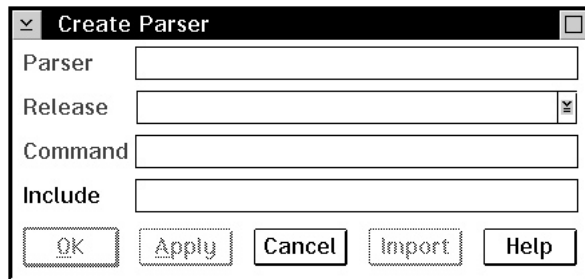


Figure 54. Create Parser window

From a command line, type the parser -create command and press Enter. The complete command syntax looks like the following:

```
teamc parser -create name -command name -release name -family name
               [-include paths]
               [-become user_name] [-verbose]
```

No matter which way you create a parser, you must specify a number of attributes for it. Together with the contents of the parser command file, the following attributes define how a parser determines the dependencies for a TeamConnection part.

Parser

The name of the parser must be unique within a release. It can be anything you want, but for best results, establish and follow a meaningful naming convention. An example of a parser name is `c_parser`.

Release

This is the name of the release that contains the parser. Parsers are release-specific objects. They are not versioned within a release; therefore you can have only one version of a parser at any time in a release.

To use the parser from a previous release, you can link to a part that uses it in that release. This action copies the parser to the new release. Otherwise, you must create the parser again in the new release.

Command

This is the name of the command file that the parser invokes to determine the dependencies. It can be any file name that exists in the execution path of the family server at the time a build is performed. TeamConnection runs the command as a subprocess on the machine where the build processor is located.

The task of the command file is to inspect the source file and return a list of dependencies. The syntax for invoking this command is discussed in “Writing a parser command file” on page 139.

Include

This is a concatenated set of paths that define where the parser looks for parts when processing the set of dependencies returned from the command file. These dependencies come in two types:

- A dependency in which the file is stored in the TeamConnection database. For example, `hello.c` includes `hello.h`, and both files are stored in the TeamConnection database. During a build, these dependencies must be extracted to a path accessible by the build processor. Because a build extracts parts from TeamConnection, anyone requesting a build needs to have PartExtract authority to all parts involved in the build.

- A dependency on a file that is not stored in the TeamConnection database. An example of such a dependency is `stdio.h`, which is typically stored in a compiler's include path and not in the TeamConnection database.

Each path named in **Include** is queried in the TeamConnection database to see if it contains a part matching the dependency name. For example, suppose you define a parser named `c_parser` with an include path as follows:

```
src\include;src\package;.;src\comm\include;
```

One of the parts to which this parser is attached, `src\example.cpp`, contains the statement `#include "example.hpp"`. Thus the command file for `c_parser` reports `example.hpp` as a dependency of `src\example.cpp`. The parser concatenates each path listed in `c_parser`'s include path with the name `example.hpp`, then inspects the contents of the TeamConnection database to see if a part with that name exists. So the TeamConnection database is queried first to find `src\include\example.hpp`, then `src\package\example.hpp`.

The period (.) in the include path tells TeamConnection to concatenate the path of the part to which the file is a dependent with the dependent's file name. In this example, that means the TeamConnection database is queried to find a part named `src\example.hpp`.

Putting a parser to work

For an application to use a parser, the parser must be attached to the TeamConnection parts that it will check for dependencies. Unlike a builder, a parser is attached to the *input* part rather than the output.

To attach a parser to a part, do one of the following:

- From the GUI, select **Parts** → **Modify** → **Properties** from the Actions menu of the TeamConnection Tasks window. On the Modify Part Properties window, type the name of the parser.

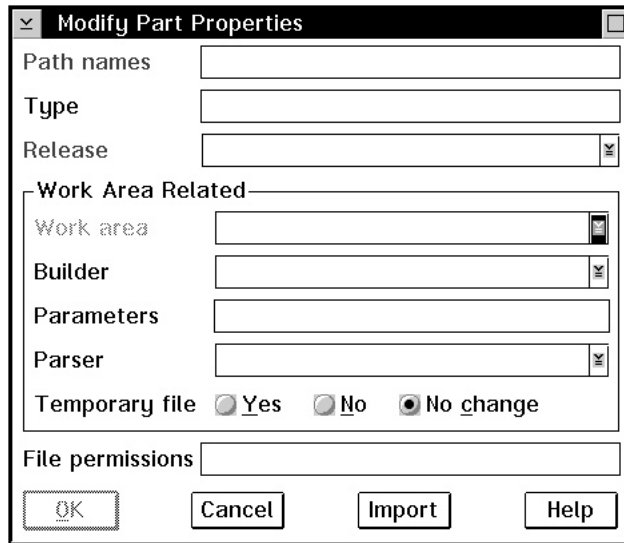


Figure 55. Modify Part Properties window

- At a command prompt, type the following and press Enter:

```
teamc part -modify part -parser name -release name
        -family name
```

The complete syntax for this command is described in the *Commands Reference*

You can also attach a parser to a part when the part is created.

After you attach a parser to a part, it is ready for action. The next time the part is used in a build, the parser will invoke its command file, which will report back a list of dependencies.

Using a parser does not keep you from defining dependencies manually by using the GUI or the part -connect command. If you explicitly define a dependency in this way, the dependency is not deleted unless you delete it, regardless of whether the parser would recognize it as such.

For more information about attaching parsers to the build tree, refer to “Creating the build tree for the application” on page 143.

Removing a parser from a part

If you no longer want to use a parser to determine dependencies for a part, do one of the following:

- From the GUI, select **Parts** → **Modify** → **Properties** from the Actions menu of the TeamConnection Tasks window. On the Modify Part Properties window, type null in the **Parser** field.

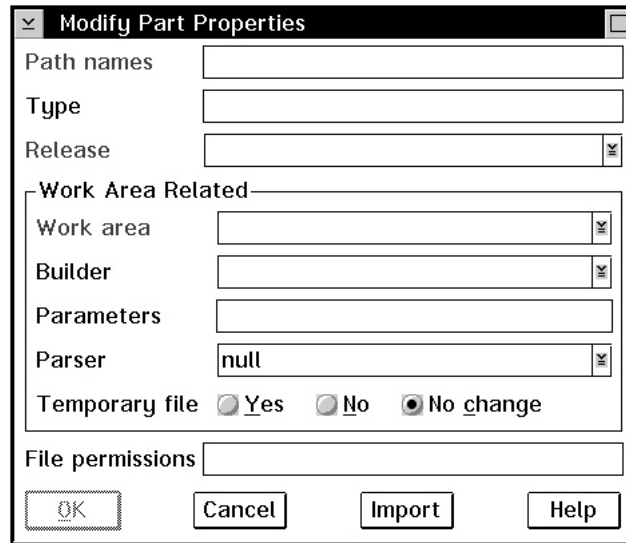


Figure 56. Modify Part Properties window

- From a command line, type the following:

```
teamc part -modify name -parser null
          -release name -family name
```

Writing a parser command file

A parser command file accepts two parameters as input:

- *source file*—the name of the file that contains the source to be parsed.
- *dependency list file*—the name of a file into which the names of the dependent files should be written, one per line. For example, the contents of the file might look like this:

```
hello.h
stdio.h
```

Both the source file and the dependency list file are created by the TeamConnection family server. They are erased after the parse is complete.

To write a command file, write a program, in any language, that does the following:

1. Reads the source file
2. Determines which other files are used by it
3. Writes out the list of such files into the dependency list file

For example, for a C source file, the program could report a list of all the files included by the source file (using `#include` statements). For a COBOL program, `COPY` statements would be the cue. TeamConnection ships a sample of a command file named `fhbopars.cmd`. It is written in REXX.

Chapter 12. Building an application: an example

This chapter uses an extended example to describe in more detail how each of the components of the build function work together. This example walks through the control flow for a sample application, explaining what happens at each step.

These are the tasks involved in building our sample application, msgcat.exe:

Task	Page
Starting build processors and build agents	142
Creating builders and parsers	143
Creating the application build tree	143
Starting the build	147
Monitoring the build	152
Building in spite of errors	153
Forcing a build of all parts	153
Finding out which parts will be built	154
Canceling a build	154

We will use a simple example build tree that looks like the following:

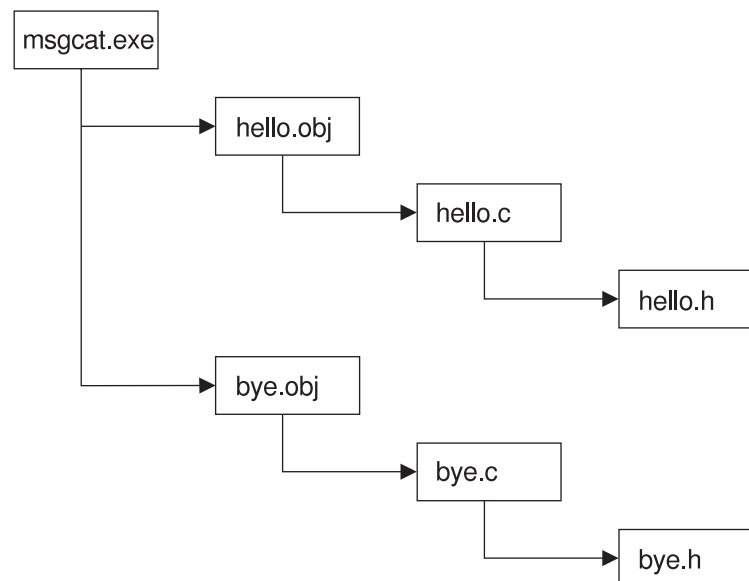


Figure 57. Sample build tree

For more examples of build trees, see “More sample build trees” on page 155.

In terms of the build object model, the objects that make up this tree look like this:

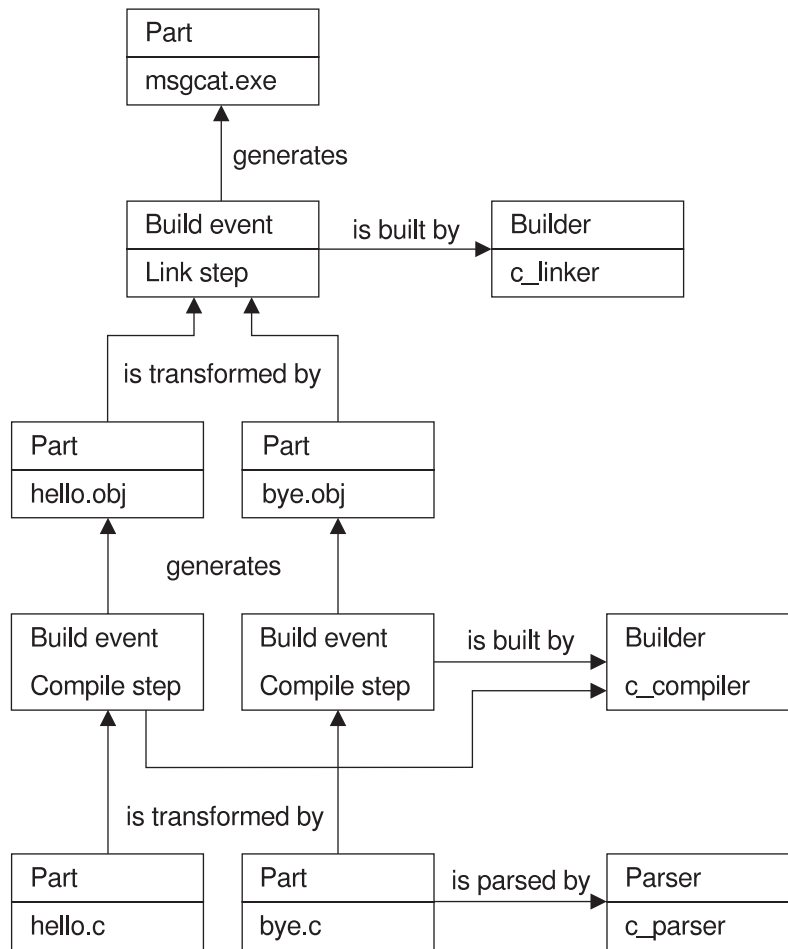


Figure 58. Sample build object model for msgcat.exe

Starting the build processors and build agents

The software development team in our example is building large applications using a family named testfam, so they set TC_FAMILY to testfam. They plan to spread the work across several build processors, taking advantage of TeamConnection's ability to perform multiple build events simultaneously.

Mark, the build administrator, has installed a number of build processors and build agents on the team's machines, for building OS/2 and MVS applications. As he starts them (in pairs), he groups the build agents into *pools*, according to the work he expects to use them for.

Mark plans for the following pools:

mvs For MVS builds

pool1 For normal OS/2 builds

pool2 For fast, high-priority OS/2 builds

Each pool is formed as Mark starts build agents and assigns them to it. Among the build processor-build agent pairs that he starts are the following:


```
teamproc -s bldsock2
teamagnt -s bldsock2 -p pool1 -e os2
```

The first command starts the build processor at TCP/IP address 0705. The second command starts the matching build agent at TCP/IP address 0705. The other parameters specify the following:

- p** The agent is assigned to the pool named pool1.
- e** The environment is os2.

Use the teamproc and teamagnt commands to start the build processor and agent when the family server has already been started. To start the family server along with the build processor and build agent, you can use the teamcd command.

Creating builders and parsers

For the parts of the application that are written in C language, Mark creates the following:

- A builder named c_compiler, to do the compiles
- A builder named c_linker, to do the links
- A parser named c_parser, to check for dependencies

For both builders Mark specifies os2 as the **Environment**, the same as that of the build agent started earlier at TCP/IP address 0705. Build events that use these builders (c_compiler and c_linker) can take place on this build agent-build processor pair.

After he creates the builders and parsers for the applications, Mark spreads the following information to the programmers who will be using them:

- The names of the build pools
- The names and purposes of the builders and parsers

Creating the build tree for the application

At this point, Greg begins defining the build tree for his portion of the application, as shown in Figure 57 on page 141. He has already created the files hello.c, hello.h, bye.c, and bye.h in the TeamConnection database. Now he does the following:

1. Creates a place-holder part for the output of the link step. This file, msgcat.exe, is the target for the entire build, the output of linking hello.obj and bye.obj using the builder c_linker, and the parent of hello.obj and bye.obj. Because the file has no contents initially, he selects **No source** (or specifies -empty on the command line), to identify it as a place holder.

Using the GUI, he can create this file by selecting **Create** from the **Actions** → **Parts** menu of the Tasks window, and completing the fields as shown in the following illustration:

Figure 59. Create Parts window

Using the command-line interface, he can create the part by issuing the following command:

```
teamc part -create msgcat.exe -builder c_linker -binary -empty
-release 9503 -workarea 223 -component comp1
```

2. Creates two place-holder parts for the output of compiling the .c files. These parts are the output of the compile step; c_compiler, the builder that manages that step, is attached to both of them. He indicates that they are input to their parent file, msgcat.exe.

Using the GUI, he can create these files by selecting **Create** from the **Actions** → **Parts** menu of the Tasks window, and completing the fields as shown in the following illustration:

Create Parts

Part names:

Release:

Work area:

Component:

File type: ☐ Text ☒ Binary ☐ None

Source: ☐ Same ☒ No source ☐ Copy from

Source file:

Source directory:

Remarks:

Parent:

Parent type:

Relation to parent: ☒ Input ☐ Output ☐ Dependent

Builder:

Parameters:

Parser:

☐ Temporary file

Figure 60. Create Parts window

Using the command-line interface, he can create the parts by issuing the following command:

```
teamc part -create hello.obj bye.obj -builder c_compiler -binary -empty
-release 9503 -workarea 223 -component comp1 -parent msgcat.exe -input
```

3. Attaches the parser c_parser to the .c files.

Using the GUI, he can attach the parser to these files by selecting **Modify → Properties** from the **Actions → Parts** menu of the Tasks window, and completing the fields as shown in the following illustration:

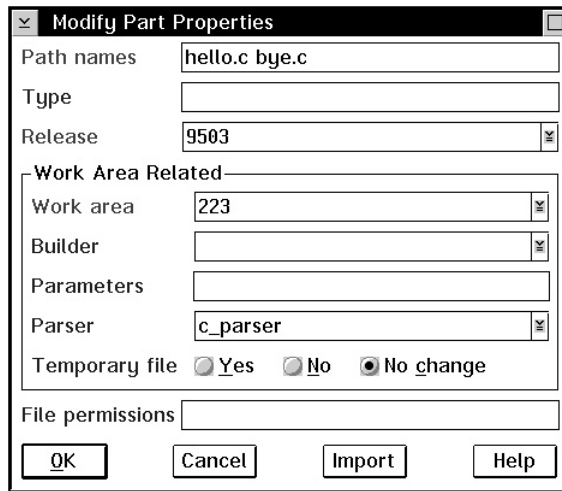


Figure 61. Modify Part Properties window

Using the command-line interface, he can attach the parser to these parts by issuing the following command:

```
teamc part -modify hello.c bye.c -parser c_parser -release 9503
-workarea 223
```

Remember, the parser is attached to an *input* file.

4. Connects the .c files into the build tree.

Using the GUI, he can connect these files by selecting **Connect** from the **Actions** → **Parts** menu of the Tasks window, and completing the fields as shown in the following illustration. He needs to execute this function twice: once to connect hello.c to hello.obj and once to connect bye.c to bye.obj.

Using the command-line interface, he can connect these parts by issuing the

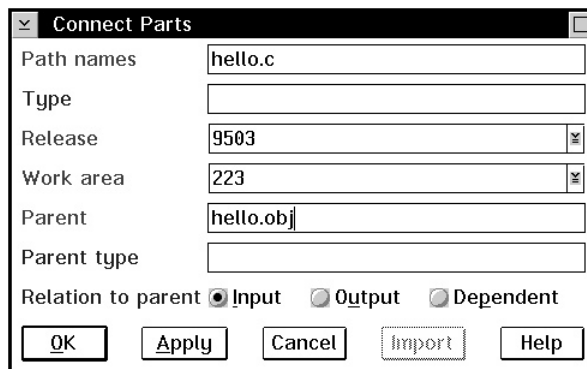


Figure 62. Connect Parts window

following commands:

```
teamc part -connect hello.c -parent hello.obj -input -release 9503
-workarea 223
```

```
teamc part -connect bye.c -parent bye.obj -input -release 9503
-workarea 223
```

- Now, Greg can see the build tree in the GUI. From the **Objects** pull-down menu on the Tasks window, he selects **Parts** → **View build tree**. The BuildView Filter window is displayed; from here he can bring up the build tree.

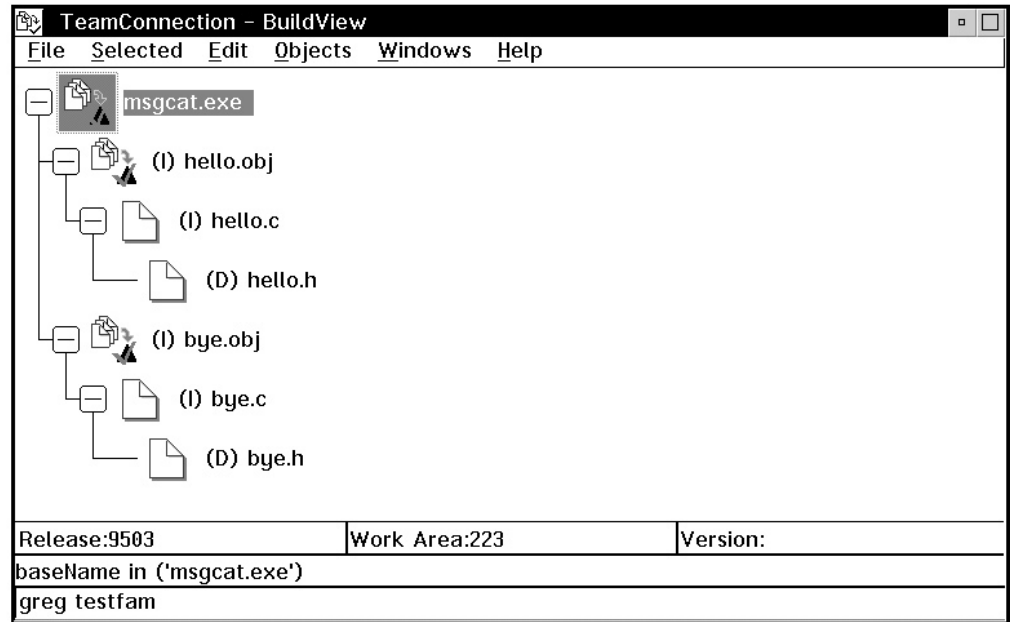


Figure 63. The build tree display

Starting the build on the client

After much hard work on his source code, Greg is ready to start building his application.

Using the GUI, he can start the build by selecting **build** from the **Actions** → **Parts** menu of the Tasks window, and completing the fields as shown in the following illustration:

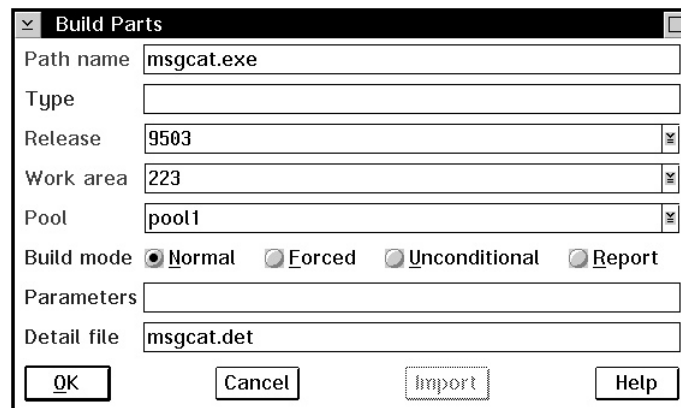


Figure 64. Build Parts window

Using the command-line interface, he can start the build by issuing the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea 223  
-pool pool1 -normal -detail msgcat.det
```

This command specifies the following:

Build target

The name of the part at the top of the build tree, msgcat.exe, which is the final output of this build. TeamConnection uses the build target to determine the scope of the build.

Work area

The version of the TeamConnection parts and build tree to be used when performing this build. This version is completely specified by naming the family, release, and work area: in this case, -release 9503 -workarea 223. The output of the build is placed in this work area.

Build pool

The set of build agents that should be used to process the build request, as defined when the build agents are started. The pool pool1 includes the build agent started in “Starting the build processors and build agents” on page 142 .

Build mode

How the build takes place. Possible values for this build option include the following:

Normal

Builds only the parts that are out-of-date. Processing stops after the first error is returned.

Force Builds all parts, even if they are not out-of-date. Processing stops after the first error is returned.

Unconditional

Builds only parts that are out-of-date but continues processing even if errors are returned. Note that outputs are not rebuilt for inputs that have failed.

Report

Gives a preview of what would be built if you invoked a build. The report identifies what steps would occur without any translations taking place.

In our example, Greg specifies -normal, which is the default. In this mode, only the parts that are stale with respect to their inputs are rebuilt. In other words, only the minimum amount of work to bring everything up-to-date is performed. “Determining the build scope” on page 149, gives more information about how TeamConnection determines which parts to build. “Running a build in spite of errors” on page 153 and following sections provide examples of using the other build modes.

In normal mode, the build is halted if an error is found. Any remaining build events in the build scope are canceled, but any build events already performed are not undone.

Detail file name

The name of an output file in which TeamConnection stores the collected stdout and stderr of the build scripts.

When Greg starts the build, the information from the command is passed to the testfam family server over TCP/IP. At this point, Greg's TeamConnection client waits to receive confirmation from the family server that the build request was received and is being processed.

Note: Only one build is allowed in a work area at one time (though the build events that make up the build might be distributed to different build agents on a number of machines). So if Greg is sharing work area 223 with Barbara, she cannot issue a part -build command in that same work area until Greg's build is complete.

TeamConnection handles the next parts of the build process automatically.

Determining the build scope

The next step is for the build function to determine the build scope. The build scope is a set of all the build events that need to be done to bring a particular build target up-to-date with respect to its inputs. When a part is built, TeamConnection marks both the input and output with the time of the build; when a later build takes place, the output part is rebuilt if its build time is different from the input part's.

In our example, the TeamConnection family server looks at the build tree for msgcat.exe. If the build time on an input part is different than the build time on its output, then the output part must be rebuilt. (Remember that the output of one build event can also be the input to the next.)

In our example, suppose the update times look like this.

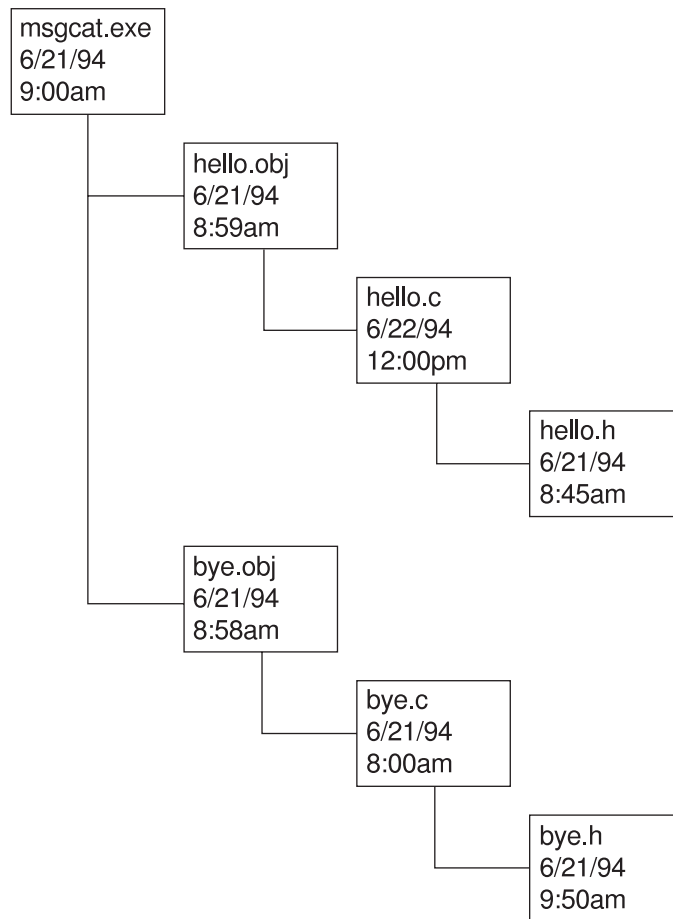


Figure 65. Build tree showing build times

In this case, Greg's build command for msgcat.exe causes the creation of a build scope including the following build events:

- A build event for creating hello.obj—that is, a compile of hello.c
- A build event for creating msgcat.exe—that is, a link of hello.obj and bye.obj

The build event for creating bye.obj is not included in the build scope, because bye.obj is up-to-date with respect to its input bye.c. The build time on bye.obj is not equal to that of its input, bye.c.

Greg wants to make sure that bye.obj is rebuilt, so he *touches* bye.c or bye.obj, using the TeamConnection command part -touch. This command marks the part as out-of-date in the work area so that it is included in the next build. Now when he starts a build of msgcat.exe, the build scope includes the build event to create bye.obj.

The resulting build scope contains three build events: two compiles and a link. Notice that the two compiles can be performed in parallel, for faster performance.

At this point any parsers associated with parts in the scope of this build are invoked. The information returned from the parsers is used to analyze dependencies.

TeamConnection can determine if a part has already been built in another context using identical inputs and time stamps. Suppose, for example, a part is build in work area 913 and that work area has been integrated. Next, a new work area, 914, is created in the same release and the same part is requested to be built in that work area. Since the time stamps in work area 913 and the release are the same as in work area 914, the build does not have to execute and the build outputs are copied to work area 914.

Adding the job to the job queue

The next step is for TeamConnection to add the build scope to the testfam family's job queue. This step ensures that the build events in it are picked up and processed by any available build agents. After adding the build scope to the job queue, the family server gives Greg's client confirmation that the build is being processed. It also reports some statistics about the build, such as the number of build events. The client reports this data to Greg; it then waits to receive and report the outcome of each build event as it happens.

Picking up the work orders

Let's look at what has been happening while the preceding steps have been going on. Each of the build agents that Mark started earlier is actively polling the family's job queue to see if there is any work they can perform.

Each build agent looks at the top-most build scope in its job queue. If it contains a build event that this build agent can perform (that is, one for the pool and environment it was started for), the build agent takes the event and starts to process it. If it does not find a build event it can process, it waits awhile and then tries again.

Suppose that Mark started four build processors and their corresponding build agents, two in pool1 and two in pool2. That means that, as soon as TeamConnection determines the build scope for msgcat.exe, each of the build agents in pool1 finds a build event it can perform, one for compiling hello.c and one for compiling bye.c.

Putting the build processors to work

Each build agent that finds an event it can perform sends a description of the event to its corresponding build processor over TCP/IP. It then sends to and receives from the TeamConnection database any parts data the processor needs.

In our example, each of the two build agents in pool1 sends a description of the compile event to its connected processor and then waits to answer any requests from the processor for parts data.

Putting the build scripts to work

At this point, the build processor looks at the description of the event it has been asked to perform, then checks its cache for each part and the build script it needs. If it does not find them there or if the cached parts are out of date, it asks the build agent.

It then invokes the build script, passing it the names of the input and output parts and the parameters specified on the builder. The parts created by the build script and the return code generated by it are sent back to the waiting build agent. The build agent then updates the contents of the TeamConnection database.

In our example, each of the two build processors receives a compile event to perform. Each asks its corresponding build agent to extract the .c source files it needs from the TeamConnection database and the contents of the build script for the c_compiler builder. It then runs the build script.

The results (the .obj files and the return code) are sent back to the build agents. After updating the TeamConnection database, the build agents re-enter their polling loop to see if any more build events await their attention.

Because the compile steps are performed in parallel, Greg can build this application a little more quickly than if they had happened in serial mode. In this simple example, the difference is hardly noticeable; but in a large build of hundreds of parts, with multiple build processors available on a local area network, the performance improvement can be enormous.

Finishing the job and reporting the results to the user

The processing described by the previous two steps is repeated until there are no more build events that comprise the build scope. The results of the build are displayed in the Build Progress window or in stdout. At this point the build is complete.

To complete our example, the previous two steps are repeated to complete the link step, using either of the two build agents in pool1. Greg now can extract the resulting executable from TeamConnection, using the part -extract msgcat.exe command, and run it.

Monitoring the progress of a build

During the course of a build, you can monitor its progress in several ways:

- If the build was started from the command line, by issuing the report -view partview command against the work area in which you are building. From this report, you can determine the states of the parts. Use the part -viewmsg command to see the build messages issued because of a failed build. For complete syntax of these commands, refer to the *Commands Reference*
- If the build was started from the GUI, in the Build Progress window. You can find the same information by looking at stdout.

Greg can see how the build is progressing by checking the Build Progress window. For example, he might see these messages:

```
6021-301 Invoking Parser c_parser for hello.c
6021-303 A successful parse resulted from using the parser c_parser. The
        parser return code is 0
6021-301 Invoking Parser c_parser for bye.c
6021-303 A successful parse resulted from using the parser c_parser. The
        parser return code is 0
6021-700 Number of distinct build events for this build: 3.
Build of 'hello.obj' started at '15:33:47 1995-08-10'
via a build agent on the host 'OCTOFVT'.
Build of 'hello.obj' successfully completed at '15:34:45 1995-08-10'.
Completed Jobs: 1
```

```
Remaining Jobs: 2
Build of 'bye.obj' started at '15:34:49 1995-08-10'
via a build agent on the host 'OCTOFVT'.
Build of 'bye.obj' successfully completed at '15:35:22 1995-08-10'.
Completed Jobs: 2
Remaining Jobs: 1
Build of 'msgcat.exe' started at '15:35:26 1995-08-10'
via a build agent on the host 'OCTOFVT'.
Build of 'msgcat.exe' successfully completed at '15:35:56 1995-08-10'.
Completed Jobs: 3
Remaining Jobs: 0
Processing Completed for 'msgcat.exe'.
```

To see the commands that TeamConnection issued during the build, he can look at the detail file that he specified in the part -build command.

Running a build in spite of errors

If you find that a build is stopping because of errors, you can check the build detail file or the Build Progress window for the cause. If the error is minor, you might decide to run the build despite the errors—for example, when you are debugging. To do this, specify that you want the build to complete unconditionally.

In our example, when Greg builds msgcat.exe for the first time, he wants to find and correct any errors that occur during the build, so he uses the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea 223 -unconditional
```

As in normal mode, only the parts that are stale with respect to their inputs are rebuilt; only the minimum amount of work to bring everything up-to-date is performed.

However, even if an error is found, the build continues if possible. As with normal mode, if the build is halted, any build events remaining in the build scope are canceled. Any build events already performed are not undone.

Building all parts, regardless of build times

To make sure that **all** parts in the build tree get built, whether or not they are stale, you specify the -force parameter on the part -build command.

In this mode, all parts that are descendants of the build target are rebuilt, no matter what.

In our example, Greg can force TeamConnection to build all parts in the msgcat.exe build tree using the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea 223 -force -pool pool1
```

If an error occurs, the build is halted. Any remaining build events in the build scope are canceled, but any build events already performed are not undone.

Finding out which parts will be built

Before running a build of a large application, you might want to find out exactly which parts will be built. If you specify that you want to run in report mode, TeamConnection determines what will be built in a normal build and produces a report showing the results.

If Greg really wants to see which parts of msgcat.exe will be built before he runs the actual build, he can issue the following command:

```
teamc part -build msgcat.exe -release 9503 -workarea d410 -report -pool pool1
```

He sees the following report:

```
6021-301 Invoking Parser c_parser for hello.c
6021-303 A successful parse resulted from using the parser c_parser. The
        parser return code is 0
6021-301 Invoking Parser c_parser for bye.c
6021-303 A successful parse resulted from using the parser c_parser. The
        parser return code is 0
6021-700 Number of distinct build events for this build: 3.
6021-407 The builder c_compiler will be invoked.
6021-406 The builder parameters consist of:
        command:  compC.cmd
        input:    hello.c
        output:   hello.obj
        dependent: hello.h
6021-407 The builder c_compiler will be invoked.
6021-406 The builder parameters consist of:
        command:  compC.cmd
        input:    bye.c
        output:   bye.obj
        dependent: bye.h
6021-407 The builder c_linker will be invoked.
6021-406 The builder parameters consist of:
        command:  linkC.cmd
        input:    hello.obj bye.obj
        output:   msgcat.exe
        dependent:
```

The report shows that bye.obj and msgcat.exe must be rebuilt.

Canceling a build

To cancel a build that is in progress, do one of the following:

- If the build was started from the GUI, on the Build Progress window select the **Cancel Build** push button.
- If the build was started from the command line, type the following command and press Enter:

```
teamc part -build name -cancel
```

Where *name* is the part that you are building. Be sure to specify the same part name that you specified when starting the build, rather than a part that is lower in the build tree.

This command stops any further build events being performed for that build scope. Any build events already performed for that build are not undone.

For example, if Greg cancels the build of msgcat.exe when the compile steps have been completed, then the link step is not performed. However, the newly compiled hello.obj and bye.obj are left in the database, with their build times updated.

3.1

You cannot use the **Stop Build** button on the Build Progress window to cancel a build in progress in the Windows environment. Always use the part `-build -cancel` command instead.

More sample build trees

The msgcat.exe example is just one possible build tree. Here are some others.

Defining multiple outputs from a single build event

Figure 66 shows part of the build tree for robot.dll:

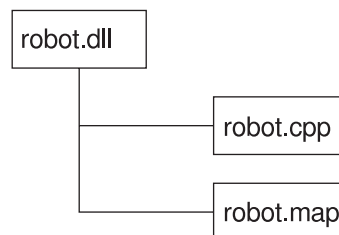


Figure 66. The build tree for robot.dll

Because the build tree shows the relationships between parts hierarchically, robot.map is a child of robot.dll, even though it is actually built from the same input part, robot.cpp. But robot.map is defined as an *output* of robot.dll. Here are the commands to set up this relationship.

First come the commands to create the parts:

```
teamc part -create robot.dll -builder dll_builder -binary -empty  
-release 9503 -component robot
```

```
teamc part -create robot.cpp -release 9503 -component robot
```

```
teamc part -create robot.map -builder dll_builder -binary -empty  
-release 9503 -component robot
```

Next are the commands to connect the parts into the build tree:

```
teamc part -connect robot.cpp -parent robot.dll -input -release 9503
```

```
teamc part -connect robot.map -parent robot.dll -output -release 9503
```

You might use this command to start the build:

```
teamc part -build robot.dll -workarea 915 -release 9503
```

The output of this build would be both `robot.dll` and `robot.map`. Any parameters specified in the `teamc` part `-build robot.dll` command would also apply to the build of `robot.map`.

Synchronizing the build of unrelated parts

An entire application can require multiple separate builds. For example, in the `robot` application, there might be one build to create the `.dll` parts, another to create the `.exe` parts, and so on. To ensure that the entire application gets built together, you can create a part that acts as a collector, with the `.dll` and `.exe` parts as input to it.

For example, Tim creates this build tree for the `robot` application:

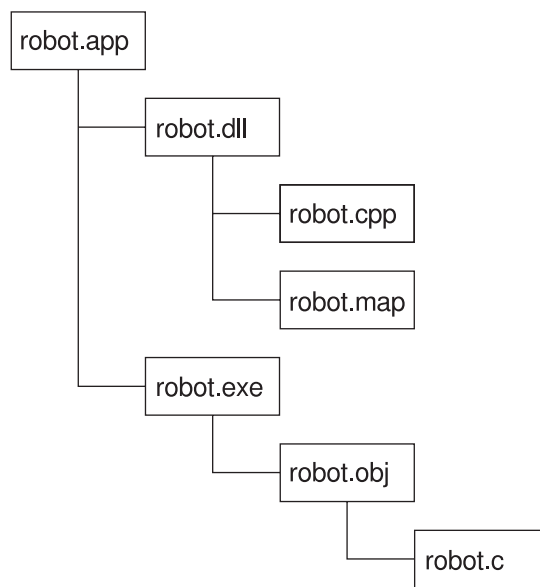


Figure 67. The build tree for `robot.app`

Assuming he already has the build trees for `robot.dll` and `robot.exe` set up, here is how he sets up the collector part:

1. He creates a null builder with no contents:

```
teamc builder -create nullBuilder -script null -none -environment os2  
-condition == -value 0
```

2. He creates the collector part:

```
teamc part -create robot.app -builder nullBuilder -none -release 9503  
-component robot
```

The `-none` flag identifies this as a part that will never have any contents.

3. Tim connects the other parts to the collector:

```
teamc part -connect robot.dll robot.exe -parent robot.app -input  
-release 9503
```

When Tim builds `robot.app`, the result is a build of both `robot.dll` and `robot.exe`.

Part 4. Using TeamConnection to package products

Chapter 13. Using TeamConnection to package a product	159
Setting up your build tree for packaging	160
Setting up a build tree for the gather tool	160
Setting up a build tree for the NVBridge tool	162
Setting up a build tree for other distribution tools	163
 Chapter 14. Using the Gather tool	165
Using the teamcpak command for the Gather tool	166
Command line flags	166
Examples of the teamcpak gather command	167
Writing a package file for the Gather tool	168
Syntax rules for a Gather package file	168
Keywords for a Gather package file	169
Using exit keywords in the DATA clause	171
Using exit keywords in the RULE clause	172
Using exit keywords: an example	172
 Chapter 15. Using the NVBridge tool	173
Using the teamcpak command for NVBridge	174
Command line flags	175
Examples of the teamcpak nvbridge command	176
Writing a package file for NVBridge	176
Syntax rules for an NVBridge package file	177
Keywords for an NVBridge package file	177
Problem determination for NVBridge	185
NVBridge utilities	186
FHPSTAT	187
Syntax	187
Return Codes	187
Example	187
FHPOBDEL	187
Syntax	187
Return Codes	187
Example	187
FHPOBMON	187
Syntax	188
Return Codes	188
Example	188
FHPOBDIF	188
Syntax	189
Return Codes	189
Example	189
FHPISCAT	189
Syntax	189
Return Codes	189
Example	189
FHPICAT	189
Syntax	190
Return Codes	190
Example	190
FHPUCAT	190
Syntax	190
Return Codes	190

Example	191
FHPMCAT	191
Syntax	191
Return Codes	191
Example	191
FHPVERIF	191
Syntax	191
Return Codes	192
Example	192
FHPRQPUR	192
Syntax	192
Return Codes	192
Example	192
FHPRQMON	192
Syntax	193
Return Codes	193
Example	193
FHPTRVER.	193
Syntax	193
Return Codes	193
Example	194
FHPTRPUR	194
Syntax	194
Return Codes	194
Example	194
Chapter 16. Using the Tivoli/Courier packaging tool.	195
Using the teamcpak command with Tivoli/Courier	195
Command line flags.	196
Example of the teamcpak softdist command	196
Writing a package file for Tivoli/Courier.	197
Syntax rules for a Tivoli/Courier package file	197
Keywords for a Tivoli/Courier package file	197
Problem determination for the Tivoli/Courier tool	200
Sample package file	200

This section describes how to use the TeamConnection packaging function, which helps you automate the packaging and distribution of your applications. This section is written for the person in your organization who is responsible for software distribution.

Chapter 13. Using TeamConnection to package a product

After you have built an application to your satisfaction, it is time to distribute it to users. This chapter describes how you can use TeamConnection to help automate the packaging and distribution steps.

TeamConnection provides the following:

- Two electronic software distribution tools:
 - **Gather**, which moves an application's parts into a single directory in preparation for distribution.



The Gather utility is available on OS/2 and Windows NT platforms.

- **NVBridge**, a bridge tool that automates the installation and distribution of software or data using IBM NetView Distribution Manager/2 as the distribution vehicle.



The NVBridge utility is available only on OS/2.

- Two sample build scripts for connecting the Gather and NVBridge tools with TeamConnection user-defined builders.
- A set of mini-utilities that can be used to develop customized electronic software distribution solutions.

To use TeamConnection in packaging a product, you might do any of the following tasks:

Task	Page
Setting up your application's build tree for packaging	160
Using the teamcpak gather command	166
Writing a package file for the gather tool	168
Using the teamcpak nvbridge command	174
Writing a package file for the NVBridge tool	176

Setting up your build tree for packaging

When TeamConnection builds an application, the application's build tree identifies the parts to be built and the tools to use in building it. Similarly, when you use TeamConnection for packaging the application, the build tree can define the parts to be packaged and the tools to do it.

The output of a packaging step might be any of the following:

- The application's parts gathered into a new directory structure
- The distribution of the application using NVBridge
- The distribution of the application using some other distribution tool

Setting up a build tree for the gather tool

To gather the parts of your application into a single directory for distribution, you create an output part whose builder calls the gather tool, and you make this output part the top level of the build tree.

For example, for the robot control application, `robot.app`, the build tree might look in part like this:

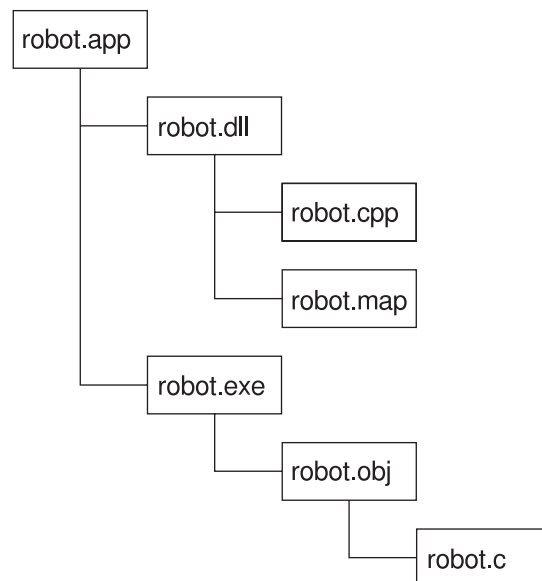


Figure 68. Part of the build tree for robot.app

After the application is built, the programming team needs to get it to the test team. They could extract the application, but doing a simple extract would preserve the existing structure, with parts contained in directories according to their application component. A better structure might be to place all of the .dll files in one directory, all of the .exe files in another, and so on. To move the parts into this structure, the test team does a different kind of build, using the gather tool.

To make this happen, Annmarie does the following:

1. She creates the top-level part for the new build tree. The name of this part is the same as the directory in which the gathered parts are to be placed. In this example, e:\robot is the output file from the gather step. Annmarie uses the following command:

```
teamc part -create e:\robot -none -builder gather1 -family octo
-release 9503 -workarea 410
```

2. She writes a package file that contains instructions for the gather tool and creates this file as a TeamConnection part:

```
teamc part -create robot.pkf -text -parent e:\robot -input -family octo
-release 9503 -workarea 410
```

For more information, see “Writing a package file for the Gather tool” on page 168 .

3. She creates a builder, gather1, that calls the gather tool:

```
teamc builder -create gather1 -script gather.cmd
-parameters "-o -x" -release 9503
-environment os2 -condition == -value 0 -family octo
```

gather.cmd is a sample build script that is shipped with TeamConnection. It specifies the teamcpak gather command.

4. She connects robot.exe and robot.dll to e:\robot as inputs:

```
teamc part -connect robot.exe -parent e:\robot -family octo
-release 9503 -workarea 410
```

```
teamc part -connect robot.dll -parent e:\robot -family octo
-release 9503 -workarea 410
```

5. She also connects a readme file for the application:

```
teamc part -connect read.me -parent e:\robot -family octo
-release 9503 -workarea 410
```

As a result of Annmarie’s work, the build tree for e:\robot looks like this:

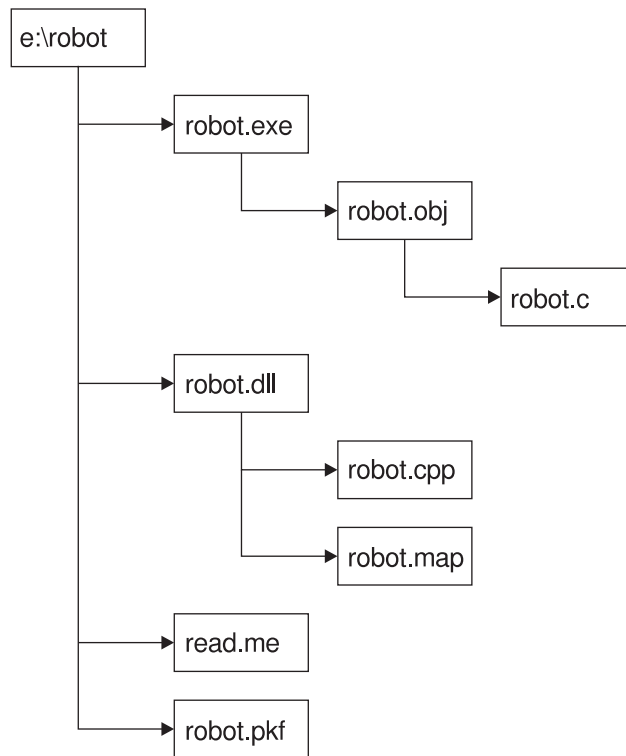


Figure 69. Adding the gather step to the build tree

The package file, robot.pkf, specifies the directories into which the robot files are gathered, with e:\robot as the target root directory. When Annmarie builds e:\robot, the .dll files are placed in e:\robot\ddl; the .bin files are placed in e:\robot\bin. Instead of extracting the built application from TeamConnection, the test team can pull the application from e:\robot.

If Annmarie wants to gather the same files into a different target directory, all she needs to do is write a different package file and connect the parts to a different parent.

Setting up a build tree for the NVBridge tool

If you have NetView DM/2 on your LAN, you can use the NVBridge tool to distribute your application to users. Setting this up is similar to setting up the build tree for the gather tool.

For example, Marylin, the packaging administrator for our robot development team, does the following:

1. She creates the top-level part for the new build tree. In this example, robotn.out is the output file from the NVBridge build. Marylin uses the following command:

```
teamc part -create robotn.out -none -builder NVB1
-release 9503 -workarea 817 -family octo
```

2. She writes a package file that contains instructions for the NVBridge tool and creates this file as a TeamConnection part:

```
teamc part -create robotnvb.pkf -text -parent robotn.out
-release 9503 -workarea 817 -family octo
```

For more about creating this package file, see “Writing a package file for NVBridge” on page 176.

3. She creates a builder, `nvb1`, that calls the NVBridge tool:

```
teamc builder -create nvb1 -script nvbridge.cmd -release 9503
           -environment os2 -condition == -value 0 -family octo
```

The build script for this builder specifies the `teamcpak` command, with its parameters.

4. She connects `e:\robot` and `robotnvb.pkf` to indicate that they are input to `robotn.out`:

```
teamc part -connect e:\robot robotnvb.pkf -parent robotn.out
          -family octo -release 9503 -workarea 817
```

As a result of Marylin’s work, the build tree for `robotnvb.out` looks like this:

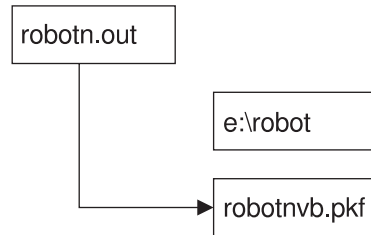


Figure 70. Adding the NVBridge step to the build tree

When Marylin builds `robotn.out`, NVBridge uses the rules in the package file to issue NetView DM/2 commands. The result is that the entire package is distributed using NetView DM/2.

In this example, Marylin uses NVBridge to distribute the output of the gather tool. This step is not required, but the gather tool is good preparation for distributing an application to users.

Setting up a build tree for other distribution tools

This process is similar to setting up for NVBridge. You create a top-level part in the build tree and a builder that invokes the distribution tool.

Chapter 14. Using the Gather tool



The Gather tool is available on OS/2 and Windows NT platforms. The command syntax for using the tool is the same for both platforms.

The Gather tool automates the movement of software and data from one directory to another on the same machine to prepare a package for electronic distribution. It can copy or erase files; it can create or delete directories.

This tool takes a list of input files and moves them into a directory structure as directed by a package specification file. You specify the target root directory path in this file, along with a collection of rules that instruct which files to copy to which directories. How these files and directories are actually handled is controlled via option flags.

By writing different package specification files, you can take the same input files and transfer them into different target directory structures.

Take the robot application as an example. We previously showed one possible directory structure, with each subdirectory containing files with the same extension:

```
e:\robot
  \dll
    hand.dll
    optics.dll

  \exe
    hand.exe
    optics.exe
```

By writing a different package file, you might put both .dll and .exe files in the same target directory:

```
f:\robot
  \bin
    hand.dll
    optics.dll
    hand.exe
    optics.exe
```

You can build both target directories concurrently.

Using the teamcpak command for the Gather tool

To start the Gather tool, use the teamcpak command. This command is found in the directory where the TeamConnection family server is installed. If it is started from a build script, it does not need to be in the execution path of the machine from which the build is started.

The complete command syntax for teamcpak gather looks like the following; you must supply a value for the words that start with a capital letter, such as String. You must specify the command parameters in the order shown.

```
teamcpak [-i] [-o "String"] gather Input_file...
```

Where

-i Specifies that only one *Input_file* is specified in the command: an include file containing the list of input files. This parameter is optional.

If you specify -i, it must precede the gather flag.

-o "String"

Specifies that the string listed in quotes be passed to the Gather tool. The opening quote must be followed by a blank. For a list of possible flags to be passed, see "Command line flags".

This parameter is optional. If you do not specify -o, the default settings for the tool are used.

If you specify -o, it must precede the gather flag.

gather

Specifies the tool to be invoked. If you specify -i or -o, they must precede this value.

Input_files

Specifies the files to be copied and the name of the package specification file. You can specify this parameter in these ways:

- Specify the name of an include file, whose contents is a list of input files. One of these input files must be a package specification file with the extension .pkf. In this case, you must also specify the -i parameter.
- Specify a list of two or more files. One of the files must be a package specification file with the extension .pkf.
- Specify the directory from which the files are to be copied and the name of the package specification file.

If more than one package file is listed, the first package file on the command line or in the include file is used, and the others are treated as ordinary files.

Command line flags

You can specify the following flags in the teamcpak command, using the -o parameter. All of these flags are optional. If you do not specify a flag, the teamcpak command runs using defaults.

-a Assume that the target tree structure might not exist. If a required directory does not exist, create it and continue processing.

This flag cannot be specified if the -t flag is specified.

If neither `-a` nor `-t` is specified, the default is to assume that the desired tree structure already exists. No verification is performed to confirm that the directories exist. If they do not, the condition is detected while the package file rules are being processed. If you stop the `teamcpak` command, some target directories might contain updated files.

- t** Ensure that the target tree is exactly the tree specified in the package file. If a directory of the same name exists, the Gather tool does the following:
 - Erases the entire contents of the directory and all of its subdirectories
 - Destroys the directory and all subdirectories
 - Performs a `mkdir` command to create the entire tree structure again as specified in the package file

This flag cannot be specified if the `-a` flag is specified.

If an `rmdir` command fails during processing, the `teamcpak` command stops.

If neither `-a` nor `-t` is specified, the default is to assume that the desired tree structure already exists. No verification is performed to confirm that the directories exist. If they do not, the condition is detected while the package file rules are being processed. If you stop the `teamcpak` command, some target directories might contain updated files.

- m** Accept missing source files.

If this flag is not specified, the default is to ensure that at least one file matches each source specification in the package file. If a match is not found, the Gather tool stops processing.

- d** Accept duplicate files. If a file is found on the target directory that matches the source file specification, it is overwritten by the source file.

If this flag is not specified, the default is to ensure that no files on the target match the source file specification. For example, if the source specification is `g*.c`, and `greg.c` is found on the target, the Gather tool stops processing.

- c** Clean up the target directories. Erase all files on all target directories that existed before writing source files to these directories. No confirmation messages are issued, and permission errors are ignored.

If this flag is not specified, the default is to write the source files into the target directories without erasing existing files.

- e** End with delete. This action removes all source files and directories after the Gather tool successfully completes.

If this flag is not specified, the default is to end without deleting source files and directories.

- x** Abort without recovery. If the program does not end successfully, no attempt is made to restore the file system.

If this flag is not specified, the Gather tool attempts to restore the file system if the program does not end successfully. To do this, the tool first backs up the file system. The backup directory is the value of the `TMP` environment variable.

Examples of the `teamcpak gather` command

The following are examples of the `teamcpak gather` command.

```
teamcpak gather d:\demoapp demoapp.pkf
teamcpak gather a.exe b.exe \help\*.hlp demoapp.pkf
```

In the first example, an input source directory is specified. In the second example, a list of files is specified. In both cases, the files are to be copied into target directories as specified in the demoapp.pkf file.

```
teamcpak -i -o " -t -m -x" gather myfiles.lst
```

The file myfiles.lst contains a list of files to be transformed by the Gather tool, and the name of the package file to be used in the gather. The -o "-t -m -x" parameter passes three flags to the Gather tool:

- -t specifies that, if the target directories already exist, they be destroyed and recreated.
- -m specifies that processing continues even if a source file cannot be found.
- -x specifies that, if the program does not end successfully, the file system is left as is, with no attempt to restore it.

Writing a package file for the Gather tool

Use the package file to specify the target directories and the rules for copying files for a gather operation. You can also specify user exit programs to run before, during, or after the gather operation.

A sample package file named gather.pkf is shipped with TeamConnection. You can customize it for your own gather operations.

Syntax rules for a Gather package file

Follow these syntax rules when you write a package file:

- Package files are free format. Text is not positional, and many statements can exist on the same line.
- Comments can appear anywhere within the file. Use the characters #| and |# as delimiters, as shown in the following example:
#| This is a comment |#
- Package file keywords must be prefixed with a left parenthesis and must have a corresponding balanced right parenthesis to end the scope of the keyword.
- If the value for a keyword is a string that contains blanks or parentheses, enclose the string in double quotes.

The following shows the syntax of a package file for the Gather tool. Keywords must appear in the order shown. The first letter of an argument is capitalized; you must supply these values.

```
(DATA
  (PACKAGEFORMAT gather)
  (TARGETROOT Filename)
  (RULE
    (SOURCE Filename...)
    (TARGET Path)
    [(EXITPRIOR String... | EXITREPLACE String... | EXITPOST String...)] )
  )
  .
  .
  [(EXITPRIOR String...)]
  [(EXITPOST String...)]
)
```

Keywords for a Gather package file

DATA This keyword is required. It must be the first keyword in the package file, and it can be specified only once.

All other keywords are nested within the DATA clause.

PACKAGEFORMAT gather

This keyword is required. It can be specified only once. It tells the teamcpak command that this package file is for Gather.

TARGETROOT target_root_path

This keyword is required. It can be specified only once.

Use this keyword to identify the target root directory. Source files are copied to this directory as specified by the RULE statements.

Follow these guidelines when you select your TARGETROOT values:

- Include the drive letter along with the target directory.
- Specify a directory that contains few if any subdirectories that are unrelated to the data you are moving.
- If you specify a drive's root directory (*drive:*), run the teamcpak command using the defaults or only the *-x* or *-x -a* flags.
- Do not set the value of TARGETROOT to *drive:* under the following circumstances:
 - The TARGETROOT drive is the same as the drive from which the teamcpak command is run, and you have recovery set (that is, you have not specified *-o "-x"*).
 - The logical drive for the TARGETROOT has less than 50% free space, and you have recovery set (that is, you have not specified *-o "-x"*).

RULE This keyword is required. You can use one or more RULE keywords within a Gather package file.

Each RULE clause represents a set of Gather operations targeted for one target subdirectory. A RULE clause must contain one SOURCE and one TARGET keyword. The files in the SOURCE directory are copied to the TARGET path. The target path is derived by concatenating the value of TARGETROOT with a backslash (\), followed by the value of the TARGET keyword specified in the RULE clause.

A RULE clause can also contain one user exit clause: EXITPRIOR, EXITPOST, or EXITREPLACE. For a description of the exit keywords, go to page 171.

The following example copies all *.exe, *.cmd, and *.hlp files to target directory f:\demoapp\bin.

```
(DATA
.
.
(TARGETROOT f:\demoapp )
.
.
(RULE
(SOURCE *.exe *.cmd *.hlp)
(TARGET bin  )
```

```
)
:
.)
```

SOURCE <list of file specifications>

This keyword is required once for each RULE clause. It must be the first keyword within the RULE clause.

This keyword specifies the files to be copied to the path specified by the TARGET keyword. Specify a list of file specifications separated by blanks. You can use the wildcard characters supported by OS/2 or Windows NT.

The directory from which these files are copied depends on how the input files are specified in the teamcpak command:

- If the teamcpak command specifies a source directory, the files specified in the SOURCE keyword come from that directory or subdirectories of it. The full path of the source files is constructed by concatenating the directory from the teamcpak command with a backslash (\), followed by the file specifications found in the SOURCE keyword. You can specify subdirectories in the SOURCE file specifications.
- If the teamcpak command specifies a list of files, these files are first copied to a temporary directory, then copied from there to the TARGET directories. In this case, you can use OS/2 or Windows NT wildcards to specify multiple file names in the SOURCE file specifications, but you cannot specify subdirectories.

In the following example, directory d:\demoapp is specified on the teamcpak command:

```
teamcpak -o "-x -t -m" gather d:\demoapp demoga.pkf
```

The resulting source path is the concatenation of d:\demoapp with the SOURCE file specifications. Therefore, all of the .exe files in the directory d:\demoapp\bin are copied to the target directory e:\demoapp\bin.

```
(DATA
  (TARGETROOT e:\demoapp)
  .
  .
  (RULE
    (SOURCE bin\*.exe)
    (TARGET bin)
  )
  .
  .
)
```

In the following example, a list of input files is specified on the teamcpak command:

```
teamcpak -o "-x -m" gather c:\a.exe c:\b.exe d:\rexx\*.cmd demoga.pkf
```

The resulting source path for the files in the SOURCE clause is the concatenation of the teamcpak temporary directory with the SOURCE file specifications. Therefore, the source for the *.exe files is d:\teamcpak.@@@*.exe. The input files d:\teamcpak.@@@a.exe and d:\teamcpak.@@@b.exe are copied to the directory e:\demoapp.

```
(DATA
  (TARGETROOT e:\demoapp)
  .
  .
```

```

(RULE
  (SOURCE *.exe )
  (TARGET targetroot)
)
:
.
)

```

TARGET Target_path

This keyword is required one time in each RULE clause. It must follow the SOURCE keyword.

The value specified by this keyword is used to construct the target path into which the files specified by the SOURCE keyword are copied. The value of the TARGETROOT keyword is concatenated with a backslash (\), followed by the value of the TARGET keyword.

If you specify targetroot as the value, files are copied directly to the target root directory, not to a subdirectory.

In the first RULE clause of this example, files are copied to the target directory f:\demoapp\bin\files. In the second RULE clause, the target directory is f:\demoapp.

```

(DATA
  (TARGETROOT f:\demoapp )
  :
  .
  (RULE
    (SOURCE *.bin *.dll )
    (TARGET bin\files )
  )
  (RULE
    (SOURCE *.hlp )
    (TARGET targetroot )
  )
  :
  .
)

```

EXITPRIOR, EXITPOST, and EXITREPLACE String...

These keywords are optional. They specify a user exit program to run as part of the gather operation.

To specify an exit that is global to the Gather operation, specify EXITPRIOR or EXITPOST in the DATA clause. You can specify each of these keywords only once in the DATA clause. These keywords must come after all of the RULE clauses. EXITREPLACE cannot be used in the DATA clause.

You can also specify an exit that is specific to one RULE clause. Only one exit keyword is allowed in each RULE clause.

These keywords accept a list of strings separated by spaces. The first string is the name of the program to execute. The strings that follow are its parameters.

Using exit keywords in the DATA clause

When used within a DATA clause, these keywords identify a program or command to be executed within a command shell. EXITPRIOR executes before all RULE statements have been processed; EXITPOST, after all RULE statements.

The exit keywords accept any executable file or command. The exit program must return an integer return value, with zero meaning the exit was successful.

Using exit keywords in the RULE clause

EXITPRIOR, EXITPOST, and EXITREPLACE are optional within a RULE clause. Only one can be specified in any given RULE clause.

When used within a RULE clause, these keywords identify a program or command to be executed within a command shell before, after, or in place of processing of each Gather copy operation. The exit program is called once for each SOURCE specification entry within the SOURCE clause. Parameters are separated by spaces and passed to the exit in this order:

- Any parameters included in the invocation string
- The resolved SOURCE file specifications
- The resolved TARGET specification

The exit keyword accepts any executable file or command. The exit program must return an integer return value, zero meaning successful; it must also accept or ignore the additional Gather parameters added to the end of the invocation string.

When used in the context of the RULE clause, exit keywords must follow the TARGET keyword.

Using exit keywords: an example

In the following example, the first EXITPRIOR statement relates to the DATA clause and specifies a user backup exit program, which executes before performing Gather copy operations. This backup exit is passed two flags. The command stream executed in an OS/2 shell is:

```
"e:\util\backup.cmd \i \t"
```

The second occurrence of the keyword illustrates how to use it in the context of a RULE clause. In this example, an encryption program will run against each source file specification. The exit program is passed the \k:347867 key option, the value for the source specification, and the value for the target specification. In this example, the command stream executed in an OS/2 shell is:

```
"encrypt \k:347867 d:\demoapp\a.exe f:\demoapp\bin":
```

The package file looks like this:

```
(DATA
  (PACKAGEFORMAT gather)
  (TARGETROOT d:\tcws)
  (RULE
    (SOURCE *.exe *.cmd)
    (TARGET exe)
    #|this program will be run for each source file|#
    (EXITPRIOR encrypt \k:347867 )
  )
  (EXITPRIOR "e:\util\backup.cmd \i \t" )
)
```

Chapter 15. Using the NVBridge tool



The NVBridge tool is available only on OS/2.

The NVBridge tool supports automated distribution between a single NetView DM/2 CC server and its LAN-connected CC clients. It also supports remote distribution to APPC-connected NetView DM/2 servers and mainstream servers.

A sample build script named `nvbridge.cmd` is shipped with TeamConnection. It can be invoked within a TeamConnection builder. This build script maps TeamConnection build parameters to the command line syntax for invoking the NVBridge tool via the `teamcpak` command line interface.

You can use NVBridge as a builder for packaging in two ways:

- Integrate it with the gather step, so that the Gather tool leaves the package files in a directory from which NVBridge picks them up.
- Use it without the gather step. In this case, the build script for NVBridge must set up the directory and move files into it to interface correctly with the `teamcpak` command.

For information about setting up a build tree for running NVBridge, see “Setting up a build tree for the NVBridge tool” on page 162.

NVBridge produces the following NetView DM/2 output files:

A change file

This file, containing all of the software deliverables, is stored in the `fsdata` subdirectory of the NetView DM/2 directory. The file name is `buildID.cf`, where `buildID` is the ID specified in the `-o "-b"` flag of the `teamcpak` command.

A procedure file

This file is stored in the `fsdata` subdirectory of the NetView DM/2 directory. It contains the command instructions for uninstalling an installed software object during its next build. The file name is system-generated.

A flat data file

This file, containing the text data of mail information that accompanies other objects, is stored in the `fsdata` subdirectory of the NetView DM/2 directory. The file name is system-generated.

Catalog entries for the generated change file, procedure file, and mail notification object

These files are named according to the following naming convention:

`_Tx_corporation_ID.buildID.xxx.0.0`

Where:

- `x` is an identifier for the TeamConnection server. The default is `C`.
- `corporation_ID` is a string of up to ten characters. The default is `NULLCORP`.

- *Build ID* is a string of up to 16 characters, representing the ID specified on the -b flag of the teamcpak nvbridge command.
- *xxx* identifies the file. The following values are used:
 - REF for the generated change file
 - CMD for the generated uninstall procedure
 - MAIL for the generated mail notification object

Using the teamcpak command for NVBridge

To start the NVBridge tool, use the teamcpak command. This command is found in the directory where the TeamConnection family server is installed. If it is started from a build script, it does not need to be in the execution path of the machine from which the build is started.

The complete syntax for the teamcpak nvbridge command is the following. You must specify the command parameters in the order shown.

```
teamcpak -o "string" nvbridge input_source-directory
package_specification_file
```

-o "string"

Specifies that the string listed in quotes be passed to the NVBridge tool. For a list of possible flags to be passed, see "Command line flags" on page 175 .

Input_source_directory

A directory containing all the files and subdirectories of the software to be distributed using NetView DM/2. You must specify this directory as an absolute path with no wildcard characters.

The source root directory can be created in an earlier build step. The Gather tool can be used to create the source root directory and move the files into it as the TARGETROOT directory.

All files contained in the source root directory are included in the NetView DM/2 object that is built and distributed via NVBridge. It is important that the source root directory contain only the subdirectories and files required for the software package distribution.

File names within the source directory cannot contain blanks. Only non-hidden files are supported. The names of the files are reproduced on the target NetView DM/2 CC clients. Therefore, HPFS file names can be included only if the target NetView DM/2 CC clients listed in the package file have target drives that support HPFS.

If you are using the uninstall function, the uninstall program must reside in the source directory. If you use the MAIL keyword for remote servers, the MAIL text file that is sent to the remote servers must also reside within this directory.

package_specification_file

A file describing how the NVBridge function is to build, catalog, and distribute software. Additional controls of NVBridge processing are provided through the optional command line flags described in the following section.

Command line flags

You can specify the following flags in the `teamcpak` command, using the `-o` parameter. All of these flags except `-b` are optional.

-b:*buildID*

This flag is required. The build ID represents the software to be distributed. It is a string of up to 16 alphabetical characters or underscores. It cannot contain blanks or other special characters. If you are distributing software to NetView DM/2 clients whose target drives do not support HPFS file systems, the build ID can be only eight characters in length.

-v

Use this flag to verify that NetView DM/2 CC clients listed in the `INSTALLS` keyword of the package file are defined to the NetView server and are in an active state. Verification takes place one time at the start of the `teamcpak` command. Changes that occur after verification are not detected.

Verification fails if one or more clients is in running rather than active status. Running status indicates that the client currently has a NetView command in progress.

If this flag is not specified, NVBridge ignores errors resulting from undefined or inactive clients.

If the `SENDS` keyword is specified in the package file, this flag also verifies the following:

- All of the remote destinations are defined in the NetView DM/2 remote destinations table.
- Any transmission queues associated with the remote destinations are in a released state.
- The transmission queues are empty.

If the `INSTALLS` or `SENDS` keywords are not specified in the package file, verification always succeeds.

-m

Use this flag to monitor NVBridge-generated requests to NetView DM/2. If any install, uninstall, or send requests do not complete successfully, messages are generated.

This flag works with the `CLIENTINTERVAL`, `SENDINTERVAL`, and `ATTEMPTS` package file keywords. See "Writing a package file for NVBridge" on page 176 for details about using them to control monitoring durations and limits.

If you specify this flag, NVBridge continues to run as long as it takes to install on all the designated clients and send to remote destinations, or until the durations set by the package file keywords are exceeded.

If this flag is not specified, NVBridge submits the install and send requests and then ends without waiting for them to complete. In this case, you can use NetView DM/2 functions to track these pending requests.

-r

Use this flag to retry an earlier failed attempt to install. Using the same package file, you can correct install failures without rebuilding the software object and without installing again on clients that were successful the previous time.

`TEST` and `SENDS` keyword in the package file are ignored, because testing and sending are assumed to have taken place already.

-a

Use this flag to issue NetView `ACTIVATE` requests. This request causes the

client machine to reboot after installation processing and SEND requests are complete. This is the last request that NVBridge performs before completion.

This flag is ignored if the TEST and INSTALLS keywords are not specified in the package file.

If you specify this flag, you must also specify the `-m` flag.

- l** Use this flag to install software in a NetView DM/2 service area. This allows installation of files that replace system-locked files.

A NetView ACTIVATE request is required to move the files from the temporary service area and to make updates to the config.sys file take effect. You can use the `-a` flag to request the ACTIVATE. If you do not use the `-a` flag, you must use NetView to reboot the clients manually.

If you use this flag, the installation program must be installed on the client already, or the installation program must reside in the CC server's NetView DM/2 shared areas.

This flag is ignored if TEST and INSTALLS keywords are not specified in the package file.

- t:time** Use this flag to set a timer so that NVBridge runs at a later time. For example, you can invoke NVBridge, go home, and let it start during the night. This timer applies to all NVBridge functions, not individual INSTALLS or SENDS requests.

Valid values for time are 0000 to 2359. For example, specify `-t:0000` to run NVBridge starting at midnight. Specify `-t:1200` to start NVBridge at noon.

- f** Use this flag to prevent the sending of objects that have install or uninstall problems. If any INSTALL or UNINSTALL requests fail, NVBridge returns a nonzero return code and purges any pending requests.

If this flag is not specified, NVBridge returns a return code of 0 and continues with SENDS requests, even if some UNINSTALL or INSTALL requests fail to complete.

Examples of the teamcpak nvbridge command

The following is an example of the teamcpak nvbridge command.

```
teamcpak -o "-b:demoapp -m -v" nvbridge d:\demoapp demoapp.pkf
```

The input source directory, d:\demoapp, contains the files to be installed. The file demoapp.pkf is the package specification file.

The `-o` parameter passes three flags to the NVBridge tool:

- `-b:demoapp` specifies that demoapp is the build ID.
- `-m` specifies that NVBridge monitor requests to NetView DM/2 to completion.
- `-v` specifies that NVBridge verifies that all clients are defined to NetView DM/2 and active.

Writing a package file for NVBridge

This section describes the NetView DM/2 package file keywords and their effect on normal processing behavior.

A sample package file named `nvbridge.pkf` is shipped with TeamConnection. You can customize it for your own use.

Syntax rules for an NVBridge package file

Follow these syntax rules when you write a package file:

- Package files are free format. Text is not positional, and many statements can exist on the same line.
- Comments can appear anywhere within the file. Use the characters `#|` and `|#` as delimiters, as shown in the following example:
`#| This is a comment |#`
- Package file keywords must be prefixed with a left parenthesis and must have a corresponding balanced right parenthesis to end the scope of the keyword.
- If the value for a keyword is a string that contains blanks or parentheses, enclose the string in double quotes.

The following shows the syntax of a package file for NVBridge. The order of the keywords inside the `NVGLOBS` clause does not matter; all other keywords must appear in the order shown. You must supply the values for the strings that are shown in *italics*.

```
(DATA
  (PACKAGEFORMAT nvbridge
  (NVGLOBS
    [(MAIL filename)]
    [(TEAMCSERV x)]
    [(CORPID name)]
    (INSTALLDIR path)
    [(INSTALLPGM path\filename)]
    [(IPARMS parameters)]
    [(UNINSTALLPGM filename)]
    [(CLIENTINTERVAL n)]
    [(SENDINTERVAL n)]
    [(ATTEMPTS n)]
  )
  [(TEST
    (ENTRY client)
  )]
  [(INSTALLS
    (ENTRY client target_directory)
    .
    .
  )]
  [(SENDS
    destination
    .
    .
  )]
)
```

Keywords for an NVBridge package file

DATA This keyword is required. It must be the first keyword in the package file, and it can be specified only once.

All other keywords are nested within the `DATA` clause.

Example:

```
(DATA
  .
  .
```

```

        other keywords go here
        .
    )

```

PACKAGEFORMAT nvbridge

This required keyword must be the first keyword within the DATA clause. It can be specified only once. It tells the teamcpak command that this package file is for NVBridge.

Example:

```

(DATA
    .
    .
    (PACKAGEFORMAT nvbridge)
    .
)

```

NVGLOBALS

This required keyword must follow the PACKAGEFORMAT keyword within the DATA clause. It can be specified only one time.

All the global and default keywords for the package file are specified in the NVGLOBALS clause. At least one INSTALLDIR keyword must be specified within this clause; all other NVGLOBALS keywords are optional.

Keywords in the NVGLOBALS clause can appear in any order.

Example:

```

(DATA
    .
    .
    (NVGLOBALS
        .
        NVGLOBALS keywords go here
        .
    )
    .
)

```

MAIL *filename*

This optional keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword identifies a text file that is sent as a mail notification object to all destinations listed in the SENDS keyword. The file name specified must contain only the file name and extension, no path information. NVBridge searches the input directory specified on the command and uses the first file that matches this file name.

The mail object is created and cataloged on the local server along with the .ref and .cmd objects. Mail objects typically take the form of readme files.

Example:

In the following example, the input directory is searched for the first occurrence of the file readme.txt during NetView DM/2 processing. This file is used to create a new .mail object cataloged to NetView DM/2. This object is then sent along with the .ref and .cmd objects during the processing of any SENDS requests.

```

(DATA
    .
    .
    (NVGLOBALS

```

```

        .
        (MAIL  readme.txt )
    .
)
.
.
)

```

TEAMCSERV *x*

This optional keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword specifies a 1-character alphanumeric ID to be used in global name generation. It identifies the TeamConnection family server that built and distributed the global named object. If this keyword is specified, the first four characters of the global name are `_Tx_`. If this keyword is not specified, the first characters of the global name default to `_TC_`.

Example:

An example of a NetView DM/2 object name based on the following example is `_t1_nullcorp.demoapp.ref.0.0`.

```

(DATA
.
.
(NVGLOBALS
.
(TEAMCSERV 1 )
.
)
.
.
)

```

CORPID *name*

This optional keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword is used in constructing the global names of NetView objects generated by NVBridge. You can use it to further identify the NetView objects, for example by specifying the name of your company. The name specified in this keyword follows the value of the TEAMCSERV keyword in global names.

The name specified in this keyword can be from one to ten alphanumeric characters. If this keyword is not specified, it defaults to the value NULLCORP.

Example:

An example of a NetView DM/2 object name based on the following example is `_t1_ibmcorp.demoapp.ref.0.0`.

```

(DATA
.
.
(NVGLOBALS
.
(CORPID  ibmcorp)
.
)
.
.
)

```

INSTALLDIR *path*

This required keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword defines the default workstation directory that is to be created as the root software directory on the clients.

The *path* must be an absolute file specification, including a valid drive. You can override this value for individual clients, within the ENTRY clause of the INSTALLS keyword. This target directory need not exist on the CC clients. NetView creates this directory and populates it with the same data as contained in the input directory specified on the teamcpak command.

Example:

In the following example, the path e:\demoapp is used as the target drive and directory for all CC clients listed in the INSTALLS keyword. NetView DM/2 will install the files and subdirectory structure specified on the teamcpak command within a target directory e:\demoapp on each CC client machine.

```
(DATA
  .
  .
  (NVGLOBALS
    .
    (INSTALLDIR e:\demoapp )
    .
  )
  .
  .
)
```

INSTALLPGM *path\filename*

This optional keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword defines the installation program to be invoked after the software files are copied on the CC client machine. If this keyword is not specified, the installation process includes only the movement of files, and an installation program is not invoked.

The installation program can be specific to the software just installed. In this case, the installation program is itself a file that was copied. Alternatively, the installation program can be a separate software product installed previously on the client machine. The installation program must meet the following criteria:

- It must be in the client's execution path, or the executable name must include its fully qualified path.
- The return value of the program must be 0 if you want NetView DM/2 to catalog the installation as successful. If the program returns a value other than 0, NetView DM/2 concludes the installation was a failure.
- The installation program cannot be interactive; that is, it cannot expect user input.

The installation program can invoke other programs.

The INSTALLS entries and hence the path to the installation program can differ from client to client. As a result, the installation program often needs to know the target directory into which the software was installed. To aid with NetView environmental information such as this, you can include the


```

        (IPARMS "\i \t \g \d:${TargetDir}" )
    .
)
.
)

```

UNINSTALLPGM *filename*

This optional keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword defines the program that will be used to uninstall this version of the software when its next version is installed. For example, if you are installing version 1.0 of your package, use this keyword to specify the program that will be used to uninstall version 1.0 when version 1.1 is installed.

The uninstall program must be contained in the input software directory. NVBridge searches the input directory specified on the teamcpak command and uses the first file that matches the file name specified on this keyword.

If both this keyword and the INSTALLS or TEST keywords are specified, then a NetView PROCEDURE object is created along with the software object. When the next version of this package is installed, this uninstall procedure is run against previous install clients, before the removal of the old version of the software and the creation of the new NetView software object. NVBridge tracks only whether the uninstall operations complete, not whether they are successful.

The uninstall program can itself invoke other programs.

Example:

In the following example, the input directory is searched for the first occurrence of the file `uninst.cmd` during NetView DM/2 processing. This file is used to create a new uninstall PROCEDURE object cataloged to NetView DM/2.

```

(DATA
.
.
(NVGLOBALS
.
  (UNINSTALLPGM  uninst.cmd )
.
)
.
)

```

CLIENTINTERVAL *n*

This optional keyword can appear at any place within the NVGLOBALS clause. It can be specified only one time.

This keyword identifies the sleep interval used for monitoring client operations such as uninstalling and installing. It is used with the `-m` flag on the teamcpak command.

Specify a value in seconds. Valid values are from 10 to 600. The default is 15 seconds.

Example:

```

(DATA
.
.

```



```

(NVGLOBS
.
(CLIENTINTERVAL 60 )
.
)
.
.
)

```

SENDINTERVAL *n*

This optional keyword can appear at any place within the NVGLOBS clause. It can be specified only one time.

This keyword identifies the sleep interval used for monitoring SEND operations. It is used with the `-m` flag on the `teamcpak` command.

Specify a value in seconds. Valid values are from 10 to 600. The default is 15 seconds.

Example:

```

(DATA
.
.
(NVGLOBS
.
(SENDINTERVAL 40 )
.
)
.
.
)

```

ATTEMPTS *n*

This optional keyword can appear at any place within the NVGLOBS clause. It can be specified only one time.

This keyword identifies the maximum number of attempts to monitor NVBridge operations. Specify a number from 1 to 6. The default is 4.

This value is combined with the CLIENTINTERVAL and SENDINTERVAL values to compute the maximum monitoring time. For each operation per client or prior remote destination, NVBridge uses this formula:

Monitor and check every *x* seconds up to *y* times

Where *x* is the value for CLIENTINTERVAL or SENDINTERVAL, and *y* is the value for ATTEMPTS.

Example:

If you are installing to four clients and you specify the `-m` flag on the `teamcpak` command, by default NVBridge monitors every 15 seconds, up to four attempts. It repeats this 4 times for each of the four clients, resulting in a total of 16 attempts at 15-second intervals.

```

(DATA
.
.
(NVGLOBS
.
(ATTEMPTS 4 )
.
)

```

```
)
:
)
```

TEST This optional keyword appears within the DATA clause before the INSTALLS or SENDS keywords. It can be specified only one time.

This keyword identifies a single NetView DM/2 CC client to be used as a test machine. All normal processing is first performed against this single client. If everything succeeds, normal processing continues for all clients listed in the INSTALLS keyword; otherwise normal processing stops.

This keyword accepts a single ENTRY keyword, which identifies the client. Do not repeat this client value in the INSTALLS entries, or NVBridge might fail.

Example:

In the following example, the test CC client named CLIENT1 will be used to perform a test installation. The software and data will be installed to the target client directory e:\demoapp.

```
(DATA
.
.
  (TEST
    (ENTRY CLIENT1 e:\demoapp)
  )
.
)
```

INSTALLS

This optional keyword is specified within the scope of the DATA clause. It must follow the NVGLOBALS and TEST keywords. It can be specified only one time.

This keyword identifies the list of ENTRY keywords specifying the clients where the software object is to be installed. Duplicate ENTRY keywords for the same client are ignored.

ENTRY *client, directory*

This required keyword is specified within the scope of the TEST or INSTALLS clauses. It can be specified many times.

This keyword identifies a NetView DM/2-defined CC client workstation on which the software is to be installed. For each ENTRY keyword, you must specify the name of a CC client machine.

Optionally, you can also specify a target installation directory for the client. This directory overrides the directory specified in the INSTALLDIR keyword. The target directory must be an absolute file specification, including a valid drive. If this value is found to be not valid, NetView uses the default value found in the INSTALLDIR keyword.

Example:

In the following example, the .ref object created by NetView DM/2 will be installed to CLIENT1, CLIENT2, CLIENT3, and CLIENT4 CC client machines. In the case of CLIENT2 and CLIENT4, the software is installed in the default INSTALLDIR value of d:\demoapp. For the others, the target directory in the ENTRY keyword overrides the INSTALLDIR value.

```

(DATA
  .
  (NVGLOBALS
    .
    (INSTALLDIR  d:\demoapp )
    .
  )
  .
  (INSTALLS
    (ENTRY  CLIENT1  e:\demoapp)
    (ENTRY  CLIENT2           )
    (ENTRY  CLIENT3  c:\demoapp)
    (ENTRY  CLIENT4           )
  )
  .
  .
)

```

SENDS *destination ...*

This optional keyword is specified within the scope of the DATA clause. It must follow the NVGLOBALS, TEST, and INSTALLS keywords. It can be specified only one time.

This keyword identifies the list of remote destinations that are to receive NVBridge-created objects. These destinations are APPC-connected NetView DM family servers.

Each remote destination in this list should be configured to accept creation or replacement of cataloged objects. If the remote server does not allow incoming SENDS of objects, then NVBridge cannot send objects to it. Also, if the remote server accepts only creates and not replacements, then NVBridge can send it only objects that do not already exist in its catalog.

Example:

```

(DATA
  .
  .
  (SENDS
    USSNANR.AUSTIN2
    USSNANR.AUSTIN3
    USSNANR.NEWYORK1
    USSNANR.CHICAGO4
  )
  .
  .
)

```

Problem determination for NVBridge

If a particular object has a status of SCHEDULED or IN PROGRESS that does not reflect its true status, then the existing version of the software object might be in a bad locked state. The result is that NetView DM/2 cannot build new versions of the object. NetView DM/2 always attempts to purge all previous locked requests when it builds new versions of software to be distributed. However, there are abnormal NetView DM/2 cases where locked objects require further manual intervention to correct locked NetView DM/2 files.

If during NetView DM/2 processing, messages indicate that NetView DM/2 failed because it could not remove a previous version of an object, try the following steps. If these steps fail to correct the problem, then contact your IBM NetView DM/2 representative.

1. Check the NetView DM/2 message.dat file on the server and, if possible, on the client, checking to see if any NetView DM/2 errors have been detected. If a NetView DM/2 error has occurred, then take appropriate steps to report and correct the problem.

Rebooting your system sometimes will temporarily correct the NetView DM/2 condition so that you can get your work done. Even if you take this route, go on to the next steps after rebooting.

2. Look at the request queue contents and at the install history to find the object name generated from NetView DM/2 processing. The name of the NetView DM/2 object is also in the messages. The locked entries can be identified by inspecting the NetView DM/2 request queue and seeing an entry for the object, where it is obvious that the entry is not being processed. At other times, a locked entry can be found by looking at the install history for clients that have install status other than INRU.
3. Temporarily undefine from the server the CC clients that involve the lock. Locked requests for undefined clients can sometimes be corrected by making the CC client unknown to the CC server.
This step causes the NetView DM/2 CC client to stop running. Restart the client either manually or via another remote access to the machine.
4. If any requests are in the request queue, try to both purge and delete the request. (If you are using the GUI, the following steps can be combined into one or two steps.)
5. Perform a Deleteit against the object's group name to remove install target information that might have been set previously. Do this for all client workstations with the /ws option.
6. Remove all install history for the object.
7. Redefine to the server the CC clients that were locked.
8. Try your initial NetView DM/2 request again.

NVBridge utilities

TeamConnection provides a collection of utilities that can be combined into a user-defined build script, to help automate customized forms of packaging steps. You can use these utilities instead of the teamcpak command, to customize your distribution steps.

Note: These interfaces are program-sensitive interfaces used by NVBridge. As a result, these interfaces are likely to change and evolve from release to release, and no IBM commitment is implied that these interfaces will remain unchanged and compatible in future versions.

The only form of parameter checking performed by these utilities is verifying that required parameters are specified and that the parameter list meets syntax requirements. These utilities assume that valid parameter values are passed to them.

To display the syntax of these utilities, type the name followed by a question mark. For example, type FHP0BDEL ? to see the syntax of this tool.

FHPSTAT

Use this function to check the status of the NetView DM/2 CC server components. This is used to determine if NetView DM/2 is available and if the necessary components have been started.

This utility continues until it has made the number of attempts specified in the *loop* parameter or until the return code is 0, whichever occurs first.

Syntax

```
fhpstat [timer], [loop], [config=lan|APPC], [display=YES|no]
```

- *timer* - time in seconds for sleep interval for monitoring.
- *loop* - number of times to loop before giving up. Defaults to 1 attempt, meaning it will not loop and monitor.
- config - LAN means to check only for Agent and Change Controller components, whereas APPC means to check also for Transmission Controller.
- display - specifies whether to display results or not.

Return Codes

- Returns 0 if the CC server components are running. What components have to be running depends on standalone LAN or APPC environment.
- Returns 1 if one or more CC server components are not running.
- Return 4 if not an APPC configured CC server.
- Return 8 if not a NetView DM/2 CC server machine.

Example

```
fhpstat 30, 2, lan, yes
```

FHPOBDEL

This function unconditionally attempts to remove all NetView DM/2 information about a cataloged NetView DM/2 object, including anything currently in the request queue. Deletes all component name information and any previous NetView DM/2 ADDIT requests that might be in effect for the object.

Syntax

```
fhpobdel object, [display=YES|no]
```

- *object* - NvDM/2 global object name.
- display - specifies whether to display results or not.

Return Codes

- Returns 0 if the syntax of FHPOBEL was valid. Check for success using the FHPISCAT tool.
- Return 8 if a parameter error occurred or object name was not specified.

Example

```
fhpobdel _tc_demoapp.nullcorp.ref.0.0, no
```

FHPOBMON

Use this function to check the install history for a particular NetView DM/2 global object against an input list of install clients listed in an input file *inclients*.

This tool continues until it has made the number of attempts specified in the *loop* parameter or until the return code is 0, whichever occurs first.

This function only supports installation monitoring related to CC clients, not to the local CC server.

Syntax

```
fhpobmon object, inclients, [timer], [loop], [display=YES|no]
```

- *object* - NetView DM/2 global object name.
- *inclients* - the name of the file containing the list of CC clients.
- *timer* - time in seconds for sleep interval for monitoring.
- *loop* - number of times to loop before giving up. Defaults to 1 attempt, meaning it will not loop and monitor.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if an *object* existed and all clients listed in *inclients* file were installed. The *inclients* file is emptied.
- Returns 1 if an *object* existed but some of the clients in the *inclients* were not reflected in the object's install history. Input file is updated with the list of only those clients in *inclients* originally that did not show up in the history of the object.
- Returns 2 if an *object* did not exist in CC server catalog. The input file is unchanged.
- Returns 8 if a parameter error occurred on invocation. The input file is unchanged.
- Returns 16 if an internal error occurred. The input file is unchanged.

Example

```
fhpobmon _tc_demoapp.nullcorp.ref.0.0, infile, 30, 2, yes
```

Each line of the file *infile* must contain a client name. No blank lines are allowed.

```
client1
client2
client3
beta1
test1
:
```

FHPOBDIF

Use this function to cross-reference the differences between the install history of a NetView DM/2 global object and a particular list of install clients listed in an input file. All clients in *file1* that are not in the install history are left in *file1*. All clients in the install history that are not in *file1* are put into *file2*. Hence *file1* is updated with the list of clients that should be installed, that are not already installed, and *file2* is updated with the list of clients that are installed, but are not in the list of clients that should be installed. This command generates the differences.

This function only supports install history related to CC clients, not the local CC server.

Syntax

`fhpbodif object, file1, file2, [display=YES|no]`

- *object* - NetView DM/2 global object name.
- *file1* - input file of comparison clients, also output file for clients that were not in the install history.
- *file2* - output file updated with clients that were in the install history but which were not in the original *file1* list of clients.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if an object existed and there were no differences. *file1* and *file2* are emptied.
- Returns 1 if an object existed but there were differences. *file1* and *file2* reflect these differences.
- Returns 2 if an object did not exist. *file1* and *file2* are unchanged.
- Returns 8 if a parameter error occurred on invocation. *file1* and *file2* are unchanged.
- Returns 16 if an internal error occurred. *file1* and *file2* are unchanged.

Example

```
fhpbodif _tc_demoapp.nullcorp.ref.0.0, file1, file2, no
```

Each line of the file *file1* must contain a client name. No blank lines are allowed.

```
client1
client2
client3
beta1
test1
:
:
```

FHPISCAT

This function checks to see if a NetView DM/2 object exists in the NetView DM/2 catalog. It checks only for a catalog entry, not the associated file.

Syntax

`fhpiscat object, [display=YES|no]`

- *object* - NetView DM/2 global object name.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if an object was cataloged.
- Returns 1 if the object was not cataloged.
- Return 8 if a parameter error occurred and object name was not specified.

Example

```
fhpiscat _tc_demoapp.nullcorp.ref.0.0
```

FHPICAT

This function does the following:

- Creates a NetView DM/2 software object based on the name specified by *object*

- Walks the input directory *sdir* to create the filespeclist section of the object
- Names the change file based on *buildid*
- Sets the InstallDir value in the profile to the value of *tdir*
- Sets the InstallSection of the profile to reference the install program of *ipgm* and parms of *iparms*

Syntax

```
fhpicat object, tdir, sdir, buildid, [ipgm], [iparm], [display=YES|no]
```

- *object* - NetView DM/2 global object name.
- *tdir* - value to be used for TargetDir in object change profile.
- *sdir* - fully qualified path of input directory where software resides.
- *buildID* - value to be used in naming the change file created and stored in NetView DM/2 fsdata subdirectory.
- *ipgm* - name of an installation program to be used for installation.
- *iparm* - value of installation program parameters to be specified in the object profile.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if a new object create attempt was successful. FHPISCAT can be used to verify that the object was created.
- Returns 1 if a new object create attempt failed.
- Returns 8 if a parameter error occurred.

Example

```
fhpicat _tc_demoapp.nullcorp.ref.0.0, d:\demoapp, c:\demoapp,  
$(targetdir)\inst.exe, -i -f -t, yes
```

FHPUCAT

This function does the following:

- Creates a NetView DM/2 PROC (procedure) object based on the name specified by *object*.
- Walks the input directory *sdir*, looking for the first occurrence of the *procedure* file.
- Copies the file into the NetView DM/2 fsdata subdirectory.
- Catalogs the object to NetView DM/2. This object can then have NetView DM/2 INITIATE commands requested against it.

Syntax

```
fhpucat object, procedure, sdir, [display=YES|no]
```

- *object* - NetView DM/2 global object name.
- *procedure* - the file name of the REXX procedure file located within the *sdir* that is to be used to create the object.
- *sdir* - fully qualified path of input directory where software resides.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if a new object create attempt was successful. FHPISCAT can be used to verify that the object was created.
- Returns 1 if a new object create attempt failed.

- Returns 8 if a parameter error occurred.

Example

```
fhpucat _tc_demoapp.nullcorp.proc.0.0, uninst.cmd, c:\demoapp, no
```

FHPMCAT

This function does the following:

- Creates a NetView DM/2 flat data text object based on the name specified by *object*.
- Walks the input directory *sdir*, looking for the first occurrence of the *mail_file* file name.
- Copies the file into the NetView DM/2 fsdata subdirectory.
- Catalogs the object to NetView DM/2. This object can then have NetView DM/2 SEND commands requested against it, or its contents can be viewed.

Syntax

```
fhpmdat object, mail_file, sdir, [display=YES|no]
```

- *object* - NetView DM/2 global object name.
- *mail_file* - the file name of a text file located within the *sdir* specified that is to be used to create the object.
- *sdir* - fully qualified path of input directory where software resides.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if a new object create attempt was successful. FHPISCAT can be used to verify that the object was created.
- Returns 1 if a new object create attempt failed.
- Returns 8 if a parameter error occurred.

Example

```
fhpmdat _tc_demoapp.nullcorp.txt.0.0, readme.txt, c:\demoapp, no
```

FHPVERIF

This function verifies that the CC clients listed in the *inclients* file are defined to the local CC server and in active status. The file *inclients* is then modified to contain only those clients that were not defined or active, or it is emptied if all clients are verified.

This tool continues until it has made the number of attempts specified in the *loop* parameter or until the return code is 0, whichever occurs first.

Syntax

```
fhpverif inclients, timer, loop, [display=YES|no]
```

- *inclients* - the name of the input file containing the list of CC client names to be verified. This file will be modified to contain the list of those clients that failed to verify.
- *timer* - time in seconds for sleep interval for monitoring.
- *loop* - number of times to loop before giving up. Defaults to 1 attempt, meaning it will not loop and monitor.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if every client in the file was defined and active to local CC server.
- Returns 1 if one or more clients in the file did not verify.
- Returns 8 if a parameter error occurred or an input file *inclients* did not exist.

Example

```
fhpverif infile1.inp, 30, 2
```

Each line in the file (infile1.inp) must contain a client name. No blank lines are allowed.

```
client1
client2
client3
beta1
test1
:
:
```

FHPRQPUR

This function purges any outstanding requests in the NetView DM/2 request queue related to the *object*. After the function completes, you can use the FHPRQMON tool to confirm that no *object*-related requests are in the request queue.

This function supports only object-related requests made to CC clients, not the local CC server. In other words, if there is a request for this *object* but it is for the CC server, then the request will not be detected or purged.

Syntax

```
fhprqpur object, forced=yes|no, [display=YES|no]
```

- *object* - NetView DM/2 global object name to be purged from request queue.
- forced - a yes or no value that defaults to no. If yes is specified, this tool will use all means possible to force the purging of related requests. If no is specified, this tool will try only normal means to purge related requests. A forced purge can result in requests being purged in the middle of processing.
- display - specifies whether to display results or not.

Return Codes

- Returns 0 if one or more requests for the specified *object* was found in the request queue, and a purge attempted.
- Returns 1 if no matching requests were found in the request queue.
- Returns 8 if a parameter error occurred.
- Returns 16 if an internal processing error occurred.

Example

```
fhprqpur _tc_demoapp.nullcorp.ref.0.0, yes, yes
```

FHPRQMON

This function monitors any outstanding requests in the NetView DM/2 request queue related to the *object*.

This tool continues until it has made the number of attempts specified in the *loop* parameter or until the return code is 0, whichever occurs first.

This function only supports object-related requests made to CC clients, not the local CC server.

Syntax

`fhprqmon object, file, [timer], [loop], [display=YES|no]`

- *object* - NetView DM/2 global object name to be monitored from request queue.
- *file* - an output file to be created/updated to contain the list of request identifiers that are in the request queue regarding the particular object.
- *timer* - time in seconds for sleep interval for monitoring.
- *loop* - number of times to loop before giving up. Defaults to 1 attempt, meaning it will not loop and monitor.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if there are no outstanding requests for the *object*.
- Returns 1 if there were requests detected in the queue and *file* has the list of their request IDs.
- Returns 8 if a parameter error occurred.
- Returns 16 if an internal processing error occurred.

Example

```
fhprqmon _tc_demoapp.nullcorp.ref.0.0, outfile, 0, 1
```

The output in the file outfile would look like the following:

```
3
4
16
⋮
```

FHPTRVER

This function verifies that every NetView DM/2 remote destination listed in the input file *indests* is defined to the local CC server and that all the related NetView DM/2 transmission queues are empty and in a released state.

This tool continues until it has made the number of attempts specified in the *loop* parameter or until the return code is 0, whichever occurs first.

Syntax

`fhptrver indests, timer, loop, [display=YES|no]`

- *indests* - the name of the input file containing the list of NetView DM/2 remote destination names to be verified. This file will be modified to contain the list of those destinations that failed to verify, or the file will be empty if they all verified.
- *timer* - time in seconds for sleep interval for monitoring.
- *loop* - number of times to loop before giving up. Defaults to 1 attempt, meaning it will not loop and monitor.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if every remote destination in the file was defined and related transmission queue was empty and in a released state.
- Returns 1 if one or more remote destinations did not verify.
- Returns 8 if a parameter error occurred or an input file *indests* did not exist.

- Returns 16 if an internal processing error occurred.

Example

```
fhptrver infile1.inp, 30, 2
```

Each line of `infile1.inp` must contain a remote destination name. No blank lines are allowed.

```
usibmnr.austin1
usibmnr.austin2
usibmnr.newyork3
usibmnr.newyork6
usibmnr.chicago1
:
:
```

FHPTRPUR

This function empties the transmission queues and releases the transmission queues corresponding to the list of remote destination names specified in the input file *indests*.

Syntax

```
fhptrpur indests, [display=YES|no]
```

- *indests* - the name of the input file containing the list of NetView DM/2 remote destination names to be purged. This file will be modified based on the return codes.
- *display* - specifies whether to display results or not.

Return Codes

- Returns 0 if no purges were performed due to all transmission queues already being empty and released. Input file is emptied.
- Returns 1 if some purge modifications were performed against transmission queues. The input file is updated to reflect that remote destinations triggered purge requests.
- Returns 8 if a parameter error occurred or there is a missing input file.
- Returns 16 if an internal processing error occurred.

Example

```
fhptrpur infile1
```

Each line of `infile1` must contain a remote destination name. No blank lines are allowed.

```
usibmnr.austin1
usibmnr.austin2
usibmnr.newyork3
usibmnr.newyork6
usibmnr.chicago1
:
:
```

Chapter 16. Using the Tivoli/Courier packaging tool

The Tivoli/Courier packaging tool supports automated distribution between a single Tivoli/Courier server and its TCP/IP-connected clients. The Tivoli/Courier tool works either by itself or in conjunction with TeamConnection's Gather tool to enable you to distribute files through Tivoli/Courier. Use of this tool requires you to be familiar with Tivoli configuration and system administration so that TeamConnection can start Tivoli/Courier to distribute file packages.

The Tivoli/Courier distribution tool must be run on a Tivoli managed node running on any of TeamConnection's UNIX platforms or Windows NT.

The Tivoli/Courier distribution tool includes a sample build script named `softdist` (on UNIX platforms) or `softdist.exe` (on Windows/NT). It can be run from within a TeamConnection builder. This build script maps TeamConnection build parameters to the command line syntax for the Tivoli/Courier tool through the `teamcpak` command line interface.

You can use Tivoli/Courier as a builder for packaging in two ways:

- Integrate it with the gather step, so that the Gather tool leaves the package files in a directory from which Tivoli/Courier picks them up.
- Use it without the gather step. In this case, the build script for Tivoli/Courier must set up the directory and move files into it to interface correctly with the `teamcpak` command.

To simplify the interface, the Tivoli/Courier tool uses a select set of options. If you want to take full advantage of Tivoli/Courier features, you can import a Tivoli/Courier package specification. Importing a package specification provides you access to all Tivoli/Courier functions.

The Tivoli/Courier tool produces a Tivoli FilePackage, which is used for distribution.

Using the `teamcpak` command with Tivoli/Courier

To start the Tivoli/Courier tool, use the `teamcpak` command. This command is found in the directory where the TeamConnection family server is installed. If it is started from a build script, it needs to be in the execution path of the build server.

The complete syntax of the `teamcpak softdist` command follows. You must specify the command parameters in the order shown.

```
teamcpak [-i] [-o "string"] softdist inputFile
```

-i Specifies that only one *inputFile* is specified in the command: an include file containing the list of input files. This parameter is optional.

-o "*string*"

Specifies that the string listed in quotes be passed to the Tivoli/Courier tool. For a list of possible flags to be passed, see "Command line flags" on page 196 .

inputFile

Specifies the files to be copied and the name of the package specification file. You can specify this parameter in these ways:

- Specify the name of an include file, whose contents is a list of input files. One of these input files must be a package specification file with the extension .pkf. In this case, you must also specify the -i parameter.
- Specify a list of two or more files. One of the files must be a package specification file with the extension .pkf.
- Specify the directory from which the files are to be copied and the name of the package specification file.

If more than one package file is listed, the first package file on the command line or in the include file is used, and the others are treated as ordinary files.

The following are examples of specifying input files.

```
teamcpak -i softdist myInputFile
teamcpak softdist inputFile1 inputFile2 inputFile3 . . .
teamcpak softdist d:\inputDir\myPkfFile.pkf
```

Command line flags

You can specify the following flags in the teamcpak command, using the -o parameter. All of these flags are optional.

- a** Create directories on the target.
- c** Clear the target (delete all specified files and directories) before the apply. If you use this option, do not use the -x option.
- t** Overwrite existing files (delete files on the target prior to distribution).
- m** Accept input errors, such as missing files and directories from the SOURCE keyword.
- n** Send no notices to Tivoli. If you want to post Tivoli notices, you must configure Tivoli Notices before using this packaging tool.
- p** Preview only; do not actually distribute files.
- r** Reboot the target after distribution.
- x** If an error occurs, leave any distributed files on the target; do not clean up. If you use this option, do not use the -c option.
- k** Keep the Tivoli file package. To enable the Tivoli/Courier tool to perform more efficiently, the Tivoli/Courier package file is created when the package part is created and then destroyed and recreated whenever the part is modified. Use the -k option to prevent the package file from being destroyed.

Example of the teamcpak softdist command

The following is an example of the teamcpak softdist command.

```
teamcpak -i -o "-a -n -t" softdist Client.lst
```

The -i parameter specifies that the input file Client.lst is to be used. The -o parameter passes the following options to Tivoli/Courier:

- -a creates directories on the target.
- -n indicates that no error notices are to be sent to Tivoli/Courier.
- -t indicates that any existing files on the target are to be overwritten.

Writing a package file for Tivoli/Courier

This section describes the Tivoli/Courier package file keywords and their effect on normal processing behavior.

A sample package file named `client.pkf` is shipped with TeamConnection. You can customize it for your own use.

Syntax rules for a Tivoli/Courier package file

Follow these syntax rules when you write a package file:

- Package file keywords must appear in the order shown below.
- Package file keywords must be prefixed with a left parenthesis and must have a corresponding balanced right parenthesis to end the scope of the keyword.
- If the value for a keyword is a string that contains blanks or parentheses, enclose the string in double quotes.
- Default options are supplied for all Tivoli/Courier required Tivoli/Courier options. Specific options can be set in your TeamConnection package file.
- Comments can appear anywhere within the file. Use the characters `#|` and `|#` as delimiters, as shown in the following example:

```
#| This is a comment |#
```

The following shows the syntax of a package file for Tivoli/Courier. The keywords must appear in the order shown here. You must supply the values for the strings that are shown in *italics*.

```
(DATA
  (PACKAGEFORMAT softdist)
  (TARGETROOT filename)
  (MANAGER ProfileManager)
  (NODES "ManagedNode... PCManagedNode...")
  (IMPORT filename |
    [(DISTRIBUTE [FULL | CHANGED])]
    [(INSTALLPGM filename)]
    [(LOGNODE ManagedNode)]
    [(LOGFILE directory)]
  )
)
```

Keywords for a Tivoli/Courier package file

DATA This keyword is required. It must be the first keyword in the package file, and it can be specified only once.

All other keywords are nested within the DATA clause.

Example:

```
(DATA
  .
  .
  other keywords go here
  .
  .
)
```

PACKAGEFORMAT *softdist*

This required keyword must be the first keyword within the DATA clause. It can be specified only once. It tells the `teamcpak` command that this package file is for Tivoli/Courier.

Example:

```
(DATA
.
.
(PACKAGEFORMAT softdist)
.
.)
```

TARGETROOT

This keyword specifies the directory path to which files are to be distributed on the target systems. You can specify only one target root. All target systems use identical target roots.

Example:

```
(DATA
.
.
(TARGETROOT /usr/local/teamc/images)
.
.)
```

MANAGER

This keyword specifies a Tivoli/Courier profile manager that you have already created in the Tivoli/Courier system.

Example:

```
(DATA
.
.
(MANAGER Distrib1)
.
.)
```

NODES

This keyword specifies the nodes to which the files are to be distributed. These must already have been defined to the profile manager as subscriber ManagedNodes or PCManagedNodes. To distribute files to non-subscribers, you need to use Tivoli/Courier options set in an import file package definition.

Example:

```
(DATA
.
.
(NODES "tcaix01 tcaix02")
.
.)
```

IMPORT

Use this keyword to select Tivoli/Courier options not supported in the -o parameter of the teamcpak softdist command. The *filename* parameter is the name of a Tivoli/Courier import file package definition. You can generate an import file using the Tivoli/Courier user interface. If you use the IMPORT keyword, then instead of calling the standard Tivoli/Courier packaging command the Tivoli/Courier tool will call wimpftp to get all of the Tivoli/Courier configuration options. Using the IMPORT keyword disables other options and causes errors if they are specified.

If you specify the IMPORT keyword, do not specify the DISTRIBUTE, INSTALLPGM, LOGNODE, or LOGFILE keywords.

If you use the INCLUDE option in an import file, it is overridden by the list of files provided to the teamcpak command.


```
(DATA
  .
  .
  (IMPORT importFile)
  .
  .
```

Specify FULL to distribute all files or CHANGED to distribute only those changed since the last distribution. The default is FULL.

```
(DATA
.
.
(DISTRIBUTE CHANGED)
.
.
```

Use this keyword to specify an installation script to be run during distribution on each node that receives files. Specify the full file path name of the script.

```
(DATA
.
.
(INSTALLPGM /tivoli/fpTeamcAIX/tcinstl.ksh)
.
.
```

This keyword specifies the system on which the log file is located. The node name you specify must be a managed node. The default is the current build machine or a machine running teamcpak.

```
(DATA
  .
  .
  (LOGNODE tcaix04)
  .
  .
```

This keyword specifies the directory path and file name of the log file on the log node. Include this keyword only if you use the LOGNODE keyword. The default value for this keyword is `softdist.log`.

```
(DATA
.
.
(LOGFILE /tmp/softdist.log)
.
.
```

Problem determination for the Tivoli/Courier tool

If you are having trouble distributing files using the Tivoli/Courier distribution tool, you can use the following tools or teamcpak options to determine what the problem is:

Log file

Check the softdist.log file (or the file name you specified in the LOGFILE keyword) for error messages.

Mail Check Tivoli mail messages generated during the distribution.

-k option

Run the teamcpak command with the -k option to keep the package file after the distribution has been run.

-x option

Run the teamcpak command with the -x option to leave any distributed files on the target.

Trace facility

Run teamcpak with the trace facility. Use this facility only under guidance of an IBM service representative. See the *Administrator's Guide* for more information.

The following message displays when a Tivoli/Courier command fails during a distribution.

```
6022-303 Tivoli/Software Distribution %s command failed with return code: RC.
To correct problem use:
- package file parameters LOGNODE and LOGFILE to record Tivoli output,
- packaging option "-k" to keep Tivoli File Package and teamcpak log file
  or "-m" to ignore input errors,
- packaging option "-x" to not clean up files that are distributed,
- TeamConnection Trace facility (see TeamConnection Administration Guide)
- or Tivoli Trace facility (see Tivoli documentation)
```

Sample package file

The following is an example of scripts and items required to automatically execute packaging, distribution, and installation of files in a AIX-based system.

- The **teamcpak** command syntax that will execute subcommands or scripts for the package, distribute, and install functions.

```
teamcpak -i -o "-a -n -t" softdist Client.lst
```

- The **Client.pkf** file you create containing keywords and parameters for distributing and packaging functions.

```
(DATA
(PACKAGEFORMAT softdist)
(TARGETROOT /user/local/teamc/images)
(MANAGER Distribi)
(NODES perlovrs tcaix02)
(INSTALLPGM /tivoli/fpTeamcAIX/tcinstall.ksh)
(LOGNODE tcaix00)
(LOGFILE /tmp/fpTeamcAIX.log)
)
```

- The **Client.lst** file you create containing the list of files passed to **teamcpak**. The first line contains the package file by convention. The example also contains customized installation files (**tcinstall.ksh**), TeamConnection tar files, and an installation script (**tcinst.ksh**).

```
Client.lst;
/usr/teamc/tivoli/Client.pkf
/usr/teamc/tivoli/tcinstall.ksh
/tcinstall/v208/fullpak/aix4/tar/client.tar
/tcinstall/v208/fullpak/aix4/tar/msgen_us.tar
/tcinstall/v208/fullpak/tcinst.ksh
```

The following presents an example of a Tivoli installation script (**tcinstall.ksh**) that is copied to the target along with the tar files and the TeamConnection installation script (**tcinst.ksh**), then executed on the target.

```
#!/bin/ksh
# Clear existing log

INST_DIR=/usr/local/teamc
INST_TMP=${INST_DIR}/tcinstl.tmp
INST_OUT=${INST_DIR}/tcinstl.out
INST_ERR=${INST_DIR}/tcinstl.err
INST_LOG=${INST_DIR}/tcinstl.log
IMAGE_DIR=${INST_DIR}/images

rm -f $INST_ERR $INST_OUT $INST_LOG $INST_TMP >/dev/null 2>&1
mkdir -p ${IMAGE_DIR}

exec 1>${INST_ERR}
exec 2>&1
exec 3>${INST_LOG}

# Install TeamConnection using responsefile
print -u3 Starting TeamC installation at 'date'
print -u3 User id= 'id'
print -u3 Input: $*

# Set up installation environment
# - assumes bourne or korn shell
grep OS_ROOTDIR '/.profile'
if (($? != 0))
then
    print -u3 Updating /.profile
    exec 4>>/.profile
    cd /
    print -u3 'ObjectStore and TeamConnection settings'
    print -u4 'OS_ROOTDIR=/usr/lpp/ODI/OS4.0'
    print -u4 'export OS_ROOTDIR'
    print -u4 'PATH=$PATH:$OS_ROOTDIR/cset/bin'
    print -u4 'export PATH'
    print -u4 'LIBPATH=$LIBPATH:$OS_ROOTDIR/common/lib'
    print -u4 'export LIBPATH'
    . /.profile
else
    print -u3 /.profile already updated
fi

# Set up error logging
# - if *.warning is in file (preceded by spaces and tabs only)
grep "^[ ]*\.warning" /etc/syslog.conf
if (($? != 0))
then
    print -u3 'Updating /etc/syslog.conf'
    touch /var/spool/syslog
    chmod 666 /var/spool/syslog
    exec 4>> /etc/syslog.conf
    print -u4 '*.warning /var/spool/syslog'
    stopsrc -s syslog
    startsrc -s syslog
else
    print -u3 /etc/syslog.conf already updated
```

```

fi

# Update services file for tcocto family
grep "tcocto" /etc/services
if (($? != 0))
then
    print -u3 Updating /etc/services
    exec 4>> /etc/services
    print -u4 'tcocto 8888/tcp'
else
    print -u3 /etc/services already updated
fi

# Generate response file
###
### Change to use enviroment variables!!
###
print -u5 '1'
print -u5 '5'
print -u5 '/usr/local/teamc/images'
print -u5 '/usr/local/teamc'
print -u5 '/usr/local/teamc/nls'
print -u5 'en_US'
print -u5 '/usr/local/teamc/X11'
print -u5 ''
print -u5 'i'

# Run provided TeamC install script
ls -laR ${IMAGE_DIR} >> ${INST_TMP} 2>&1
cd ${IMAGE_DIR}
${IMAGE_DIR}/tcinst.ksh < ${IMAGE_DIR}/tcinst1.response
if (($? != 0))
then
    # Failed installation
    print -u3 TeamC installation failed
    exit 1
else
    # Clean up installation directory after listing contents
    print -u3 We have successfully copied TeamC installation files
    print -u3 Installation directory contents:
    ls -laR ${INST_DIR} >> ${INST_TMP} 2>&1
fi

cd /
# Remove installation stuff
print -u3 TeamC cleaning up temporary installation directory
rm -rf ${IMAGE_DIR}
cat ${INST_TMP} >> ${INST_LOG}
rm -rf ${INST_TMP}
exit 0

# end of file

```

Appendix A. Environment Variables

You can set environment variables to describe the TeamConnection environment in which you are working. You are not required to set your TC_FAMILY environment variable for the TeamConnection client command line interface. However, if the TC_FAMILY environment variable is not set, the -family must be specified for every client command. See “Setting environment variables” on page 207 for more information about setting environment variables.

The names of the TeamConnection environment variables, the purpose they serve, the equivalent TeamConnection flag, the equivalent Settings notebook field, and the TeamConnection component that uses the environment variable are listed in the following table.

You can override the value you set for an environment variable by using the corresponding flag in a TeamConnection command. When an environment variable has a Settings notebook equivalent, TeamConnection uses the two as follows:

- The environment variable controls the command line interface.
- The Settings notebook controls the graphical user interface.

If there is no Settings notebook equivalent for the environment variable, then the environment variable takes effect regardless of the interface you are using.

To see a list of current environment variable settings, you can issue the following command from a command prompt:

```
teamc report -testServer
```

This command returns information like the following:

```
Connect to Family Name:      ptest
Server TCP/IP Name:         amachine.company.com
Server IP Address:          9.1.23.45
Server TCP/IP Port Number:  9999
```

```
Server Specific Information -----
Product Version:            3.0.0
Operating System:           AIX
Message catalog language:   English
Server Mode:                non-maintenance
Authentication Level:        HOST_ONLY
TC_RELEASE|v300
TC_FAMILY|ptest@amachine.company.com@9999
```

Table 1. TeamConnection environment variables

Environment variable	Purpose	Flag	Setting	Used by
LANG	Specifies the language-specific message catalog.			Client, family server
NLSPATH	Specifies the search path for locating message files.		NLS path	Client, family server
OS_NETWORK	Specifies the networking protocols used by ObjectStore servers and clients. Set during installation.			ObjectStore

Table 1. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
OS_ROOTDIR	Specifies the path where the TeamConnection server is installed. Set during installation.			ObjectStore
OS_TMPDIR	Specifies where ObjectStore is to place temporary files. Set during installation. This variable is used only in Windows 32s environments.			ObjectStore
PATH	Specifies where tcadmin is to search for the family create utilities.			Family server
TC_ALLOWTRACKFIX	Allows users to add works areas in fix state to drivers.			Family server
TC_BECOME	Identifies the user ID you want to issue TeamConnection commands from, if the user ID differs from your login. You assume the access authority of the user ID you specify.	-become	Become user	Client, build server
TC_BUILDENVIRONMENT	Specifies the build environment name, such as OS/2 or MVS. The value you specify here can be anything you like, but it must exactly match the environment specified for a builder in order for the builder to use this build agent. This value is case-sensitive.	-e		Client
TC_BUILDOPTS	Specifies build options for sending build log file messages to the screen, and setting the logging level. Possible values are TOSCREEN, and VERBOSE. If you do not specify any of these options, then the build server writes build messages to the build log file (teamcbld.log), and writes a minimum level of messages to the log file.	-C, -S, -I		Build server
TC_BUILDPOOL	Specifies the build pool name.	-pool	Pool	Build server
TC_BUILD_RSSBUILDS_FILE	Specifies the name of startup files to be used to provide information about build servers to the build process.			Build server

Table 1. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_BULKCHUNKSIZE	Identifies the size, in bytes, of the storage blocks used to store binary parts in ObjectStore. Do not use, unless you are trying to tune database performance with guidance from IBM service.			Family server
TC_CASESENSE	Changes the case of the arguments in commands, not in queries.		Case	Client
TC_CATALOG	Specifies a specific file for the TeamConnection message catalog. Sometimes, depending upon the operating system environment, the catalog open command will only look in a particular directory for the catalog. If the host is running multiple versions of TeamConnection, this variable may be used. To set this environment variable, specify the file path name of the message catalog as in the following example: TC_CATALOG="/family/msgcat/teamc.cat"			Family server
TC_COMPONENT	Specifies the default component.	-component	Component	Client, make import tool
TC_DBPATH	Specifies the database directory path.			Family server
TC_FAMILY	Identifies the TeamConnection family you work with.	-family	Family	Build server, client, family server, make import tool
TC_MAKEIMPORTRULES	Specifies the name of the rules file that TeamConnection uses when importing the makefile data into TeamConnection. If you set this environment variable, then you do not have to use the /u option with the fhomigmk command. Specify the full path name of the rules file. If neither this environment variable nor the /u option is used, TeamConnection uses default rules.			Make import tool

Table 1. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_MAKEIMPORTTOP	Strips off the leading part of the directory name when importing parts into TeamConnection. For example, you have parts with the following directory structure: g:\octo\src\inc\. To create these parts without the g:\octo structure, you can set TC_MAKEIMPORTTOP=g:\octo before you invoke the make import tool. The parts created in TeamConnection have the directory structure of src\inc\.			Make import tool
TC_MAKEIMPORTVERBOSE	Causes the -verbose flag to be added to part commands created by fhomigmk.			Make import tool
TC_MIGRATERULES	Specifies the name of a file containing the rules to be applied for migration of makefiles if the name is not supplied on the fhomigmk command line as a parameter.			Client
TC_NOTIFY_DAEMON	An alternate way of starting notifyd with the teamcd command. If you set this environment variable, then you do not have to use the -n option with the teamcd command. Specify the full path name of the mail exit to use with notifyd.			Family server
TC_RELEASE	Specifies a release.	-release	Release	Client, make import tool
TC_SYSTEM_LOG	Specifies where syslog messages are to be written. Specify the full path name of the file to use for syslog messages. The default is syslog.log.			Family server
TC_TOP	Specifies the source directory.	-top	Top	Client
TC_TRACE	Specifies the variable that lets the user designate which parts should be traced. You should modify this only when directed to do so by an IBM service person. Otherwise it is set to null. To trace all parts, specify TC_TRACE=*.			Client, family server, build server

Table 1. TeamConnection environment variables (continued)

Environment variable	Purpose	Flag	Setting	Used by
TC_TRACEATTEMPTS	Specifies maximum number of failed trace attempts accepted before giving up. You should modify this only when directed to do so by an IBM service person.			Client, family server
TC_TRACEDELAY	Specifies the amount of time, in seconds, that TeamConnection waits, when a trace attempt fails, before attempting another trace. The default is 1 second. You should modify this only when directed to do so by an IBM service person.			Client, family server
TC_TRACEFILE	Specifies the output (part path and name) of the trace that the user designates using TC_TRACE. The default trace file name is tctrace. For the MVS build server, the default trace file is stdout.			Client, family server
TC_TRACESIZE	Specifies the maximum size of the trace file in bytes. If the maximum is reached, wrapping occurs. The default is one million bytes.			Client, family server
TC_USER	Specifies the user login ID for single-user environments OS/2, Windows 3.1, and Windows 95. This environment variable is not used in multiuser environments AIX, HP-UX, Solaris, and Windows NT.		User ID	Client, build server
TC_WORKAREA	Specifies the default work area name.	-workarea	Work area	Client, make import tool
TC_WWWPATH	Specifies the path for the HTML helps and image files for Web client.	-workarea	Work area	Client, family server

Setting environment variables

For methods of setting your environment variables, refer to your operating system documentation. For example, for OS/2 you can use the following command to set the TC_FAMILY environment variable:

```
SET TC_FAMILY=familyName@hostname@portnumber
```

Appendix B. Importing makefile information into TeamConnection

TeamConnection provides a command to help you import makefile information into the TeamConnection database. The `fhomigmk` command reads a makefile and creates the parts in it. Build tree connections are created based on a rules file. The command syntax of the `fhomigmk` command is:

```
fhomigmk /m [makefile]  
         /f [family]  
         /r [release]  
         /w [work area]  
         /c [command file]  
         /u [rules file]  
         /x  
         /s  
         /k
```

You can precede the parameter with either a slash (/) or a dash (-).

The parameters are defined as follows:

/m [*makefile*]

The name of the makefile you want to import into TeamConnection. If you do not specify this parameter, TeamConnection uses `makefile`.

/f [*family*]

The name of the TeamConnection family into which the makefile data will be imported. If not specified, TeamConnection uses the value of the `TC_FAMILY` environment variable. If the value of `TC_FAMILY` is not defined, the value `none` is used.

/r [*release*]

The name of the TeamConnection release into which the makefile data will be imported. If not specified, TeamConnection uses the value of the `TC_RELEASE` environment variable. If the value of `TC_RELEASE` is not defined, the value `none` is used.

/w [*work area*]

The name of the TeamConnection work area into which the makefile data will be imported. If not specified, TeamConnection uses the value of the `TC_WORKAREA` environment variable. If the value of `TC_WORKAREA` is not defined, the value `none` is used.

/c [*command file*]

The name of the command file that will be produced and saved. If this file already exists, commands created by the specified makefile are appended to the existing contents.

/u [*rules file*]

The name of the rules file that TeamConnection will use when importing the makefile data into TeamConnection. If not specified, TeamConnection uses the value of the `TC_MAKEIMPORTRULES` environment variable. If no rules file is found, TeamConnection uses default rules. "Creating a rules file" on page 210 explains the rules, the format of this file, and the default rules.

/x

Specifies that you want to run the command file that was produced by the `/c` parameter.

- /s** Specifies that the build tree is to be displayed after the command is issued. If specified, the command file is run even if the `/x` parameter is not specified.
- /k** Specifies that you want TeamConnection not to erase the intermediate files it uses to process this command. This might be useful in debugging problems that arise during the import. However, in general, you will not specify this parameter. When specified, the following intermediate files are saved:

modified makefile

A modified form of the imported makefile. The command invocations (of things like linkers and compilers) are replaced by calls to a TeamConnection command that captures dependency data. To find the cause of import errors, type the following command at an OS/2 command line:

```
nmake -f mod_make
```

create file

A list of all the objects referenced by the makefile that should be created in the TeamConnection database.

connect file

A list of all the objects referenced by the makefile that should be connected to other objects in the TeamConnection database. Each line contains one dependency relationship in the format `<child> <parent>`.

TeamConnection provides an environment variable, `TC_MAKEIMPORTTOP`, that when set strips off the leading part of the directory name. For example, you have parts with the following directory structure: `g:\octo\src\inc\`. Because you want the parts created without the `g:\octo` structure, you set `TC_MAKEIMPORTTOP=g:\octo` before you invoke the make import tool. The parts created in TeamConnection have the directory structure of `src\inc\`.

Another environment variable provided by TeamConnection, `TC_MAKEIMPORTVERBOSE`, when set causes the `-verbose` flag to be added to part commands.

The following is an example of invoking the make import tool:

```
fhomigmk /m Mymak /w mywork /s /u myrules
```

In this example, the makefile called `mymak` is used to create a temporary command file containing TeamConnection commands. The commands are formed based on the rules defined in the file `myrules`. The family and release used in the commands are those specified in the environment variables `TC_FAMILY` and `TC_RELEASE`. The work area used in the commands is `mywork`. After the commands are issued, the resulting build tree is shown using the TeamConnection GUI.

Creating a rules file

The import rules file is a text file that describes how you want TeamConnection to create and connect parts. In this file you supply a set of rules, one per line, using the following syntax:

file mask

The mask specifying the names of the files to which this rule applies. The * and ? wildcards are supported. For example, you could specify file names such as *.cbl, abc*.cpp, or foo\src*.obj.

type

The type of contents of the files to which the rule applies when they are stored in TeamConnection as a part. Allowed values are binary, text, none, or ignore. If you specify ignore as the file type, then all files that match the file mask are bypassed.

builder

The name of the TeamConnection builder to be associated with the part. The builder is not created for you. If you specify a builder, it must exist in TeamConnection before you run fhomigmk. A value of none means that no builder will be associated with the part.

parser

The name of the TeamConnection parser to be associated with the part. The parser is not created for you. If you specify a parser, it must exist in TeamConnection before you run fhomigmk. A value of none means that no parser will be associated with the part.

connect

How the part will be connected to other parts in TeamConnection. The following values are allowed:

- input
- output
- dependent
- none

When none is specified, the part is not connected to another part even though a dependency was found for the part in the make file. For example, when you indicate none for a file mask of *.h files, the *.h files are created in TeamConnection, but not connected to the files that include them. The value you will use most often is input.

content

Where the initial content of the part can be found:

- none indicates that the part is initially created as *empty*.
- directory\ indicates to concatenate with the name of the file in the makefile. This is where the contents are expected to be found.
- * indicates to use the name in the makefile, relative to the current working directory.

For example, if a makefile specifies a file src\abc.cbl and the makefile specifies f:\mysrc\, the content is expected in f:\mysrc\src\abc.cbl. For a file of *.cbl, the content is expected in src\abc.cbl relative to the current working directory.

parameters

The build parameters to be attached to the part. Enclose the parameter in double quotes if it has spaces. Use the value none to indicate no parameters.

component

The TeamConnection component that will contain the part. If none is specified, the value of the TC_COMPONENT environment variable is used.

As TeamConnection processes each part referenced in the makefile, it looks for a rule that matches the part name. If a match is found, the rule is used. The rules are searched from top to bottom. The first matching rule is used.

Comments are denoted by a pound sign (#) in the first column.

Columns are separated by spaces.

A sample rules file, called fhomigmk.rul, is supplied with TeamConnection. Use this file to help you create a rules file that is appropriate for your development environment.

The following is a simple example of an import rules file:

```
<top of file>
#-----
# file mask  type    builder  parser  connect  content  parameters  component
#-----
#   *.exe      binary  linker   none    input    none     /Debug      ship
#   *.obj      binary  icc       none    input    none     /Ti+        objects
#   *.cpp      text    none      cplus   input    *        none        source
#   *.h*       text    none      cplus   none     *        none        source
<end of file>
```

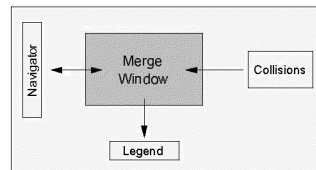
If you do not specify a rules file in the /u parameter of the fhomigmk command, TeamConnection uses the value of the TC_MAKEIMPORTRULES environment variable. If no rules file is found, TeamConnection uses the following default rules:

```
<top of file>
#-----
# file mask  type    builder  parser  connect  content  parameters  component
#-----
#   *.*       text    none      none    none     none     none        root
```

Appendix C. TeamConnection Merge

TeamConnection provides a tool that you can use to merge two or three selected files together to make one single file. With the TeamConnection VisualMerge tool, you can select options for viewing differences and collisions, as well as view the composite output of the merged files. Both a command line and a graphical user interface are provided.

VisualMerge combines changes made to a part by more than one developer into a single part. The figure below describes the merge of three files into a single file. Items one through six represent the main stream of development for a particular part or file. Item 2 represents the base or common file from which two streams proceed. At the time of the merge, the most recent version in the two streams are items 4 and 2.3. In this scenario, the user selects the base or common file, item 2, as input 1. The user also uses item 4, the latest version in development, as input 2. The latest version in the branch is represented as item 2.3, or input 3. Item 5 represents the outcome of the merge.



Once the merge is complete a window appears with the three original files and the final merged file.

Appendix D. Enabling a Workframe project for TeamConnection

TeamConnection lets you create a Workframe version 3 project that has TeamConnection options as well as a set of TeamConnection actions. For each project, you specify on the Project Options window the values for these options. By doing this, you logically connect a Workframe project with a set of TeamConnection parts. This makes it easier for you to perform TeamConnection actions, such as checking parts in and out, directly from the Workframe.

Creating a TeamConnection-enabled Workframe project

Follow these steps to create a Workframe project that is enabled for TeamConnection.

1. On an OS/2 command line, type the following command and press Enter:

```
photcini.cmd
```

This command creates a TeamConnection **Project Smarts** catalog on your desktop. (If you have already created this catalog, there is no need to perform this step again for additional projects.)

2. Open the TeamConnection **Project Smarts** catalog. Select the TeamConnection project, and select the **Create** pushbutton.
3. Specify the location for the TeamConnection project you want to create; then select **OK**.

When the action completes, you will see a TeamConnection Project on your desktop.

Setting up your project options

Options are provided so that you can set up each TeamConnection Workframe project. To set the options, do the following:

1. Select **Tools Setup** from the project's Views pull-down menu.
2. Select the **Project Options** or **File Options** menu from any of the TeamConnection actions.

The following options are provided:

Family

The TeamConnection family.

Release

The TeamConnection release.

Work area

The TeamConnection work area in which you will perform TeamConnection actions.

Query mask

Any valid TeamConnection -where clause for parts. Leave blank to see all parts. (This is used in the project's **Show Parts** action.)

Show filter

Check this if you want to display the PartFull Filter window instead of using the query mask in the **Show Parts** action.

Profile

Names the rules file to use for the **Import Make** action. Specify the fully qualified name unless you are sure it will be found in your path correctly. Select the **Find** push button if you need help.

Using your TeamConnection Workframe project

You can perform a set of TeamConnection actions from within your project:

- “Project actions” lists the actions you can perform without selecting a part.
- “Part actions” lists the actions you can perform against a selected TeamConnection part.

Project actions

Extract part

Displays an unprimed Extract Parts window.

Checkout part

Displays an unprimed Check Out Parts window.

Checkin part

Displays an unprimed Check In Parts window.

Unlock part

Displays an unprimed Unlock Parts window.

Lock part

Displays an unprimed Lock Parts window.

Create part

Displays an unprimed Create Parts window.

Build part

Displays an unprimed Build Parts window.

View part contents

Displays an unprimed View Part Contents window.

View part information

Displays an unprimed View Part Information window.

Edit part

Displays an unprimed Edit Part window.

Show parts

If the project attribute **Show filter** is not set, issues a query based on the project attribute's query mask. If the project attribute **Show filter** is set, displays the PartFull Filter window.

Part actions

Extract part

Displays the TeamConnection Extract Parts window to extract the selected part.

Checkout part

Displays the TeamConnection Check Out Parts window to check out the selected part to the work area specified in the project options.

Checkin part

Displays the TeamConnection Check In Parts window to check in the selected part to the work area specified in the project options.

Unlock part

Displays the TeamConnection Unlock Parts window to unlock the selected part.

Lock part

Displays the TeamConnection Lock Parts window to lock the selected part.

Create part

Displays the unprimed TeamConnection Create Parts window.

Build part

Displays the TeamConnection Build Parts window to start a build of the selected part.

View part contents

Displays the TeamConnection View Part Contents window for the selected part.

View part information

Displays the TeamConnection View Part Information window for the selected part.

Edit part

Displays the TeamConnection Edit Part window for the selected part.

Import makefile

Imports the information contained in the selected makefile into the TeamConnection database.

The **Import makefile** action is restricted to files with the extension .mak. The other actions in this list apply to files of all types.

Using your project: a simple scenario

Suppose you are working on a defect in the family FAMILY1, release REL1_1. You have created a TeamConnection work area called SANDBOX to work in. You want to use the Workframe to access your TeamConnection parts. Here is what you might do.

1. Create a TeamConnection Workframe project called DefectABC.
2. Open the project. Select **Tools setup** from the View pull-down menu.
3. Select any of the actions. Press mouse button 2 to display the context menu; then select **Project options** or **File options** from the context menu. The result is a window in which you can specify the TeamConnection information about the project.
4. Specify the family FAMILY1, the release REL1_1, and the work area SANDBOX. Check the **Show filter** check box. Select **OK**.
5. Specify the general Workframe attributes of the project using the project's Settings notebook. These attributes include information such as the location of the OS/2 files for the project. For example, in this scenario, you specify that you want this project to contain all files in the directory c:\defect_abc, which is initially empty.
6. Select **TeamConnection → Show files** from the Project context menu. The PartFull Filter window is displayed. Specify the filter criteria; then select **OK**. For

example, specify that you want to see all the parts with extensions .cpp, .exe, and .hpp. The Parts window is displayed.

7. Select the parts client.exe, server, client.cpp, client.hpp, server.cpp, and server.hpp. Select **Extract** from the context menu.
8. On the Extract Parts window, type c:\defect_abc in the **Target directory** field and select **OK**. Now you can interact with these parts directly from the Workframe project.
9. In the Workframe project, run the ipmd.exe debugger until you determine the cause of the problem. Suppose you find the bug is in client.cpp.
10. Go back to the Parts window. Select **client.cpp** from the list of parts, and select **TeamConnection → Checkout part** from the context menu for the object. The part is checked out to the SANDBOX work area.
11. Edit the file to fix the problem; then select **TeamConnection → Checkin part** to check the part back into SANDBOX, the TeamConnection work area from which it was checked out.
12. Build the part by selecting **TeamConnection → Build part** on the context menu for the file client.exe.
13. When the build completes, extract the resulting executable by selecting **TeamConnection → Extract part** from the file's context menu.
14. Run the executable to verify that the problem has been fixed.

Appendix E. Enabling and Using the ENVY/Manager-TeamConnection Bridge

Overview of the ENVY/Manager-TeamConnection Bridge

ENVY provides a repository with operational support tailored specifically for highly-interactive, prototyping environments that emphasize iterative development, such as VisualAge Smalltalk or VisualAge Generator. A bridge from ENVY to TeamConnection provides access to the powerful software configuration management (SCM) support provided by ENVY, along with the scalable, enterprise-level support provided by TeamConnection. TeamConnection's ability to manage all development artifacts (not just source code), to share information in a common model, and to integrate multiple tools and multiple languages across the enterprise on a single baseline extends the capabilities of software development groups. The ENVY/Manager-TeamConnection Bridge (also referred to as **the bridge** in this documentation) will provide essential integration for VisualAge tools which use ENVY as their day-to-day operational library.

VisualAge Generator Version 3.0 has access to the TeamConnection-ENVY Bridge through VisualAge Smalltalk, which can interface directly with ENVY/Manager. The bridge supports the import and export of VisualAge Generator objects (*parts*) to and from TeamConnection.

ENVY/Manager provides a collaborative component development environment for application development and integration using fine-grained object languages, such as Smalltalk. The ENVY repository is designed for languages that run on the universal virtual machine (uVM). The repository includes persistence, versioning, and configuration management.

TeamConnection can be used to manage artifacts (*parts*) that need to be shared with non-uVM based languages or tools for purposes of build management, problem tracking, and other configuration management functions. These artifacts can be exported to the TeamConnection server through the ENVY/Manager-TeamConnection Bridge and stored as TeamConnection parts.

ENVY objects stored in a TeamConnection database can be queried and retrieved back into the ENVY/Manager development environment as needed. The units of storage in TeamConnection include exported ENVY components (such as applications and configuration maps) and large grained objects (files). Small-grained objects, such as VisualAge Generator data items, are imported and exported as constituents of applications. The data items in an application are exported to TeamConnection in an array that makes their definitions available to other tools through the data model.

Scope of this documentation

This documentation is intended for users and administrators installing and using the bridge. It is assumed that you are familiar with both the VisualAge Smalltalk and TeamConnection products.

The following subsections describe the mechanics of enabling the ENVY/Manager-TeamConnection Bridge for VisualAge Smalltalk Pro (Version 4.0 or later), the process of exporting ENVY components to TeamConnection, and the process of importing these components back into ENVY/Manager. See the

VisualAge Generator documentation for tool-specific details. TeamConnection information related to change tracking and build processing are addressed in the TeamConnection documentation.

Many terms used by VisualAge Smalltalk and TeamConnection are problematic because the tools may define these terms differently. *Release* and *component* are typical examples. To avoid any ambiguity, such terms may precede the name of the tool they are applied to, such as *TeamConnection release*.

Description of the ENVY/Manager-TeamConnection Bridge

Basic functionality

It makes sense to describe the functionality of the bridge from the perspective of a Smalltalk developer, because it is through the Smalltalk image that the user drives the bridge. The bridge is an import/export facility for three types of entities:

- Smalltalk configuration maps
- Smalltalk applications
- Files residing on local and networked file systems that are accessible through the image

The bridge allows a Smalltalk developer to store any of these entities in a TeamConnection database and retrieve them at a later time. As is the case when exporting to other ENVY/Manager libraries, configuration maps and applications must be versioned before they can be exported. This enforces the notion that the Smalltalk developer uses the bridge and TeamConnection to maintain baselines rather than for managing work-in-progress.

Developers use ENVY/Manager's fine-grained support to facilitate the process of shared development in open editions of components on a daily basis. At appropriate junctures, components are versioned and promoted to TeamConnection, where together with other project elements, they form a baseline across an entire project. The resulting baseline may contain objects such as program elements, file, and metadata.

From the perspective of the TeamConnection user or administrator, the bridge allows the Smalltalk image to function as a TeamConnection client, storing and retrieving parts in a TeamConnection family database.

How the bridge communicates with TeamConnection

The bridge functions are initiated from within the VisualAge Smalltalk environment. Each operation that interacts with TeamConnection runs for some time in the Smalltalk image, but at some stage will make use of functions built into an appropriate version of the TeamConnection client and server. The bridge itself is implemented in Smalltalk, with the primitive functions provided in one of the DLLs in TeamConnection.

The unit of transfer used by the bridge for Smalltalk components is an ENVY/Manager library. Each library stored in TeamConnection contains one of the following:

- a Smalltalk application and its released subapplications (and their released subapplications, and so forth)
- a configuration map without any of its released applications

Note: Subapplications cannot be exported through the bridge without an enclosing application.

ENVY/Manager libraries are stored in TeamConnection databases as TeamConnection parts. When the bridge exchanges a library with TeamConnection, the target in a TeamConnection database is specified by a TeamConnection context. A TeamConnection context is comprised of the following TeamConnection parameters:

- Family name
- Release name
- Work area name

Note: Each TeamConnection context can contain only one version of any named application or configuration map. This is unlike ENVY/Manager libraries, in which multiple versions of a named Smalltalk component can co-exist.

The bridge is aware of the various relationships between ENVY/Manager components. When an application is transferred through the bridge, all of its released subapplications are transferred with it. When a configuration map is transferred through the bridge, the bridge will also transfer the released applications in separate operations. Depending on a user-specified setting, the bridge can also transfer required maps of configuration maps.

Preparing to use the ENVY/Manager-TeamConnection Bridge

The bridge is delivered as a configuration map suitable for loading into a VisualAge Smalltalk Version 4.0 (or later) image. The library, TCEMBR.DAT, will contain the configuration map **ENVY/Manager-TeamConnection Bridge** and, for VisualAge Generator build support, **VAGen ENVY/TC Bridge**.

These configuration maps should be imported into your development library so that it can be loaded by all of the users sharing that library. The step-by step instructions are described in “Installing and activating the ENVY/Manager-TeamConnection Bridge” on page 222.

Usually, the **Library Supervisor** or the first user to use the bridge will perform this operation and then inform other users that the tool is available in the library.

The sections that follow describe the steps necessary to set up the bridge and verify that it is functional.

Setting up the bridge environment

The following information is especially pertinent to the individual(s) responsible for bridge setup and administration.

Prerequisites

Before the bridge will work, you must have the following:

- VisualAge Smalltalk Pro Version 4.0 or later installed. See the VisualAge Smalltalk documentation to confirm that you have the appropriate hardware and software prerequisites available.
- A TeamConnection server that is running.

- A TeamConnection GUI client installed on the machine where you are running your Smalltalk image.

Note: This release of the bridge only runs on OS/2 and Windows platforms.

You should verify that you are able to communicate with the relevant TeamConnection server by using the TeamConnection GUI client. If you cannot communicate with the TeamConnection server in this manner, the bridge will definitely not function correctly.

Only certain releases of TeamConnection support the bridge. If you have received the bridge with your release of TeamConnection, you probably have a matching version. If not, then you may have to upgrade your release of TeamConnection. The DLL TCEMBR.DLL should be available to programs in your environment, because this is the DLL that contains the primitives used by the bridge.

Environment variables

The bridge relies on the user to specify the various parameters that make up the TeamConnection context. By default, the bridge will query the variables in the environment that the image is running. These variables, **TC_FAMILY** , **TC_RELEASE** , and **TC_WORKAREA**, are used as initial values for the default TeamConnection context.

There are two additional environment variables that can be defined for the bridge, as follows:

- **TC_COMPONENT** is used as the default TeamConnection component for parts stored in TeamConnection through the bridge. If **TC_COMPONENT** is not defined or is empty, the value root is used.
- **TC_RELATIVE** is used to specify the initial destination path for files retrieved from TeamConnection through the bridge. If **TC_RELATIVE** is not defined or is empty, the current directory according to the image is used.

It is not necessary to define any of these variables for the bridge to work. Defining them only makes setting up the default bridge configuration in the image easier for a bridge user.

A system administrator may want to have the environment variables automatically defined in a network login script. When a user logs into a LAN and then uses the bridge, the user will be provided with the defined values as hints for setting up the default TeamConnection configuration.

Installing and activating the ENVY/Manager-TeamConnection Bridge

The bridge is loaded into the image like any other configuration map using the **Load** option from the **Editions** menu of the **Configuration Maps Browser**.

Once the bridge is loaded, the submenu **TeamConnection Bridge** will appear on the **Tools** menu of the **System Transcript** window. This submenu is referred to as the *bridge menu*. The bridge menu is the launching point for all of the bridge operations.

“Loading the ENVY/Manager-TeamConnection Bridge” on page 223 provides step-by-step instructions for the bridge loading process.

Loading the ENVY/Manager-TeamConnection Bridge

Follow these steps to load the ENVY/Manager-TeamConnection Bridge:

1. Open the VisualAge for Smalltalk Pro - Client.
2. Go to the **System Transcript** window and select **Browse Configuration Maps** from the **Tools** pulldown menu.
3. In the Configuration Maps Browser window, select Import from the **Names** pulldown menu.
4. A dialog will prompt you to enter the full path name of the library that you want to import. For purposes of activating the bridge, you will need to supply a TeamConnection pathname (determined by where you have installed TeamConnection) for the file called TCEMBR.DAT. Select the **OK** pushbutton to continue and display the **Selection Required** window.
5. In the **Selection Required** window, select ENVY/Manager-TeamConnection Bridge in the **Names** list, which will prime the **Versions** list with a version number.
6. Select the version in the **Versions** list and move it to the **Selected Versions** list using the right-arrow pushbutton.
7. For the additional interoperability with VisualAge Generator described in “Using the ENVY/Manager-TeamConnection Bridge: a simple scenario for VisualAge Generator developers” on page 230, you must also import the configuration map called VAGen ENVY/TC Bridge, as described in the previous steps.
8. Select the **OK** pushbutton to initiate the import process. During the process of importing the TCEMBR.DAT file into the VisualAge Smalltalk Pro manager.dat file, the **System Transcript** window will issue a message stream that confirms the success of the import.
9. In the **Configuration Maps Browser** window, select ENVY/Manager-TeamConnection Bridge from the **Names** list.
10. Select the item (there should only be one available) in the Editions and Versions list.
11. Select all of the items in the **Applications** list, click mouse button 2, and select **Load** from the pop-up menu.

Note: For the additional interoperability with VisualAge Generator described in “Using the ENVY/Manager-TeamConnection Bridge: a simple scenario for VisualAge Generator developers” on page 230, you must also load VAGen ENVY/TC Bridge, as described in the two previous steps. ENVY/Manager-TeamConnection Bridge *must be loaded first*.

12. After the application loading progress dialog completes without errors, the ENVY/Manager-TeamConnection Bridge should be functional. You can close the **Configuration Maps Browser** window at this time.

Testing the ENVY/Manager-TeamConnection Bridge

To verify that the bridge is active and ready for ENVY component export/import functions, follow these steps:

1. Go to the **System Transcript** window and select the **Tools** pulldown menu. Then select **Default Properties** from the **TeamConnection Bridge** cascade menu. This will display the **Default Properties** notebook.
2. Verify that the TeamConnection family in the **Family** field on the **Context** page of the **Default Properties** notebook is appropriate for your project. You may need to coordinate your access to the family with your family administrator.

3. Select the **Test Server** pushbutton. If the bridge is properly configured, the server connection test will return an information window that provides server-specific information. Select the **OK** pushbutton to dismiss the server information window.

You are now ready to export ENVY components to a TeamConnection server.

Note: When exiting VisualAge for Smalltalk Pro - Client, you should save your image so that the ENVY/Manager-TeamConnection Bridge will be preserved for future use.

Using the ENVY/Manager-TeamConnection Bridge

You can perform TeamConnection functions on ENVY components, provided that you supply parameters necessary to identify a *bridge configuration*. Bridge configuration parameters are defined by the **Default Properties** notebook, as described in “Setting default properties”.

Each time the bridge interacts with TeamConnection, it uses the parameters in a bridge configuration to ensure that the behavior of the operation is in accordance with the users’ specifications. Because specifying a configuration for each operation would be time-consuming and most operations would use the same configuration, you can specify a default configuration. Each time the user initiates an operation, you can use the default configuration or modify it.

The default configuration is stored in the image so that once it is setup, it will be maintained until the bridge is reloaded from the library.

Setting default properties

To set properties for import and export actions across the ENVY/Manager-TeamConnection Bridge, open the **Default Properties** notebook as described in “Testing the ENVY/Manager-TeamConnection Bridge” on page 223. The **Default Properties** notebook contains four pages of settings, as follows:

- Context page
- Operations page
- Import page
- Export page

Each page of the **Default Properties** notebook includes the following controls:

Show this dialog when exporting and importing checkbox

The **Show this dialog when exporting and importing** checkbox specifies whether the dialog should be shown each time an import/export operation for the bridge is initiated by the user. If the dialog is shown, it gives the user the opportunity the default configuration for a particular operation only.

push buttons

OK Saves the current settings as default setting. This option may not be available if some fields are left incomplete or contain invalid values.

Cancel

Closes the **Default Properties** notebook and ignores any changes made in the dialog.

Defaults

Updates the dialog fields with the values in the current default bridge configuration.

Reset Updates the dialog fields with the initial values that are set when the bridge is first loaded into the image.

Context page

The context page is used to specify the TeamConnection family, release, and work area used as the context for the default bridge configuration.

Family

Use this field to input the name of your TeamConnection family server. Select the **Test Server** pushbutton to return an information window that provides server-specific information. If you cannot successfully communicate with the TeamConnection server, you may have specified an invalid family name. Your TeamConnection family administrator may be of some assistance at this point.

Release

The TeamConnection release. By selecting the **Query releases** pushbutton, you can prime the **Release** field drop-down menu with valid release choices based on the **Family** field value.

Work area

The TeamConnection work area in which you will perform TeamConnection actions. By selecting the **Query work areas** pushbutton, you can prime **Work area** field drop-down menu with valid work area choices based on the **Release** field value.

Note: Any communication with a TeamConnection server takes time. Querying the available releases and work areas typically takes a few seconds, which is the reason that this data is not automatically used to populate the dialog.

Operations page

The **Operations** page determines how operations are performed in TeamConnection, including whether operations are forced and how parts in the database are locked.

Force The **Force** and **Don't force** radio buttons are mutually exclusive.

In TeamConnection terms, *force* is an indication that changes should be forced into the TeamConnection repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the specified version might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not specified.

The force option is important only if you specify that a part version is to be locked. If you want to retrieve or store a locked part in a particular release or work area that is linked to another release or work area, you might want to specify the force option when you are checking in or checking out the

part, even if someone else might have the part checked out in another context. See the discussion of locking below for a description of TeamConnection locking options.

Locking

These mutually-exclusive radio buttons enable you to instruct TeamConnection cache services (TCCS) on how to manage the locking behavior for parts that you are exporting to or importing from the TeamConnection repository.

Obtain and release

Also known as *optimistic* locking, TCCS will attempt to check out the part(s) before checking in changes that you have made in the ENVY environment. If this action is successful, the part(s) will not be locked in TeamConnection after the export.

Obtain and retain

TCCS will attempt to check out the part(s) before checking in changes that you have made in the ENVY environment. If this action is successful, the part(s) will remain locked in TeamConnection after the export.

Retain

For parts already locked in TeamConnection, after changes are exported from ENVY the locked parts should remain locked (i.e., the lock is *retained* by the original owner).

Release

For parts already locked in TeamConnection, after changes are exported from ENVY the locked parts should be unlocked, and therefore available to other developers in that context.

Import page

The **Import** page provides default settings options when importing ENVY components or files previously exported to a TeamConnection database.

Configuration Maps

If the **Import all required maps too** checkbox is checked, it specifies that when a configuration map is imported, its required configuration maps (along with any other required configuration maps, recursively) should be retrieved from TeamConnection as part of the import action. If this option is enabled, and a configuration map being imported does have required maps, the maps can only be imported if they actually exist in the TeamConnection database.

Note: The checkbox is checked as the default.

Destination for Files

The **Destination path for files** field identifies the target (base) directory for imported files.

Replacing Existing Files

These mutually-exclusive radio buttons enable you to select a desired default method for overwriting files (or not) in your working target directory.

Ask user

This choice enables you to choose which files are to be overwritten.

Do not replace existing files

Files that currently exist in the target directory will not be overwritten.

Replace existing files

Files that currently exist in the target directory are automatically overwritten.

Export page

The **Export** page provides default settings options for exporting ENVY components or files to a TeamConnection database.

Storage in TeamConnection

The **Component** field identifies the TeamConnection target component for your export action. This component designation, along with TeamConnection family, release, and work area values supplied in the **Context** page of the **Default Properties** notebook, is necessary to define the context for any new TeamConnection parts created by an export action.

Configuration Maps

If the **Export all required maps too** checkbox is checked, it specifies that when a configuration map is exported, its required configuration maps (along with any other required configuration maps, recursively) should be exported TeamConnection. If this option is enabled, and a configuration map being exported does have required maps, the maps can only be exported if they actually exist in the TeamConnection database.

This option is used to prevent version mismatches when a configuration map requires other configuration maps, as in the following case:

1. For a configuration map that requires other configuration maps, you do an export *with* the required maps.
2. At some later time, you export again *without* the required maps.
3. When you attempt to import *with* the required maps, the import may fail, because a configuration map level in TeamConnection does not match the level previously exported from ENVY.

Note: The checkbox is checked as the default.

Exporting ENVY components to TeamConnection

The **TeamConnection Bridge** cascade menu provides an **Export** choice, which offers the following choices:

- Configuration Maps
- Applications
- Files

Note: You must have the appropriate authority to update all parts associated with the configuration maps, applications, or files to be exported.

As a general rule, it is advisable to export applications and configuration maps along with any configuration maps required by these ENVY components to avoid version mismatches. If you make a change to an application, it is important to update all the exported configuration maps that contain the application and to export all of the configuration maps again.

Note: The **Export all required maps too** checkbox located on the **Export** page of the **Default Properties** notebook defaults to this behavior.

The following describes two simple cases in which a mismatch might occur:

1. Export a configuration map that contains several applications.
2. Make a change to one of the contained applications.
3. Export the changed application only.
4. Attempt to import the configuration map.

or

1. Export two configuration maps that contain the same application.
2. Make a change to the common application and export only one of the configuration maps that contains the application.
3. Attempt to import the second configuration map.

As the number of programmers authorized to version components and the complexity of your applications increase, so does the possibility for these types of problem to occur. Therefore, it is important to coordinate update authority in such a way that all affected parties are notified about configuration changes, and that someone in the development group has authority over all levels of components. It may also be advisable to limit export actions to higher levels of authority than you have previously.

Exporting components is a substantial operation that typically takes at least ten to twenty seconds (possibly minutes for a large collection of components). Such an operation begins with the bridge exporting the components to temporary ENVY/Manager libraries and then generating detailed descriptions of the library contents for the benefit of TeamConnection. To guarantee atomicity and minimize the number of times that the bridge must communicate with TeamConnection (thus avoiding unnecessary overheads), all components are transferred in one primitive operation.

Even a single configuration map usually counts as more than one component, because it typically contains at least one release application. Once the primitive operation is invoked, control of the process is in the TeamConnection client code, which is effectively blocked against the TeamConnection server. Because the Smalltalk image is blocked waiting for the primitive to return, the user interface will not update, and the user cannot halt the operation.

Exporting configuration maps and applications

The process for exporting ENVY-based configuration maps and applications to a TeamConnection family database includes the following steps:

1. Select **Configuration Maps** or **Applications** from the **Export** cascade menu. You will be prompted to select an appropriate version of the configuration map or application that you want to export.

Note: ENVY components must be versioned in ENVY before being exported to TeamConnection.

2. In the **Selection Required** window, select the desired configuration map or application in the **Names** list, which will prime the **Versions** list with a version number.
3. Select the version in the **Versions** list and move it to the **Selected Versions** list using the right-arrow pushbutton. Because only one version of any named

configuration map can exist in a TeamConnection context, it is only possible to choose one version for any particular name.

4. Select the **OK** pushbutton to initiate the export process.
5. If the **Show this dialog when exporting and importing** option has been set in the default bridge configuration, you will be presented with the **Export Properties** notebook, which is primed by values in the **Default Properties** notebook. If you are satisfied with the current values in the **Export Properties** notebook, select the **OK** pushbutton to initiate the export process.

If the export succeeds without errors, a message is logged to the **System Transcript** window. Users are informed of any errors with a message box.

Exporting files

The process for exporting ENVY-based files to a TeamConnection family database includes the following steps:

1. Selecting **Files** from the **Export** cascade menu.
2. You will be prompted to select the files that you want to export. You can add to or delete files from the list using the **Add**, **Remove**, or **Remove All** pushbuttons.
3. Select the **OK** pushbutton to initiate the export process.
4. If the **Show this dialog when exporting and importing** option has been set in the default bridge configuration, you will be presented with the **Export Properties** notebook, which is primed by values in the **Default Properties** notebook. If you are satisfied with the current values in the **Export Properties** notebook, select the **OK** pushbutton to initiate the export process.

If the export succeeds without errors, a message is logged to the **System Transcript** window. Users are informed of any errors with a message box.

Importing ENVY components from TeamConnection

The **TeamConnection Bridge** cascade menu provides an **Import** choice, which offers the following choices:

- Configuration Maps
- Applications
- Files

The process for importing any of these ENVY components is essentially the same, and includes the following steps:

1. Select **Configuration Maps**, **Applications**, or **Files** from the **Import** cascade menu.
2. If the **Show this dialog when exporting and importing** option has been set in the default bridge configuration, you will be presented with the **Import Properties** notebook, which is primed by values in the **Default Properties** notebook.
3. When you are satisfied with the current values in the **Import Properties** notebook, select the **OK** pushbutton.
4. You will be prompted to supply a query pattern to further reduce the number of candidates for import. Select the **OK** pushbutton to launch the query of the TeamConnection context that you have specified up to this point.

Note: Use the wildcard characters (*) and (?) as delimiters for your queries.

5. A list of ENVY components matching your query is returned. Each of these components exists in a TeamConnection database specified by the TeamConnection context in the configuration used for this operation. Select the objects you want to import from this list and select the **OK** pushbutton to initiate the import action.
6. In the case of configuration maps and applications, the selected components will be imported into the default ENVY/Manager library that the image is connected to. For files, the selected files will be written to the path specified by the **Destination path for files** option in the bridge configuration used for this operation. If any of the files already exist, they may be overwritten, or the user may be prompted, depending on the value of the **Replace existing files** option.

If the import succeeds without errors, a message is logged to the **System Transcript** window. Users are informed of any errors with a message box.

Using the ENVY/Manager-TeamConnection Bridge: a simple scenario for VisualAge Generator developers

The following scenario is a generalized case used to illustrate the way that VisualAge Generator developers might use the ENVY/Manager-TeamConnection Bridge to accomplish change tracking and build processing. An actual implementation requires that a Smalltalk development team begin with versioned ENVY components and a plan for sharing common applications.

After you have installed the ENVY/Manager-TeamConnection Bridge, you can export a versioned configuration map to a TeamConnection family database. For VisualAge Generator developers, this means that you can generate programs, tables, and map groups using the TeamConnection build interface. See the VisualAge Generator *Generator's Guide* and "Part 3. Using TeamConnection to build applications" on page 91 in this document for details.

Scenario assumptions

For purposes of describing the scenario, the following assumptions are established:

- A development team using VisualAge Generator wants to perform problem tracking and build generation.
- A family is created in TeamConnection with a release r1, defined with a track-driver process (i.e., all part changes are made in reference to work areas).
- A build agent and its corresponding build processor has been started to handle build requests for generation, and similarly for preparation.
- Data item definitions and records used to access data in a database are kept in a "common" application, while programs and their other associates are kept in a separate application.

Note: This assumption enables the scenarios to include application prerequisites.

Exporting ENVY components to TeamConnection

To prepare for exporting the ENVY components to TeamConnection, perform the following activities:

1. **In TeamConnection:**

- a. Create a feature called f1 and accept the feature.
 - b. Create a work area called wa1 for implementation of the feature.
2. **In ENVY:**
 - Create applications for the common data and for other VisualAge Generator parts.
 3. **In VisualAge Generator Developer:**
 - Create programs and their associates.
 - Create generation option, linkage table, resource association, link edit, and bind parts as necessary.
 4. **In ENVY:**
 - a. Create a configuration map that gathers the common data application and the application containing all the other parts.
 - b. The class developers version their classes.
 - c. The class owners release versioned classes into the two applications and the application managers version the applications.
If more than one developer has been working on the feature, each may have opened a new edition of a part's class extension, so a merge of the method editions will have to be performed.
 - d. The configuration map manager releases the versioned applications into the configuration map and versions the configuration map.
 - e. An administrator uses the ENVY/Manager-TeamConnection Bridge to export the configuration map to the work area associated with feature f1. See "Exporting ENVY components to TeamConnection" on page 227 for ENVY/Manager-TeamConnection Bridge export instructions.
After the ENVY/Manager-TeamConnection Bridge export action is completed, there will be an EmLibrary part for the configuration map and for each of its applications, and proxy parts for each entry in each application's BOM file. The BOM file for each application contains an entry (at least the name, edition/timestamp, and TCPart type) for each class and method in each application.

Object mapping in TeamConnection

After a ENVY/Manager-TeamConnection Bridge export action, the following parts are created in TeamConnection in wa1 for f1 in the component and release specified as context for the export action:

- For each application of the configuration map there will be an application part.
- For each entry in the BOM, a proxy part with a name qualified by the application name for uniqueness, as described in Table 2 on page 232.

The ENVY/Manager-TeamConnection Bridge must map ENVY components to part names in TeamConnection in such a way that the parts can be retrieved in a reusable form when they are imported from TeamConnection back into the ENVY environment.

Table 2. Name generation mapping for the ENVY/Manager-TeamConnection Bridge

Class Type	Naming Convention	Mapped Name Example
EmLibrary	<class_name_of_blob_object>.<name_of_blob_object>	EmApplication.MyApp, EmConfigurationMap.MyConfigMap
EmConfigurationMap	<config_map_name>	MyConfigMap
EmApplication	<application_name>	MyApp
EmSubapplication	<app_name>.<subapp_name>	MyApp.MySubapp
EmClass OR EmClassExtension	<app_name>.<subapp_name>.<class_name>	MyApp.MyClass, MyApp.MySubapp.MyClass
EmInstanceMethod OR EmClassMethod	<app_name>.<class_name>.<method_name> OR <app_name.subapp_name>.<class_name>.<method_name>	MyApp.MyClass.MyMethod, MyApp.MySubapp.MyClass.MyMethod

See the VisualAge Generator *Generator's Guide* for additional information related to generation part output names in TeamConnection.

Build generation

The VisualAge Generator *Generator's Guide* provides detailed VisualAge Generator build generation instructions. The following overview is provided to place these activities in the context of the ENVY/Manager-TeamConnection Bridge:

1. In VisualAge Generator Developer:

- For each program, the Options Override (OVR) part that has been exported to TeamConnection creates an initial build tree for VisualAge Generator applications in TeamConnection.

Refer to the VisualAge Generator *Generator's Guide* for more details on the OVR part.

2. In TeamConnection (build function):

- The build administrator selects the EZEPREP collector of the initial build tree of a program proxy in wa1 for f1, and requests a build.
- TeamConnection places the generator build event on the build queue, and the generator build agent detects a new build event that it can service.
- The build processor invokes the generator build script, which parses the name of the generation configuration map name from the generated part's build parameters.
- The build script invokes the generator, which imports the configuration map and its references to the generation ENVY manager. The ENVY manager used by generation is identified by a VisualAge INI file on the generation build server.

The ENVY/Manager-TeamConnection Bridge determines whether each application referenced already exists in the generation Envy manager, and only imports an application if that version of the application is not already in the manager.

Note: For VisualAge Generator builds, you can use the environment variable TC_ENVY_REFRESH to control when VisualAge Generator builders will import the required configuration map from TeamConnection. TC_ENVY_REFRESH can be used to affect the following behaviors:

- If TC_ENVY_REFRESH=null, the configuration map will not be imported from TeamConnection if that version of the configuration map is already in the connected Envy Manager.
- If TC_ENVY_REFRESH=null, the configuration map will be imported from TeamConnection if that version of the configuration map is not already in the connected Envy Manager.
- If TC_ENVY_REFRESH=notnull, the configuration map will always be imported from TeamConnection. This setting is important if you use VisualAge DataAtlas to modify data elements that are dependents of the configuration map. In that case, if the data elements have been modified since the configuration map was exported to TeamConnection, the builder will warn you that the configuration map is not synchronized with its dependent data elements *only if* TC_ENVY_REFRESH=notnull. Such a warning allows you to import the changed data elements into a new edition of the configuration map and export the resulting new version to TeamConnection, before trying the build again

Setting TC_ENVY_REFRESH is only relevant in the environment of the TeamConnection build server that performs the VisualAge generator builds.

- e. The generator uses the ENVY/Manager-TeamConnection Bridge to update the outputs and dependencies in the build tree.
- f. If there are tables and/or map groups used by the program, the generator determines whether there is already a build tree for them. If not, initial build trees are added for them using the program's OVR part.
- g. TeamConnection re-examines the build tree of the EZEPREP collector and determines that new build events have been added to the build scope for preparation of the generation outputs, and possibly for generation and preparation of tables and map groups. Build events are started to complete the preparation of generation outputs, and generation and preparation of tables and map groups if necessary.

Note: See the VisualAge Generator *Generator's Guide* for greater detail on this process.

3. In TeamConnection (change control):

- A project administrator completes the fix record(s) for the feature f1 and adds the work area wa1 to a system test driver. Eventually the driver is committed to the release and the feature is completed.

Making a change to a member

1. In TeamConnection:

- a. Defect d1 is created and accepted in TeamConnection
- b. Work area wa2 is created for the implementation of the defect d1.

2. In ENVY:

- a. An application manager creates new edition of an application that requires a change.
- b. A developer makes a change to one or more parts.
- c. The class developer of the changed parts versions the class, the class owner releases the class into the new edition of the application, and the application manager versions the application. If more than one defect is in progress, the class owner must release only the versions that apply for defect d1.

- d. The configuration map owner opens a new edition of the configuration map used to generate the program being changed, and releases the new application version into the configuration map. The configuration map owner versions the configuration map.
 - e. The administrator uses the ENVY/Manager-TeamConnection Bridge to put the configuration map back into the work area wa2 for defect d1.
3. **In TeamConnection (build function):**
- a. Build administrator builds the program(s) affected by the change. This can be done by selecting the preparation collector for each program and requesting a build, or by selecting a collector for a subsystem, and building the subsystem collector. Only programs, tables, or map group affected by the changes to proxy members will be rebuilt.
 - b. The generation process continues as it did for the initial build (see “Build generation” on page 232 for details), except that there should be no need to add new build trees unless a new table or map group was added to a program being built
4. **In TeamConnection (change control):**
- A project administrator completes the fix record(s) for the defect d1 and adds the work area wa2 to a system test driver. Eventually the driver is committed to the release and the feature is completed.

Appendix F. Source Code Control User's Guide

Differences between other source code control providers and TeamConnection

The purpose of this document is to help Visual Basic, Visual C++, and Power Builder users, make TeamConnection their Visual environments source code control provider. This document assumes the reader is a new user of TeamConnection, but has some familiarity with source code control.

Projects vs Families

Most source code control providers group all code into projects. TeamConnection uses an object oriented approach that provides much more control over the software product while allowing greater flexibility. Projects have one dimension of control. Development environments like Visual Basic group all of their files into projects. Using projects to group source code has several limitations. First, the source code control system is limited to providing just version control. While version control is useful, once the enterprise-size organization is reached, it is often not sufficient to control just versions of the source code. TeamConnection provides not only versioning but defect and feature tracking, build and driver management, access control, and much more. TeamConnection uses families, releases, components, and work areas for management and control.

TeamConnection uses several layers of control. The highest level is the family. The family is the name of the data base, where TeamConnection stores all of the code, the versions, and all other information related to the code. A family represents a complete and self-contained collection of TeamConnection users and development data. Data within a family is completely isolated from data in all other families. One family cannot share data with another. It is important to know the name of the family where TeamConnection will store your code and associated information.

A part in TeamConnection is a collection of data that is stored by the family server. This can include files, text, objects, binary objects, or modeled objects. Parts can be stored by a user, a tool, or generated from other parts, such as when a linker generates an executable file.

Components are used to organize the data in a family. Components are arranged in a hierarchical tree structure, with a single top component called root. The component owns the parts that may be in it, and controls access to the parts. Once you are given access to a component, you have access to all the parts and subcomponents in that component. The component also controls the process that TeamConnection uses, for example, to report and fix a defect. Within each family, development data is organized into groups called components. The component hierarchy of each family includes a single top component, initially called root, and descendants of that root. Each child component has at least one parent component; a child can have multiple parents.

The release is somewhat analogous to a project. A release is a logical grouping of the components that make up a product. An application is likely to contain parts from more than one component. Because you probably want to use some of the same parts in more than one application, or in more than one version of an application, TeamConnection groups parts into releases. A release is a logical organization of all parts that are related to an application; that is, all parts that must

be built, tested, and distributed together. Each time a release is changed, a new version of the release is created. Each version of the release points to the correct version of each part in the release. Each part in TeamConnection is managed by at least one component and contained in at least one release. One release can contain parts from many components; a component can span several releases. Each time a new development cycle begins, you can define a separate release. Each subsequent release of an application can share many of the same parts as its predecessor. You need to know the name of the release.

A work area is basically a view of a release. For example, a work area can be opened for each defect that needs to be fixed. More than one programmer can work in the same work area at the same time. A programmer can have more than one work area active at a time. A release contains the latest integrated version of each of its parts. As users check parts out of the releases, update them, and then check them back in, TeamConnection keeps track of all these changes, even when more than one user updates the same part at the same time.

You need to know the name of the work area in which you will be working. A good practice is to create and name a work area after the defect being addressed in the work area. For example, name work area W1557 for defect 1557. You can create a work area if you have the authority in TeamConnection, but this must be done through the TeamConnection GUI.

For more information about families, releases, components, work areas, parts, and what you can do with them, see your TeamConnection Documentation.

Installing the TeamConnection source code control DLL

Before you can use the integrated support from your development environment you must install TeamConnection and the TeamConnection Source Code Control DLL. If you are using TeamConnection Version 2.08 or later, the source code control DLL is already installed.

Note: If you have not already done so, follow the directions and install the TeamConnection client for your workstation. The following directions assume that you have successfully installed the TeamConnection client.

Connecting TeamConnection to Visual Basic 4.0

If you are using TeamConnection Version 2.08 or later, the source code control add-in for Visual Basic is already included.

Removing the TeamConnection Source Code Control DLL

To change the default source code control system for Visual Basic, change the value in the ProviderRegKey to the registry key of another provider.

To remove TeamConnection, leave the value in the ProviderRegKey blank.

Using TeamConnection as your source code control provider

Once the installation procedure is complete, starting your development environment automatically links the TeamConnection Source Code Control DLL.

Before you start

There are several things you must know before you can start using TeamConnection as your source code control provider. If you are not sure of this information, contact your family administrator. Your family administrator can help you find the following information:

- Family
- Component
- Release
- WorkArea

You also need to know the project name. The project name is used by your development tool to relate to the TeamConnection attributes of family, release, work area, and component by the Source Code Control DLL.

Opening a project

One of the few differences you see when using TeamConnection as your source code control provider occurs when you open a project. When you open a new project, the TeamConnection Source Code Control Settings window opens. At the top of this window is a field with your development project name. In addition to the project name field, there are fields for family, work area, release, and component. If this is a new project, these fields are blank. If this is not a new project, the fields contain the previous values. You can change these values only when this window is open. If at anytime you decide to change any of these values, you must first close the project and reopen it.

Once all the fields are filled in, select **OK**. The project will open. If you select **Cancel**, the source code control system disconnects from the development environment until another project is opened.

Under some versions of Visual Basic, projects automatically close and open after certain operations. This causes this TeamConnection Source Code Control Settings window to open at times when it may appear unnecessary. When this occurs, select **OK**. If you select **Cancel**, you will be left in a state that requires shutting down and restarting Visual Basic to reconnect the source code control system.

Integrated features

Once you open a project you can use the integrated features of the development environment to access your files in TeamConnection. The development environment keeps track of the files that are known to TeamConnection, and the checkout status of each file. For example, the development environment keeps track of files checked out to other users.

The exact steps necessary to perform each of the following actions depend on the development environment being used. However, for a given environment, the steps are the same regardless of the source code control provider. For example, if you check out a file in the Visual C++ development environment when it is connected to Visual SourceSafe, the steps you use are exactly the steps you use when C++ is connected to TeamConnection.

Check-in

The steps to check-in a file vary by the development environment. In most cases pressing mouse button 2 when the mouse pointer is over a file icon of a file checked out to you, brings up a menu that includes the file check-in option. Selecting the file checkin option opens the Check-In window. Checking the **keep checked out** box on the Check-In window sets the keep locked flag, TeamConnection saves the file, but keeps it checked out to you. Selecting **OK** causes the TeamConnection part check-in function to execute and the file is checked in.

Check-out

Similar to check-in, the check-out action can be started by pressing mouse button 2 on the file icon of a file not already checked-out. Check-out calls the TeamConnection Part Check-out function.

Uncheck-out

A checked out file can be unchecked out. Again this action can often be started by right clicking the file icon of a file that is checked out. Uncheck-out calls the part unlock function in TeamConnection.

Get Version

Rather than check out a file, you can also get the latest version of the file. Get Version calls the TeamConnection Part Extract function.

Adding Files to source code control: Adding a file that is not already under source code control places the selected file into the source code control system. Add calls the TeamConnection part create function.

Properties

Selecting **Properties** from a pull-down menu opens the properties GUI. Information that TeamConnection needs to correctly check out and check in parts is provided here. For example, the work area field changes each time an existing work area is integrated and a new work area is created.

Full features of TeamConnection

Most development environments allow you to evoke TeamConnection from the pull-down menus. In Visual Basic, TeamConnection appears as an option in the Add-Ins pull-down menu. In Visual C++, TeamConnection appears in the Source Code Control option of the Tools pull-down menu. From the TeamConnection GUI you can create new work areas (if you have the correct authority), retrieve previous versions of a part, open or process defects, and perform many other actions against parts.

Migrating project data bases

One key issue for programmers and project managers moving from another source code control system to TeamConnection is how to migrate the database of projects. The following describes one way to bring the current level of code for a small to medium sized project into TeamConnection.

Migrating an existing project: The following example illustrates the simplest way to migrate an existing Visual Basic source code control database into TeamConnection. Lets say we were using the ABC source code control system, and we are going to migrate our project, Austin, to TeamConnection. The idea is to extract all the files in Austin using the ABC source code control system, and then add them as parts in TeamConnection. Follow the steps below to perform this migration.

1. Make ABC the default source code provider. To do this, set the registry key ProviderRegKey to point to the registry entry for source code control provider ABC. See "Installing the TeamConnection source code control DLL" on page 236 for more information on how to perform this step. Once you complete this step, ABC will be the Source Code Control provider when we open Visual Basic.
2. Start the Visual Basic development environment.
3. Open project Austin.
4. Extract all the files to your system.
5. Exit the Visual Basic development environment.
6. Edit the registry key ProviderRegKey to be:
SOFTWARE\IBM\TeamConnection\

See "Installing the TeamConnection source code control DLL" on page 236 for more information on how to perform this step. TeamConnection is now the default source code control provider and is attached when the development environment starts.

7. Restart the Visual Basic development environment.
8. Open project Austin again.
9. When the TeamConnection Source Code Control Settings window opens it will have Austin listed as the project. Fill in the values for family, release, component, and work area, then select **OK**.
10. Add the files to TeamConnection following the steps in the Visual Basic development environment.
11. Repeat these steps until all of your projects are migrated to TeamConnection.

Starting a new project: Starting a new project in Microsoft Visual Basic or Visual C++ is essentially the same regardless of the source code provider. The only operational difference is that the TeamConnection Source Code Control Settings window opens at some point. When the TeamConnection Source Code Control Settings window opens, enter the names of your family, component, work area, and release.

Starting Visual Basic: To create a new project under Visual Basic, do the following:

1. Start Visual Basic
2. Create and save a new project
3. Select the **Add Project to TeamConnection** option from the TeamConnection option in the Add-Ins pull-down menu. The TeamConnection Source Code Control Settings window opens. Fill in the family, release, component, and work area then select **OK**.
4. The Add To TeamConnection window opens. Select the files you want to add. Type a comment in the comment field. Visual Basic requires that a comment be entered. Select **OK**.

Starting Visual C++: To create a new project under Visual C++, do the following:

1. Start the Visual Developers Studio as normal.
2. On the File pulldown, select New. A new window will open.
3. On the New window select Project Workspace, then **OK**.
4. The New Project Workspace window will open. On the New Project Workspace window, do the following:
 - a. Select the type of project
 - b. Type a name
 - c. Select create.
5. The TeamConnection Source Code Control Settings window will open. On the TeamConnection Source Code Control Settings window, enter the family name, release, component, and work area. Then, select **OK**.
6. Files can now be added to the project using the Insert menu pull-down.
7. To place the files under source code control, select the add to source code control option of the Tools menu pull-down.

Appendix G. Supported keywords

TeamConnection supports keywords in text files. When a file containing keywords is extracted from TeamConnection, the current value of each keyword is added to the file. This information can help you identify what version of source code is used for your deliverables. .p .TeamConnection supports the following keywords.

Keyword	Description
\$ChkD;	The time and date stamp applied during check in.
\$FN;	The file name complete with its path.
\$KW=@(#);	The start of keyword expansion.
\$EKW;	Keyword expansion is ended until the next \$KW keyword.
\$Own;	The user ID of the owner of the component that manages the part.
\$Ver;	Identification necessary to locate the part in TeamConnection, such as family, release, or component.

The following examples show lines of code that change in a text file as a user extracts a part. The text file used in this example is filex.hdr.

```
#ifndef_filex_hdr_

#define_filex_hdr_

    static char=_filex_hdr[]="$KW=@(#); $FN=mtdkyws.ide; $Ver=tcid30:1; $ChkD=1997/01/24 11:43:08";

#endif
```

TeamConnection ignores keywords until it finds a \$KW keyword. It then expands all keywords until a \$EKW keyword is found. If the semicolon (;) following a keyword is omitted, the keyword is not expanded.

No change occurs when the part is checked in to TeamConnection. However, when the part is extracted, the keyword variables are updated. The following example shows how the keywords are expanded.

```
#ifndef_filex_hdr_

#define_filex_hdr_

    static char=_filex_hdr[]="$KW=@(#) $FN=bin/filex.hdr ; $Ver=V1.1 ;

$ChkD=93/04/06 18:13:19 ;";

#endif
```

In the previous example, each keyword and its value appears in the output. The value of the keyword is replaced each time the part is extracted. If you do not want the keyword to appear in the output, add a minus sign (-) after the dollar sign (\$). For example:

```
static char=_filex_hdr[]="$KW=@(#); mtdkyws.ide tcid30:1 1997/01/24 11:43:08";
```

Be aware that if a file is extracted, then locked and checked in, the version information can no longer be updated because the keyword does not appear in the output.

Appendix H. Authority and notification for TeamConnection actions

TeamConnection ships with IBM-supplied authority groups, interest groups, component processes, and release processes. Your family administrator can modify these preconfigured authority groups, interest groups, and processes to fit the needs of your organization.

Each authority group consists of actions normally performed by a particular type of user. Your family administrator can modify these groups or create new ones to reflect the needs of your organization.

Authority groups provide explicit authority to perform the actions included in each group. You might also have implicit authority to perform certain actions according to the objects that you own. Authority groups are defined in a file called `authorit.Id`.

To determine your authority groups, from the Actions pull-down menu, select Lists → Access lists → Show authority actions. On the Show authority actions window select an action.

Each notification group consists of actions normally of interest to a particular type of user. Your family administrator can modify these groups or create new ones to reflect the needs of your organization. Interest groups are defined in a file called `interest.Id`.

To determine your interest notification groups, from the Actions pull-down menu, select Lists → Notification lists → Show interest actions. On the Show authority actions window select an action.

The following table lists all of the TeamConnection actions, the required level of implicit and explicit authority to perform the action, and the users who are notified when an action is performed. To explicitly assign authority to a user, add the user's ID to a component's access list.

Note: The user who performs the action is excluded from the notification that is sent out after the action is successfully completed.

For this action	These users have authority	These users are notified
AccessCreate	<ul style="list-style-type: none">• Component owner• Explicitly defined for the component where access is being added	User being given new access, subscribers
AccessDelete	<ul style="list-style-type: none">• Component owner• Explicitly defined for the component where access is being altered	User whose access was deleted, subscribers
AccessRestrict	<ul style="list-style-type: none">• Component owner• Explicitly defined for the component where access is being restricted	User whose access is being restricted, subscribers

For this action	These users have authority	These users are notified
ApprovalAbstain	<ul style="list-style-type: none"> Approval record owner Explicitly defined for the component that manages the associated release 	Approval record owner, subscribers
ApprovalAccept	<ul style="list-style-type: none"> Approval record owner that manages the associated release 	Approval record owner, subscribers
ApprovalAssign	<ul style="list-style-type: none"> Approval record owner Explicitly defined for the component that manages the associated release 	New and original approval record owners, subscribers
ApprovalCreate	<ul style="list-style-type: none"> Work area owner Explicitly defined for the component that manages the associated release 	New approval record owner, subscribers
ApprovalDelete	<ul style="list-style-type: none"> Explicitly defined for the component that manages the associated release 	Approval record owner, subscribers
ApprovalReject	<ul style="list-style-type: none"> Approval record owner Explicitly defined for the component that manages the associated release 	Approval record owner, subscribers
ApproverCreate	<ul style="list-style-type: none"> Release owner Explicitly defined for the component that manages the associated release 	New approver, subscribers
ApproverDelete	<ul style="list-style-type: none"> Release owner Explicitly defined for the component that manages the associated release 	Deleted approver, subscribers
BuilderCreate	<ul style="list-style-type: none"> Explicitly defined for the component that manages the associated release 	Subscribers
BuilderDelete	<ul style="list-style-type: none"> Explicitly defined for the component that manages the associated release 	Subscribers
BuilderExtract	<ul style="list-style-type: none"> Explicitly defined for the component that manages the associated release 	Not applicable
BuilderModify	<ul style="list-style-type: none"> Explicitly defined for the component that manages the associated release 	Subscribers

For this action	These users have authority	These users are notified
BuilderView	<ul style="list-style-type: none"> Explicitly defined for the component that manages the associated release 	Not applicable
CollisionAccept	<ul style="list-style-type: none"> Component owner Explicitly defined for the component that manages the associated release 	Release owner, subscribers
CollisionReconc	<ul style="list-style-type: none"> Component owner Explicitly defined for the component that manages the associated release 	Release owner, subscribers
CollisionReject	<ul style="list-style-type: none"> Component owner Explicitly defined for the component that manages the associated release 	Release owner, subscribers
CompCreate	<ul style="list-style-type: none"> Parent component owner Explicitly defined for the parent component 	New component owner
CompDelete	<ul style="list-style-type: none"> Component owner Explicitly defined for the component being removed 	Component owner, subscribers
CompLink	<ul style="list-style-type: none"> Component owner of the component being linked Explicitly defined for the component being linked 	Owners of both components, subscribers
CompModify	<ul style="list-style-type: none"> Component owner Explicitly defined for the component being modified 	New component owner if applicable, subscribers
CompRecreate	<ul style="list-style-type: none"> Parent component owner Explicitly defined for the parent component 	Owners of both components, subscribers
CompUnlink	<ul style="list-style-type: none"> Component owner of the component being unlinked Explicitly defined for the component being unlinked 	Owners of both components, subscribers
CompView	<ul style="list-style-type: none"> Component owner Explicitly defined for the component being viewed 	Not applicable
CoreqCreate	<ul style="list-style-type: none"> Work area owner of all specified work areas Explicitly defined for the component managing the associated work area and release 	Not applicable

For this action	These users have authority	These users are notified
CoreqDelete	<ul style="list-style-type: none"> • Work area owner of all specified work areas • Explicitly defined for the component associated with the release 	Not applicable
DefectAccept	<ul style="list-style-type: none"> • Defect owner for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectAssign	<ul style="list-style-type: none"> • Defect owner, defect originator • Explicitly defined for the component associated with the defect <p>Note: Originators who do not have DefectAssign authority can reassign the defect only when it is in the open state.</p>	New owner, defect originator, duplicate defect originators, subscribers
DefectCancel	<ul style="list-style-type: none"> • Defect originator • Explicitly defined for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectClose	Automatic action; no authority is required	Defect owner, defect originator, duplicate defect originators, subscribers
Not applicable; this is a base authority that can be performed by all users in the family	Defect owner, defect originator, duplicate defect originators, subscribers	
DefectDesign	<ul style="list-style-type: none"> • Defect owner • Explicitly defined for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectModify	<ul style="list-style-type: none"> • Defect owner can modify: <ul style="list-style-type: none"> – answer, abstract, environment, driver, prefix, reference, release, and all configurable fields <p>Defect originator can modify:</p> <ul style="list-style-type: none"> – originator, severity, name, abstract, environment, driver, prefix, reference, release, and all configurable fields <ul style="list-style-type: none"> • Explicitly defined for the component associated with the defect, these users can modify: <ul style="list-style-type: none"> – abstract, answer, name, environment, driver, originator, prefix, reference, release, severity, phaseFound*, phaseInject*, priority*, symptom*, and target* <p>*If these fields have been configured by the family administrator, the field names might differ from those shown.</p>	Defect owner, defect originator, duplicate defect originators, subscribers

For this action	These users have authority	These users are notified
DefectOpen	Not applicable; this is a base authority that can be performed by all users in the family	Component owner, subscribers
DefectReopen	<ul style="list-style-type: none"> Defect originator Explicitly defined for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectReturn	<ul style="list-style-type: none"> Defect owner Explicitly defined for the component associated with the defect 	Defect originator, duplicate defect originators, subscribers
DefectReview	<ul style="list-style-type: none"> Defect owner Explicitly defined for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectSize	<ul style="list-style-type: none"> Defect owner Explicitly defined for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectVerify	<ul style="list-style-type: none"> Defect owner Explicitly defined for the component associated with the defect 	Defect owner, defect originator, duplicate defect originators, subscribers
DefectView	<ul style="list-style-type: none"> Defect owner, defect originator Explicitly defined for the component associated with the defect 	Not applicable
DriverAssign	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	New owner, subscribers
DriverCheck	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Not applicable
DriverCommit	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Subscribers
DriverComplete	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Subscribers

For this action	These users have authority	These users are notified
DriverCreate	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Subscribers
DriverDelete	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Subscribers
DriverExtract	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Not applicable
DriverFreeze	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
DriverModify	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
DriverRefresh	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Component owner, subscribers
DriverRestrict	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
DriverView	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Not applicable
EnvCreate	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Tester, subscribers
EnvDelete	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Subscribers

For this action	These users have authority	These users are notified
EnvModify	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Tester, subscribers
FeatureAccept	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureAssign	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	New owner, feature originator, duplicate feature originators, subscribers
FeatureCancel	<ul style="list-style-type: none"> Feature originator Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureClose	Occurs automatically; no authority is required	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureComment	Not applicable; this is a base authority that can be performed by all users in the family	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureDesign	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureModify	<ul style="list-style-type: none"> Feature owner can modify: <ul style="list-style-type: none"> abstract, prefix, reference, and all configurable fields Feature originator can modify: <ul style="list-style-type: none"> abstract, name, prefix, reference, and all configurable fields Explicitly defined for the component associated with the feature, these users can modify: <ul style="list-style-type: none"> abstract, name, originator, prefix, reference, priority*, and target* <p>*If these fields have been configured by the family administrator, the field names might differ from those shown.</p>	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureOpen	Not applicable; this is a base authority that can be performed by all users in the family	Component owner, subscribers

For this action	These users have authority	These users are notified
FeatureReopen	<ul style="list-style-type: none"> Feature originator Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureReturn	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureReview	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureSize	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureVerify	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Feature owner, feature originator, duplicate feature originators, subscribers
FeatureView	<ul style="list-style-type: none"> Feature owner Explicitly defined for the component associated with the feature 	Not applicable
FixActive	<ul style="list-style-type: none"> Fix record owner, component owner, work area owner Explicitly defined for the component associated with the fix record 	Subscribers
FixAssign	<ul style="list-style-type: none"> Fix record owner, component owner, work area owner Explicitly defined for the component associated with the fix record 	New fix record owner, subscribers
FixComplete	<ul style="list-style-type: none"> Fix record owner, component owner, work area owner Explicitly defined for the component associated with the fix record 	Subscribers
FixCreate	<ul style="list-style-type: none"> Defect or feature owner, work area owner Explicitly defined for the component associated with the defect or feature 	Subscribers

For this action	These users have authority	These users are notified
FixDelete	<ul style="list-style-type: none"> Defect or feature owner, work area owner Explicitly defined for the component associated with the defect or feature 	Subscribers
HostCreate	<ul style="list-style-type: none"> Owner of the user ID for which a host list entry is being created or deleted Superuser 	Not applicable
HostDelete	<ul style="list-style-type: none"> Owner of the user ID for which a host list entry is being deleted Superuser 	Not applicable
MemberCreate	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
MemberCreateR	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
MemberDelete	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
MemberDeleteR	<ul style="list-style-type: none"> Driver owner Explicitly defined for the component associated with the release 	Driver owner, subscribers
NotifyCreate	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the notification list 	Not applicable
NotifyDelete	<ul style="list-style-type: none"> Component owner Owner of user ID Explicitly defined for the component associated with the notification list <p>Note: Users can delete themselves from a notification list without requiring any authority</p>	Not applicable
ParserCreate	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Subscribers

For this action	These users have authority	These users are notified
ParserDelete	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Subscribers
ParserModify	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Subscribers
ParserView	<ul style="list-style-type: none"> Explicitly defined for the component associated with the release 	Not applicable
PartAdd	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the part 	Subscribers
PartBuild	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the part 	Subscribers
PartCheckIn	<ul style="list-style-type: none"> User who checked out or locked the part originally, component owner Explicitly defined for the component associated with the part <p>Note: The user who is explicitly given this authority can check in a part that is checked out by someone else.</p>	Subscribers
PartCheckOut	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the part 	Subscribers
PartChildInfo	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the part 	Not applicable
PartConnect	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the part 	Subscribers
PartDelete	<ul style="list-style-type: none"> Component owner Explicitly defined for the component associated with the part 	Subscribers

For this action	These users have authority	These users are notified
PartDeleteForce	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartExtract	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Not applicable
PartForceIn	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartForceOut	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartLink	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartLock	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartLockForce	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartModify	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartRecreateForce	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartRecreate	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers

For this action	These users have authority	These users are notified
PartRefresh	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartRename	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartRenameForce	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartResolve	Not applicable; this is a base authority that can be performed by all users in the family	Not applicable
PartTouch	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartUndo	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartUndoForce	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Subscribers
PartUnlock	<ul style="list-style-type: none"> • User who checked out or locked the part originally, component owner • Explicitly defined for the component associated with the part 	Subscribers
PartView	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Not applicable
PartViewMsg	<ul style="list-style-type: none"> • Component owner • Explicitly defined for the component associated with the part 	Not applicable
PrereqCreate	<ul style="list-style-type: none"> • Work area owner of all specified work areas • Explicitly defined for the component managing the associated work area and release 	Not applicable

For this action	These users have authority	These users are notified
PrereqDelete	<ul style="list-style-type: none"> Work area owner of all specified work areas Explicitly defined for the component managing the associated work area and release 	Not applicable
ReleaseCreate	<ul style="list-style-type: none"> Explicitly defined for the component associated with the new release 	New release owner, component owner, subscribers
ReleaseDelete	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Release owner, component owner, subscribers
ReleaseExtract	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Not applicable
ReleaseLink	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Release owner, subscribers
ReleaseModify	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release <p>Note: To identify a new component to manage the release, you must have ReleaseCreate in an authority group in the component that you are modifying</p>	Release owner, subscribers, new owner (if applicable)
ReleasePrune	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Subscribers
ReleaseRecreate	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Release owner, component owner, subscribers
ReleaseView	<ul style="list-style-type: none"> Release owner Explicitly defined for the component associated with the release 	Not applicable
Report	Not applicable; this is a base authority that can be performed by all users in the family	Not applicable

For this action	These users have authority	These users are notified
SizeAccept	<ul style="list-style-type: none"> Sizing record owner Explicitly defined for the component associated with the sizing record 	Subscribers
SizeAssign	<ul style="list-style-type: none"> Sizing record owner Explicitly defined for the component associated with the sizing record 	New sizing record owner, defect/feature owner, subscribers
SizeCreate	<ul style="list-style-type: none"> Defect/feature owner Explicitly defined for the component associated with the defect/feature 	Component owner, defect/feature owner, subscribers
SizeDelete	<ul style="list-style-type: none"> Defect/feature owner Explicitly defined for the component associated with the defect/feature 	Subscribers, sizing record owner, defect/feature owner
SizeReject	<ul style="list-style-type: none"> Sizing record owner Explicitly defined for the component associated with the sizing record 	Subscribers
TestAbstain	<ul style="list-style-type: none"> Test record owner Explicitly defined for the component associated with the test record's release 	Subscribers
TestAccept	<ul style="list-style-type: none"> Test record owner Explicitly defined for the component associated with the test record's release 	Subscribers
TestAssign	<ul style="list-style-type: none"> Test record owner Explicitly defined for the component associated with the test record's release 	New test record owner, subscribers
TestReady	<ul style="list-style-type: none"> Test record owner Explicitly defined for the component associated with the test record's release 	Subscribers
TestReject	<ul style="list-style-type: none"> Test record owner Explicitly defined for the component associated with the test record's release 	Subscribers
UserCreate	Superuser	New user
UserDelete	Superuser	Not applicable

For this action	These users have authority	These users are notified
UserModify	<ul style="list-style-type: none"> Owner of the user object can modify all characteristics except the superuser privilege Must be a superuser to grant the superuser privilege 	Not applicable
UserRecreate	Superuser	Not applicable
UserView	Not applicable; this is a base authority that can be performed by all users in the family	Not applicable
VerifyAbstain	<ul style="list-style-type: none"> Verification record owner Explicitly defined for the component associated with the verification record's defect or feature 	Subscribers
VerifyAccept	<ul style="list-style-type: none"> Verification record owner Explicitly defined for the component associated with the verification record's defect or feature 	Subscribers
VerifyAssign	<ul style="list-style-type: none"> Verification record owner Explicitly defined for the component associated with the verification record's defect or feature 	New verification record owner, subscribers
VerifyReady	Takes place automatically; no authority is required	Verification record owners
VerifyReject	<ul style="list-style-type: none"> Verification record owner Explicitly defined for the component associated with the verification record's defect or feature 	Subscribers
WorkAreaAssign	<ul style="list-style-type: none"> Work area owner Explicitly defined for the component associated with the release 	New work area owner, subscribers
WorkAreaCancel	<ul style="list-style-type: none"> Defect or feature owner Explicitly defined for the component associated with the defect or feature 	Subscribers, owners of approval records for work area being canceled
WorkAreaCheck	<ul style="list-style-type: none"> Work area owner Explicitly defined for the component associated with the release 	Not applicable
WorkAreaCommit	<ul style="list-style-type: none"> Work area owner Explicitly defined for the component associated with the release 	Subscribers

For this action	These users have authority	These users are notified
WorkAreaComple	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Subscribers
WorkAreaCreate	<ul style="list-style-type: none"> • Defect or feature owner • Explicitly defined for the component associated with the defect or feature 	Work area owner, subscribers
WorkAreaFix	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Subscribers
WorkAreaFreeze	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Subscribers
WorkAreaIntegra	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Subscribers
WorkAreaModify	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Subscribers
WorkAreaRefresh	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Work area owner, subscribers
WorkAreaTest	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Subscribers
WorkAreaUndo	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	
WorkAreaView	<ul style="list-style-type: none"> • Work area owner • Explicitly defined for the component associated with the release 	Not applicable

Appendix I. Sample REXX execs, build scripts, and parsers

This appendix is composed of the IBM-supplied REXX execs, build scripts, and parsers. Your family administrator can modify these samples to fit the needs of your organization.

The samples in this appendix may not be available on all platforms. Refer to the readme file for a complete list of samples available with TeamConnection. All samples are provided as-is and any use of or modifications to the samples are the sole responsibility of the customer.

Sample REXX execs

This section lists the sample REXX execs that are shipped with TeamConnection. The client.smp file contains this same listing. It is located in the bin subdirectory of the directory where the TeamConnection client is installed.

Users running these execs must have user and host access to your TeamConnection family.

Most of the execs require input parameters, and some require that the TC_FAMILY or TC_RELEASE environment variables be set. If the user who is running the script is acting for another user, the TC_BECOME environment variable must also be set. These variables can be set from a command line prompt.

The following convention is used to show the required, optional, and selective input parameters:

- Brackets ([]) indicate that the input or variable is optional.
- Braces ({ }) indicate that one of the inputs is required.
- An input or variable that is not surrounded by brackets or braces is required.

Script name	Function	Inputs	Environment variables
accComp	Lists the explicit access of users in a specified component.	componentName	TC_FAMILY [TC_BECOME]
compChld	Lists the direct children of a specified component. Also lists the description and owner of each child component.	componentName	TC_FAMILY [TC_BECOME]
compOwnr	Displays a list of component owners' addresses in a specified family.	familyName	[TC_BECOME]
compPrnt	Lists the parent component of a specified component.	componentName	TC_FAMILY [TC_BECOME]
compWalk	Displays the children and grandchildren of a specified component.	componentName	TC_FAMILY [TC_BECOME]
defClone	Creates a new defect based on values contained in a specified defect.	defectNumber	TC_FAMILY [TC_BECOME]
defDrvr	Lists all defects for a specified driver.	driverName	TC_FAMILY [TC_BECOME]
defFfea	Creates a new defect based on values contained in a specified feature.	featureNumber	TC_FAMILY [TC_BECOME]
defNew	Displays the number of the most recent defect that was entered in the system.		TC_FAMILY [TC_BECOME]

Script name	Function	Inputs	Environment variables
defReopn	Reopens a previously canceled or returned defect.	defectNumber	TC_FAMILY [TC_BECOME]
defRept	Generates a global defect report showing work areas, test records, approval records, and fix records.		TC_FAMILY [TC_BECOME]
defState	Lists the defect number of all defects that are in a specified state.	stateName	TC_FAMILY [TC_BECOME]
defStats	Displays total active defect statistics on a defect owner area basis.	ownerArea	TC_FAMILY [TC_BECOME]
defWRef	Displays the full details of defects that contain the specified reference field value.	reference	TC_FAMILY [TC_BECOME]
dfDesc	Displays the full remarks that were entered when the specified defect or feature was created.	{defectNumber featureNumber}	TC_FAMILY [TC_BECOME]
feaClone	Creates a new feature based on values contained in a specified feature.	featureNumber	TC_FAMILY [TC_BECOME]
feaDvr	Lists all features contained in a specified driver.	driverName	TC_FAMILY [TC_BECOME]
feaFdef	Creates a new feature based on the values contained in the specified defect.	defectNumber	TC_FAMILY [TC_BECOME]
feaNew	Displays the number of the most recent feature that was entered in the system.		TC_FAMILY [TC_BECOME]
feaReopn	Reopens a previously canceled or returned feature.	featureNumber	TC_FAMILY [TC_BECOME]
feaRept	Generates a global feature report showing work areas, test records, size records, and fix records.		TC_FAMILY [TC_BECOME]
feaState	Lists the feature number of all features that are in a specified state.	stateName	TC_FAMILY [TC_BECOME]
feaStats	Displays total active feature statistics on a feature owner area basis.	ownerArea	TC_FAMILY [TC_BECOME]
drvByDF	Lists the name of the drivers that contain a specified defect or feature.	{defectNumber featureNumber}	TC_FAMILY [TC_BECOME]
drvMem	Lists the defect and feature members of a specified driver for a specified release.	driverName [releaseName]	TC_FAMILY [TC_RELEASE] [TC_BECOME]
mailTo	Sends a message to the addresses read through stdin.	messagefile subject	
ownerChg	Re-assigns all current work and objects owned by userLogin1 to userLogin2.	userLogin1 userLogin2	TC_FAMILY [TC_BECOME]
prtChckin	Checks parts into the TeamConnection family. When common parts are encountered, the script requests the releases for which the part should remain common.	partPathName [releaseName]	TC_FAMILY [TC_RELEASE] [TC_BECOME]
prtChgDf	Lists all the parts that were changed for a specified defect or feature.	{defectNumber featureNumber}	TC_FAMILY [TC_BECOME]
prtChgDr	Lists the parts that were changed for a specified driver.	driverName	TC_FAMILY [TC_BECOME]

Script name	Function	Inputs	Environment variables
prtComGt	Extracts all the parts associated with a specific component. The parts are placed in a directory that represents the release name to which the version of the part is associated. This directory is created relative to the relativePathName parameter.	componentName relativePathName [committed]	TC_FAMILY [TC_BECOME]
prtComp	Lists all parts related to a specified component.	componentName	TC_FAMILY [TC_BECOME]
prtHist	Lists all defect and feature numbers and abstracts that caused a change to a specified part in a specified release.	partName [releaseName]	TC_FAMILY [TC_RELEASE] [TC_BECOME]
prtInfo	Displays information for a specified part.	partName	TC_FAMILY [TC_RELEASE] [TC_BECOME]
prtLock	Lists all parts that a specified user has locked.	userLogin	TC_FAMILY [TC_BECOME]
prtLokBy	Lists who has a specified part checked out.	partName	TC_FAMILY TC_RELEASE [TC_BECOME]
PrtPath	Finds and lists all parts that match a partial part path name.	partPathName	TC_FAMILY [TC_BECOME]
prtRel	Lists all parts related to a specified release.	releaseName	TC_FAMILY [TC_BECOME]
prtWaGt	Extracts all the parts associated with a specific work area and places them in the path specified by the relativePathName parameter.	releaseName workareaName relativePathName	TC_FAMILY [TC_BECOME]
rByArea	Generates a manager's report based on the specified areas or departments of interest.	areaName ...	TC_FAMILY [TC_BECOME]
relOwner	Displays a list of addresses of all release owners in a specified TeamConnection family.	familyName	[TC_BECOME]
showConf	Lists the valid values pertaining to a specified configurable type.	configType	TC_FAMILY [TC_BECOME]
userAuth	Lists the users who have the authority to give other users access to a specified component.	componentName	TC_FAMILY [TC_BECOME]
userInfo	Finds user information based on part of the user's name. A fuzzy search is performed.	userName	TC_FAMILY [TC_BECOME]
usersAll	Lists the addresses of all users in a specified TeamConnection family.	familyName	[TC_BECOME]
usrAcc	Lists the explicit access of a specified user for the specified component and its descendant components.	userLogin componentName	TC_FAMILY [TC_BECOME]
usrRept	Generates a user's report based on the specified user login.	userLogin	TC_FAMILY [TC_BECOME]
verByPrt	Lists the version numbers, release names, and path names for the specified part.	partName releaseName	TC_FAMILY [TC_BECOME]
waComit	Lists the work areas that are in the commit state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]
waFix	Lists all the work areas that are in the fix state for a given release.	releaseName	TC_FAMILY [TC_BECOME]

Script name	Function	Inputs	Environment variables
waInLvl	Lists the work areas that are in the integrate state and are associated with at least one development driver for the specified release.	releaseName	TC_FAMILY [TC_BECOME]
waInt	Lists the work areas that are in the integrate state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]
waPrdLv	Lists the work areas that are included in a production driver and are in the integrate state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]
waStat	Generates a work area activity statistics report on a user area basis.	userArea	TC_FAMILY [TC_BECOME]
waTest	Lists the work areas that are in the test state for a specified release.	releaseName	TC_FAMILY [TC_BECOME]

Sample build scripts

fhbcob2.cmd

Calls the COBOL Visual Set for OS/2 compiler.

fhbcob2l.cmd

Calls the COBOL Visual Set for OS/2 compiler and link editor.

fhbocomp.cmd

Calls the VisualAge for C++ icc compiler.

fhbolib.cmd

Calls the OS/2 implib utility.

fhbolin2.cmd

Calls the VisualAge for C++ icc link editor.

fhbolink.cmd

Calls the link386 link editor.

fhborc.cmd

Calls the OS/2 resource compiler.

fhbplbld.cmd

Calls the OS/2 PL/1 compiler.

fhbpllnk.cmd

Calls the OS/2 PL/1 link editor.

edcc.jcl

Calls the C/370 JCL procedure.

fhbcobm.jcl

Calls the COBOL for MVS compiler.

fhbm370.jcl

Calls the C/370 compiler.

fhbmasm.jcl

Calls the MVS assembler.

fhbmc.jcl

Calls the C/370 compiler.

fhbmlink

Calls the MVS linkage editor.

fhbmpli.jcl
Calls the PL/1 MVS compiler.

fhbplked.jcl
Calls the C370 prelinker.

fhbtclnk
Calls the TeamConnection pseudo linker.

fhbwcomp.c
Calls the Microsoft Visual C++ compiler

fhbwlink.c
Calls the Microsoft linker

gather.cmd
Calls the Gather tool.

nvbridge.cmd
Calls the NetView bridge tool (NVBridge).

Sample parsers

fhbcbprs.cmd
A parser for COBOL applications.

fhbopars.cmd
A parser for C applications.

fhbplprs.cmd
A parser for PL/1 applications.

Sample package files

gather.pkf
A package file for the Gather tool.

nvbridge.pkf
A package file for the NetView bridge tool (NVBridge).

Appendix J. Program specifications for TeamConnection version 2.0

TeamConnection integrates software configuration management (SCM) function and object-based repository services to support application development in OS/2, Windows (3.1, 95, and NT), AIX, and HP-UX client/server team programming environments. TeamConnection supports the development of OS/2, Windows, AIX, HP-UX, and MVS client/server and distributed applications.

TeamConnection provides the following SCM services:

Version control

Version control provides the ability to store versions of an entire application so that the state of the application at a particular point in time can be re-created. Version control supports both serial and parallel development and includes a merge tool for reconciling source code changes.

Configuration management

Configuration management provides the ability to identify, organize, manage, and control access to development data. It also provides a notification mechanism to facilitate team communications.

Integrated build

Integrated build automates and optimizes the process of building an application or part of an application. The build process can segment the build activity so that multiple build processors can be used in parallel to complete the build faster.

Packaging and distribution support

Packaging and distribution support extends the conventional compile and link steps of the build function to perform the transformation steps that are required to prepare the application for use. The packaging and distribution support enables applications to be distributed electronically using LAN-based file server technology such as IBM NetView Distribution Manager/2.

Problem tracking and change control

TeamConnection provides a very strong development model that controls and tracks changes to development data. It associates identified application defects and features with the changes. TeamConnection manages the process of integrating related changes through the ability to build the application based on defects and features.

Customer support

Your options for IBM VisualAge TeamConnection support, as described in your License Information and Licensed Program Specifications, include electronic forums. You can use the electronic forums to access IBM VisualAge TeamConnection technical information, exchange messages with other TeamConnection users, and receive information regarding the availability of fixes. The following forums are available.

- **IBM Talklink**

Use the TEAMC CFORUM. For additional information about TalkLink, call

- United States 1-800-547-1283
- Canada 1-800-465-7999 ext. 228

- **CompuServe**

From any ! prompt, type GO SOFSOL, then select TeamConnection. For additional information, call 1-800-848-8199 and ask for representative 239.

- **Internet**

Go to the IBM homepage, <http://www.ibm.com>. Use the search function with keyword TeamConnection to go to the TeamConnection area.

If you cannot access these forums, contact your IBM representative.

There are several other support offerings available after purchasing IBM VisualAge TeamConnection. For a list of these offerings, please contact your IBM representative.

Bibliography

IBM VisualAge TeamConnection library

The following is a list of the TeamConnection publications.

- **License Information (GC34-4497):**
Contains license, service, and warranty information.
- **Administrator's Guide (GC34-4551):**
Lists the hardware and software that are required before you can install and use the IBM VisualAge TeamConnection product, provides detailed instructions for installing and configuring the TeamConnection family and build servers, and provides instructions for administering a TeamConnection family.
- **Getting Started with the TeamConnection Clients (SC34-4552):**
Tells first-time users how to install the TeamConnection clients on their workstations, and familiarizes them with the command line and graphical user interfaces.
- **User's Guide (SC34-4499):**
A comprehensive guide for TeamConnection administrators and client users that helps them install and use TeamConnection.
- **Commands Reference (SC34-4501):**
Describes the TeamConnection commands, their syntax, and the authority required to issue each command. This book also provides examples of how to use the various commands.
- **Quick Commands Reference (GC34-4500):**
Lists the TeamConnection commands along with their syntax.
- **Staying on Track with TeamConnection Processes (83H9677):**
Poster showing how objects flow through the states defined for each TeamConnection process.
- The following publications can be ordered as a set (SBOF-8560):
 - Administrator's Guide**
 - Getting Started with the TeamConnection Clients**
 - User's Guide**
 - Commands Reference**
 - Quick Commands Reference**
 - Staying on Track with TeamConnection Processes**

Tool Builder's Development Kit

The following publications are part of the Tool Builder's Development Kit feature:

- **Tool Builder's Development Guide (SC34-4553):**
Explains how to create and extend tools for accessing objects in the TeamConnection database. It contains guidance and reference information.
- **Information Model Reference (SC34-4554):**
Details the TeamConnection information model. This publication is available in softcopy only.

TeamConnection Technical reports

- 29.2147**
SCLM Guide to TeamConnection Terminology
- 29.2196**
Using REXX command files with TeamConnection MVS Build Scripts
- 29.2231**
TeamConnection Interoperability with MVS and SCLM
- 29.2235**
Using REXX command files with TeamConnection MVS Build Scripts for PL/I programs
- 29.2253**
Comparison between CMVC 2.3 and TeamConnection 2
- 29.2254**
Migrating from CMVC 2.3 to TeamConnection 2
- 29.2267**
TeamConnection frequently asked questions: how to do routine operating system tasks

ObjectStore

The following publications are part of the ObjectStore library of documents and are available for order from Object Design, Inc. To order these documents call (617) 674-5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time.

- **ObjectStore C++ Installation:**
Contains step-by-step procedures for installing the latest release of ObjectStore on a specific platform:
 - 310-100-40 I**
UNIX
 - 310-310-40 I**
Windows
 - 310-320-40 I**
OS/2
- **ObjectStore C++ API User Guide (310-000-40 U):**
Provides information about the application programming interface for application programmers.
- **ObjectStore C++ API Reference (310-000-40 R):**
Describes the API to the features provided by ObjectStore for application programmers.
- **ObjectStore C++ Building Applications (310-000-40 B):**
Provides information and instructions for compiling code, generating schemas, and linking files using all supported compilers; and provides instructions for developing ObjectStore client applications for use on multiple platforms.
- **ObjectStore Management (310-000-40 M):**
Provides information and instructions for performing management tasks on ObjectStore server and client systems. It includes server parameters, environment variables, and database utilities.
- **ObjectStore C++ Performance (310-000-40 P):**

Explains the fundamentals of designing and tuning ObjectStore applications for optimal performance.

IBM Exchange library

The publications listed below can be ordered as a set (SBOF-6098) or separately as indicated below. IBM Exchange will be available at a later date.

- *Licensed Programming Specification (GC34-4525):*
- *Installation Guide (SC34-4509):*
- *Bridge Builder's Guide (SC34-4508):*
- *User's Guide 1 (SC34-4506):*
- *User's Guide 2 (SC34-4507):*

Related publications

- Transmission Control Protocol/Internet Protocol (TCP/IP)
 - *TCP/IP 2.0 for OS/2: Installation and Administration (SC31-6075)*
 - *TCP/IP for MVS Planning and Customization (SC31-6085)*
- MVS
 - *MVS/XA JCL User's Guide (GC28-1351)*
 - *MVS/XA JCL Reference (GC28-1352)*
 - *MVS/ESA JCL User's Guide (GC28-1830)*
 - *MVS/ESA JCL Reference (GC28-1829)*
- NLS and DBCS
 - *AIX 4, General Programming Concepts: Writing and Debugging Programs (SC23-2533-02)*. See chapter 16 "National Language Support" for an updated contents of the AIX 3 material (see below).
 - *AIX 4, System Management Guide: Operating System and Devices (SC23-2525-03)*. See chapter 10, "National Language Support" for system tasks.
 - *AIX Version 3.2 for RISC System/6000, National Language Support (GG24-3850)*.
 - *Internationalization of AIX Software, A Programmer's Guide (SC23-2431)*.
 - *National Language Design Guide Volume 1 (SE09-8001-02)*. This manual contains very good information on how to enable an application for NLS.
 - *National Language Design Guide Volume 2 (SE09-8002-02)*. This manual provides information on the IBM language codes (consult the "Language codes" chapter).

Glossary

This glossary includes terms and definitions from the *IBM Dictionary of Computing*, 10th edition (New York: McGraw-Hill, 1993). If you do not find the term you are looking for, refer to this document's index or to the *IBM Dictionary of Computing*.

This glossary uses the following cross-references:

Compare to

Indicates a term or terms that have a similar but not identical meaning.

Contrast with

Indicates a term or terms that have an opposed or substantially different meaning.

See also

Refers to a term whose meaning bears a relationship to the current term.

A

absolute path name. A directory or a part expressed as a sequence of directories followed by a part name beginning from the root directory.

access list. A set of objects that controls access to data. Each object consists of a component, a user, and the authority that the user is granted or is restricted from in that component. See also *authority*, *granted authority*, and *restricted authority*.

action. A task performed by the TeamConnection server and requested by a TeamConnection client. A TeamConnection action is the same as issuing one TeamConnection command.

agent. See *build agent*.

alternate version ID. In collision records, the name of a version of a driver, release, or work area where the conflicting version of a part is visible.

approval record. A status record on which an approver must give an opinion of the proposed part changes required to resolve a defect or implement a feature in a release.

approver. A user who has the authority to mark an approval record with accept, reject, or abstain within a specific release.

approver list. A list of user IDs attached to a release, representing the users who must review part changes that are required to resolve a defect or implement a feature in that release.

attribute. Information contained in a field that is accessible to the user. TeamConnection enables family administrators to customize defect, feature, user, and part tables by adding new attributes.

authority. The right to access development objects and perform TeamConnection commands. See also *access list*, *base authority*, *explicit authority*, *granted authority*, *implicit authority*, *restricted authority*, and *superuser privilege*.

B

base authority. The set of actions granted to a user when a user ID is created within a TeamConnection family. See also *authority*. Contrast with *implicit authority* and *explicit authority*.

base name. The name assigned to the part outside of the TeamConnection server environment, excluding any directory names. See also *path name*.

base part tree. The base set of parts associated with a release, to which changes are applied over time. Each committed driver or work area for a release updates the base part tree for that release.

build. The process used to create applications within TeamConnection.

build agent. A program that handles access to persistent data on behalf of the build processor. Each build agent is connected to one and only one build processor, through a TCP/IP connection.

build associate. A TeamConnection part that is not an input to or an output from a build. An example of such a part is a read.me file.

build cache. A directory that the build processor uses to enhance performance.

build dependent. A TeamConnection part that is needed for the compile operation to complete, but it will not be passed directly to the compiler. An example of this is an include file. See also *dependencies*.

builder. An object that can transform one set of TeamConnection parts into another by invoking tools such as compilers and linkers.

build event. An individual step in the build of an application, such as the compiling of hello.c into hello.obj.

build input. A TeamConnection part that will be used as input to the object being built.

build output. A TeamConnection part that will be generated output from a build, such as an .obj or .exe file.

build pool. A group of build servers that resides in an environment. The environment in which several build servers operate. Typically, several servers are set up for each environment that the enterprise develops applications for.

build processor. A program that invokes the tools, such as compilers and linkers, that construct an application. Each build processor is connected to one and only one build agent, through a TCP/IP connection. See also *build agent* and *build cache*.

build scope. A collection of build events that implement a specific build request. See also *build event*.

build script. An executable or command file that specifies the steps that should occur during a build operation. This file can be a compiler, a linker, or the name of a .cmd file you have written.

build server. The combination of a build processor and a build agent. See also *build agent* and *build processor*.

build target. The name of the part at the top of the build tree which is the final output of a build. TeamConnection uses the build target to determine the scope of the build. See also *build tree*.

build tree. A graphical representation of the dependencies that the parts in an application have on one another. If you change the relationship of one part to another, the build tree changes accordingly.

C

change control process. The process of limiting and auditing changes to parts through the mechanism of checking parts in and out of a central, controlled, storage location. Change control for individual releases can be integrated with problem tracking by specifying a process for the release that includes the tracking subprocess.

check in. The return of a TeamConnection part to version control.

check out. The retrieval of a version of a part under TeamConnection control. In non-concurrent releases, the check out operation does not allow a second user to check out a part until the first user has checked it back in.

child component. Any component in a TeamConnection family, except the root component, that is created in reference to an existing component. The existing component is the parent component, and the new component is the child component. A parent

component can have more than one child component, and a child component can have more than one parent component. See also *component* and *parent component*.

child part. Any part in a build tree that has a parent defined. A child part can be input, output, or dependent. See also *part* and *parent part*.

client. A functional unit that receives shared services from a server. Contrast with *server*.

collision record. A status record associated with a work area or driver, a part, and one of the following:

- The work area or driver's release
- Another work area

TeamConnection generates a collision record when a user attempts to replace an older version of a part with a modified version, another user has already modified that part, and the first user's modification is not based on this latest version of the part.

command. A request to perform an operation or run a program from the command line interface. In TeamConnection, a command consists of the command name, one action flag, and zero or more attribute flags.

command line. (1) An area on the Tasks window or in the TeamConnection Commands window where a user can type TeamConnection commands. (2) An area on an operating system window where you can type TeamConnection commands.

committed version. The revision of a part that is visible from the release.

common part. A part that is shared by two or more releases, and the same version of the part is the current version for those releases.

comparison operator. An operator used in comparison expressions. Comparison operators used in TeamConnection are > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), and = (equal to).

component. A TeamConnection object that organizes project data into structured groups, and controls configuration management properties. Component owners can control access to data and notification of TeamConnection actions. Components exist in a parent-child hierarchy, with descendant components inheriting access and notification information from ancestor components. See also *access list* and *notification list*.

concurrent development. Several users can work on the same part at the same time. TeamConnection requires these users to reconcile their changes when they commit or integrate their work areas and drivers with the release. Contrast with *serial development*. See also *work area*.

configuration management. The process of identifying, managing, and controlling software modules as they change over time.

connecting parts. The process of linking parts so that they are included in a build.

context. The current work area or driver used for part operations.

corequisite work areas. Two or more work areas designated as corequisites by a user so that all work areas in the corequisite group must be included as members in the same driver, before that driver can be committed. If the driver process is not used in the release, then all corequisite work areas must be integrated by the same command. See also *prerequisite work areas*.

current version. The last visible modification of a part in a driver, release, or work area.

current working directory. (1) The directory that is the starting point for relative path names. (2) The directory in which you are working.

D

daemon. A program that runs unattended to perform a standard service. Some daemons are triggered automatically to perform their task; others operate periodically.

database. A collection of data that can be accessed and operated upon by a data processing system for a specific purpose.

default. A value that is used when an alternative is not specified by the user.

default query. A database search, defined for a specific TeamConnection window, that is issued each time that TeamConnection window is opened. See also *search*.

defect. A TeamConnection object used to formally report a problem. The user who opens a defect is the defect originator.

delete. If you delete a development object, such as a part or a user ID, any reference to that object is removed from TeamConnection. Certain objects can be deleted only if certain criteria are met. Most objects that are deleted can be re-created.

delta part tree. A directory structure representing only the parts that were changed in a specified place.

dependencies. In TeamConnection builds there are two types of dependencies:

- **automatic.** These are build dependencies that a parser identifies.

- **manual.** These are build dependencies that a user explicitly identifies in a build tree.

See also *build dependent*.

descendant. If you descendant a development object, such as, a part or a user ID, any reference to that object is removed from TeamConnection. Certain objects can be descendant only if certain criteria are met. Most objects that are descendants can be re-created.

disconnecting parts. The process of unlinking parts so that they are not included in a build.

driver. A collection of work areas that represent a set of changed parts within a release. Drivers are only associated with releases whose processes include the track and driver subprocesses.

driver member. A work area that is added to a driver.

E

end user. See *user*.

environment. (1) A user-defined testing domain for a particular release. (2) A defect field, in which case it is the environment where the problem occurred. (3) The string that matches a build agent with a build event.

environment list. A TeamConnection object used to specify environments in which a release should be tested. A list of environment-user ID pairs attached to a release, representing the user responsible for testing each environment. Only one tester can be identified for an environment.

explicit authority. The ability to perform an action against a TeamConnection object because you have been granted the authority to perform that action. Contrast with *base authority* and *implicit authority*.

extract. A TeamConnection action you can perform on a builder, part, driver or release builder. An extraction results in copying the specified builder, part, or parts contained in the driver or release to a client workstation.

F

family. A logical organization of related data. A single TeamConnection server can support multiple families. The data in one family cannot be accessed from another family.

family administrator. A user who is responsible for all nonsystem-related tasks for one or more TeamConnection families, such as planning, configuring, and maintaining the TeamConnection environment and managing user access to those families.

family server. A workstation running the TeamConnection server software.

FAT. See *file allocation table*.

feature. A TeamConnection object used to formally request and record information about a functional addition or enhancement. The user who opens a feature is the feature originator.

file. A collection of data that is stored by the TeamConnection server and retrieved by a path name. Any text or binary file used in a development project can be created as a TeamConnection file. Examples include source code, executable programs, documentation, and test cases.

file allocation table (FAT). The DOS- and OS/2-compatible file system that manages input, output, and storage of files on your system. File names can be up to 8 characters long, followed by a file extension that can be up to 3 characters.

fix record. A status record that is associated with a work area and that is used to monitor the phases of change within each component that is affected by a defect or feature for a specific release.

freeze. The freeze action saves changed parts to the work area. Thus, TeamConnection takes a snapshot of the work area, including all of the current versions of parts visible from that work area, and saves this image of the system. The user can always come back to this stage of development in the work area. Note, however, that a freeze action does not make the changes visible to the other people working in the release. Compare with *refresh*.

full part tree. A directory structure representing a complete set of active parts associated with the release.

G

Gather. A tool to organize files for distribution into a specified directory structure. This tool can be used as a prelude to further distribution, such as using CD-ROM or through electronic means like Netview DM/2. It can also be used by itself for distributing file copies to network-attached file systems.

GID. A number which uniquely identifies a file's group to an AIX system.

granted authority. If an authority is granted on an access list, then it applies for all objects managed by this component and any of its descendants for which the authority is not restricted. See also *access list*, *authority*, and *inheritance*. Contrast with *restricted authority*.

graphical user interface (GUI). A type of computer interface consisting of a visual metaphor of a real-world

scene, often as a desktop. Within that scene are icons, representing actual objects, that the user can access and manipulate with a pointing device.

GUI. Graphical user interface.

H

high-performance file system (HPFS). In the OS/2 operating system, an installable file system that uses high-speed buffer storage, known as a cache, to provide fast access to large disk volumes. The file system also supports the existence of multiple, active file systems on a single personal computer, with the capacity of multiple and different storage devices. File names used with HPFS can have as many as 254 characters.

host. A host node, host computer, or host system.

host list. A list associated with each TeamConnection user ID that indicates the client machine that can access the family server and act on behalf of the user. The family server uses the list to authenticate the identity of a client machine when the family server receives a command. Each entry consists of a login, a host name, and a TeamConnection user ID.

host name. The identifier associated with the host computer.

HPFS. See *high-performance file system*.

I

implicit authority. The ability to perform an action on a TeamConnection object without being granted explicit authority. This authority is automatically granted through inheritance or object ownership. Contrast with *base authority* and *explicit authority*.

import. To bring in data. In TeamConnection, to bring selected items into a field from a matching TeamConnection object window.

inheritance. The passing of configuration management properties from parent to child component. The configuration management properties that are inherited are access and notification. Inheritance within each TeamConnection family or component hierarchy is cumulative.

integrated problem tracking. The process of integrating problem tracking with change control to track all reported defects, all proposed features, and all subsequent changes to parts. See also *change control*.

interest group. The list of actions that trigger notification to the user IDs associated with those actions listed in the notification list.

J

job queue. A queue of build scopes. One job queue exists for each TeamConnection family.

L

lock. An action that prevents editing access to a part stored in the TeamConnection development environment so that only one user can change a part at a time.

login. The name that identifies a user on a multi-user system, such as AIX or HP-UX. In OS/2 and Windows, the login value is obtained from the TC_USER environment variable.

M

map. The process of reassigning the meaning of an object.

metadata. In databases, data that describe data objects.

N

name server. In TCP/IP, a server program that supplies name-to-address translation by mapping domain names to Internet addresses.

National Language Support (NLS). The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, MRI, or documentation of a product.

Network File System (NFS). The Network File System is a program that enables you to share files with other computers in networks over a variety of machine types and operating systems.

notification list. An object that enables component owners to configure notification. A list attached to a component that pairs a list of user IDs and a list of interest groups. It designates the users and the corresponding notification interest that they are being granted for all objects managed by this component or any of its descendants.

notification server. A server that sends notification messages to the client.

NTFS. NT file system.

NVBridge. A tool for automatic electronic distribution of TeamConnection software deliverables within a NetView DM/2 network.

O

operator. A symbol that represents an operation to be done. See also *comparison operators*.

originator. The user who opens a defect or feature and is responsible for verifying the outcome of the defect or feature on a verification record. This responsibility can be reassigned.

owner. The user who is responsible for a TeamConnection object within a TeamConnection family, either because the user created the object or was assigned ownership of the object.

P

parent component. All components in each TeamConnection family, except the root component, are created in reference to an existing component. The existing component is the parent component. See also *child component* and *component*.

parent part. Any part in a build tree that has a child defined. See also *part* and *child part*.

parser. A tool that can read a source file and report back a list of dependencies of that source file. It frees a developer from knowing the dependencies one part has on other parts to ensure a complete build is performed.

part. A collection of data that is stored by the family server and retrieved by a path name. They include text objects, binary objects, and modeled objects. These parts can be stored by the user or the tool, or they can be generated from other parts, such as when a linker generates an executable file.

path name. The name of the part under TeamConnection control. A path name can be a directory structure and a base name or just a base name. It must be unique within each release. See also *base name*.

pool. See *build pool*.

pop-up menu. A menu that, when requested, appears next to the object it is associated with.

prerequisite work areas. If a part is changed to resolve more than one defect or feature, the work area referenced by the first change is a prerequisite of the work area referenced by later changes. A work area is a prerequisite to another work area if:

- Part changes are checked in, but not committed, for the first work area.
- One or more of the same parts are checked out, changed, and checked in again for the second work area.

problem tracking. The process of tracking all reported defects through to resolution and all proposed features through to implementation.

process. A combination of TeamConnection subprocesses, configured by the family administrator, that controls the general movement of TeamConnection objects (defects, features, work areas, and drivers) from state to state within a component or release. See also *subprocess* and *state*.

Q

query. A request for information from a database, for example, a search for all defects that are in the open state. See also *default query* and *search*.

R

raw format. Information retrieved on the report command that has the vertical bar delimiter separating field information, and each line of output corresponds to one database record.

refresh. This TeamConnection action updates a work area with any changes from the release, and it also freezes the work area, if it is not already frozen.

relative path name. The name of a directory or a part expressed as a sequence of directories followed by a part name, beginning from the current directory.

release. A TeamConnection object defined by a user that contains all the parts that must be built, tested, and distributed as a single entity.

restricted authority. The limitation on a user's ability to perform certain actions at a specific component. Authority can be restricted by the superuser, the component owner, or a user with AccessRestrict authority. See also *authority*.

root component. The initial component that is created when a TeamConnection family is configured. All components in a TeamConnection family are descendants of the root component. Only the root component has no parent component. See also *component*, *child component*, and *parent component*.

S

search. To scan one or more data elements of a set in a database to find elements that have certain properties.

serial development. While a user has parts checked out from a work area, no one else on the team can check out the part. The user develops new material without interacting with other developers on the project. TeamConnection provides the opportunity to hold the part until the user is sure that it integrates with the rest

of the application. Thus, the lock is not released until the work area as a whole is committed. Contrast with *concurrent development*. See also *work area*.

server. A workstation that performs a service for another workstation.

shared part. A part that is contained in two or more releases.

shell script. A series of commands combined in a file that carry out a function when the file is run.

SID. The name of a version of a driver, release, or work area.

sizing record. A status record created for each component-release pair affected by a proposed defect or feature. The sizing record owner must indicate whether the defect or feature affects the specified component-release pair and the approximate amount of work needed to resolve the defect or implement the feature within the specified component-release pair.

stanza format. Data output generated by the Report command in which each database record is a stanza. Each stanza line consists of a field and its corresponding values.

state. Work areas, drivers, features, and defects move through various states during their life cycles. The state of an object determines the actions that can be performed on it. See also *process* and *subprocess*.

subprocess. TeamConnection subprocesses govern the state changes for TeamConnection objects. The design, size, review (DSR) and verify subprocesses are configured for component processes. The track, approve, fix, driver, and test subprocesses are configured for release processes. See also *process* and *state*.

superuser. This privilege lets a user perform any action available in the TeamConnection family.

system administrator. A user who is responsible for all system-related tasks involving the TeamConnection server, such as installing, maintaining, and backing up the TeamConnection server and the database it uses.

T

task list. The list of tasks displayed in the Tasks window. The user can customize this list to issue requests for information from the server. Tasks can be added, modified, or deleted from the lists.

TCP/IP. Transmission Control Protocol/Internet Protocol.

TeamConnection client. A workstation that connects to the TeamConnection server by a TCP/IP connection and that is running the TeamConnection client software.

TeamConnection part. A part that is stored by the TeamConnection server and retrieved by a path name, release, type, and work area. See also *part*, *common part*, and *type*.

TeamConnection superuser. See *superuser*.

tester. A user responsible for testing the resolution of a defect or the implementation of a feature for a specific driver of a release and recording the results on a test record.

test record. A status record used to record the outcome of an environment test performed for a resolved defect or an implemented feature in a specific driver of a release.

track subprocess. An attribute of a TeamConnection release process that specifies that the change control process for that release will be integrated with the problem tracking process.

Transmission Control Protocol/Internet Protocol (TCP/IP). A set of communications protocols that support peer-to-peer connectivity functions for both local and wide area networks.

type. All parts that are created through the TeamConnection GUI or on the command line will show up in reports with the type of TCPart as the part type. The TeamConnection GUI and command line can only check in, check out, and extract parts of the type TCPart.

Note: Parts created through an API can have other specified types. Refer to the *Commands Programming Reference* for more information.

U

user exit. A user exit allows TeamConnection to call a user-defined program during the processing of TeamConnection transactions. User exits provide a means by which users can specify additional actions that should be performed before completing or proceeding with a TeamConnection action.

user ID. The identifier assigned by the system administrator to each TeamConnection user.

V

verification record. A status record that the originator of a defect or a feature must mark before the defect or feature can move to the closed state. Originators use verification records to verify the resolution or implementation of the defect or feature they opened.

version. (1) A specific view of a driver, release, or work area. (2) A revision of a part.

version control. The storage of multiple versions of a single part along with information about each version.

view. An alternative and temporary representation of data from one or more tables.

W

work area. An object in TeamConnection that you create and associate with a release. When the work area is created, you see the most current view of the release and all the parts that it contains. You can check out the parts in the work area, make modifications, and check them back into the work area. You can also test the modifications without integrating them. Other users are not aware of the changes that you make in the work area until you integrate the work area to the release. While you work on files in a work area, you do not see subsequent part changes in the release until you integrate or refresh your work area.

working part. The checked-out version of a TeamConnection part.

Y

year 2000 ready. IBM VisualAge TeamConnection is Year 2000 ready. When used in accordance with its associated documentation, TeamConnection is capable of correctly processing, providing and/or receiving date data within and between the twentieth and twenty-first centuries, provided that all products (for example, hardware, software and firmware) used with the product properly exchange accurate date data with it.

Index

Special Characters

/Ft(dir) builder parameter 120

A

Accept Defects window 48
Accept Test Records 86
Activate Fix Records 80
Add Driver Members 78
approval command
 approving a fix 73
Approval Records window 73
approve state 40
authority
 basic 24
 build 103, 136
 for checking in parts 31
 for checking out parts 30
 for extracting parts 30

B

build action 6
build administrator 12
build agent
 accessing database remotely 102, 107
 assigning to build pools 106
 at work 151
 description of 94
 startup file, creating 107
 stopping 108
build directory 103
build environment 113
build event
 criteria used to determine success 114
 defining multiple outputs from one event 155
 definition of 95
 determining available agents 106
 timeout setting 114
 with VisualAge C ++ templates 120
build function
 authority 103, 136
 building a driver 82
 canceling a build 154
 collector part 156
 concepts of 93
 definition of 11
 diagram showing physical structure of 93
 features of 93
 monitoring build progress
 using Build Progress window 152
 using part -viewmsg command 152
 object model 97
 startup files, creating 107
 testing part updates 74
build mode 148
Build Parts window 54
build pool
 assigning agents to 106
 specifying when starting build 148
build processor
 at work 151
 build directory 103
 cache directory 103
 description of 94
 starting MVS 104
 startup file, creating 107
 stopping 108
build scope
 definition of 95
 determining 149
build scripts
 at work 151
 debug variable 117
 definition of 95
 for MVS
 compile example 129
 definition of 121
 file name conversions 125
 link example 132
 steps for working with 111, 121
 supported JCL syntax 127, 128
 writing 125
 for OS/2 115
 modifying contents of 118
 samples shipped 262
 testing 118
 timeout of 114
 writing 116
 writing an executable file for 117
build server
 definition of 94
 timeout setting 114
build target 148
build tree
 creating 143
 diagram showing build times 149
 display of 147
 example of 98
 multiple outputs from single event 155
 setting up for packaging
 for other distribution tools 163
 for the gather tool 160
 for the NVBridge tool 162
 setting up for packaging 160
 versions of 98
 working with 98
builder
 command
 connecting builder to its parts 119
 creating a builder 112, 118
 extracting a builder 119
 modifying builder contents 119
 connecting to parts 119, 137
 creating a null builder 113, 156

- builder (*continued*)
 - definition 95
 - for MVS
 - creating 121
 - environment supported 123
 - naming 122
 - passing parameters to a build script 126
 - versions of 122
 - for OS/2
 - creating 111
 - environment supported 113
 - naming 112
 - passing parameters to a build script 115
 - versions of 112
 - removing from a part 120
- building an application
 - a build agent at work 151
 - an example
 - adding job to job queue 151
 - build processors at work 151
 - build scripts at work 151
 - creating a build tree 143
 - creating builders and parsers 143
 - determining the build scope 149
 - extracting resulting executable 152
 - list of tasks 141
 - starting a build on the client 147
 - starting processors and agents 142
 - authority 103, 136
 - building all parts ignoring times 153
 - canceling a build 154
 - monitoring progress of build 152
 - preparing your parts 28, 99
 - report of which parts will be built 154
 - running in spite of errors 153
 - testing part updates 54
 - with VisualAge C ++ and templates 120
- buildView action 25

C

- cache data set
 - attributes of 104
 - deleting 105
 - description of 104
- cache directory
 - controlling size of 103
 - description of 103
- canceled state 39
- change control 3
- check-in action 6
- Check In Parts window 53
- check-out action 6
- checking in parts
 - authority needed to 31
 - example of 53
 - explanation of 31
- checking out parts
 - an example of 50
 - authority needed to 30
 - explanation of 30
- Checkout Parts window 50

- client
 - definition 5
 - starting 15
 - stopping 16
- closed state 39
- collector part
 - example of 156
 - using a null builder 113
- collision command
 - reconciling differences 67
- collision record
 - example of 31
 - reconciling using command line 66
 - reconciling using GUI 66
 - when creating driver members 79
- command file
 - fhbopars.cmd sample shipped 139
 - specifying 136
 - writing 139
- command line interface
 - using 19
 - viewing syntax online 19
- commands
 - becoming familiar with 19
 - fhomigmk 209
 - for client
 - teamc.log file 19
 - viewing syntax online 19
 - sendmail 101
 - tcmerge 67
 - teamagnt 106
 - teamcgui 15
- Commit Drivers 84
- commit state
 - of drivers 43
 - of work areas 42
- committing
 - a driver 84
 - versus integrating 86
- common parts
 - between releases 29
 - between work areas 29
 - breaking link 29
 - definition of 29
 - locking 29
- Complete Drivers 85
- Complete Fix Records 77
- complete state
 - moving work area to 86
 - of drivers 43
 - of work areas 42
- components
 - definition of 7
 - displaying structure of 23
 - example of hierarchy 7
 - information stored about 7
- Components window 23
- concepts of
 - build function 93
 - TeamConnection 4

- concurrent development
 - definition of 47
 - example of 31, 63
 - how to work in 28
 - reconciling differences in no-track process 65
 - reconciling differences in tracking process 79
- configuration management 3
- configuring
 - processes 86
- connect function 28
- courier.cmd 195
- create action 6
- Create Builder 111
- Create Parser 135
- Create Parts window 143
- Create Work Areas window 49

D

- DEBUG 117, 118
- defect command
 - accepting 48
 - closing 63
 - modifying ownership 71
 - verifying 63
- defects
 - accepting 48, 71
 - approving the fix 72
 - closing 62
 - definition of 9
 - designing 70
 - reassigning ownership 70
 - reviewing 70
 - sizing 70
 - states of
 - canceled state 39
 - closed state 39
 - design state 39
 - open state 39
 - return state 39
 - review state 39
 - size state 39
 - verify state 39
 - working state 39
 - verifying 62
 - working with 34
- delta file tree 43
- dependencies on a build
 - defining through parsers 95
 - viewing through a build tree 98
- dependent part 28
- design, size, review process 70
- design changes 34
- design state 39
- development mode 29
- differences in parts
 - reconciling in no-track process 65
 - reconciling in tracking process 79
- driver command
 - committing a driver 84
 - completing a driver 85
 - refreshing driver 82

- driver command (*continued*)
 - restricting driver 84
- driver member 78
- driverMember command
 - adding driver members 79
- drivers
 - building 82
 - committing changes into release 84
 - completing 85
 - definition of 9
 - preparing for formal test 85
 - refreshing 81
 - states of
 - commit state 43
 - complete state 43
 - integrate state 43
 - restrict state 43, 83
 - working state 42
 - versioning 33

E

- edit action 6
- Edit Task List window 51
- editing parts 30
- environment variables 203
 - setting 207
 - setting before invoking MVS build script 126
 - setting before invoking OS/2 build script 115
 - setting before invoking Windows NT build script 115
 - setting for command line usage 19
 - setting for GUI usage 17
- examples of
 - build script for an MVS compile 129
 - build script for an MVS link 132
 - build tree 98
 - build tree showing build times 149
 - building an application 141
 - client/server network 4
 - component hierarchy 7
 - concurrent development 63
 - display of a build tree 147
 - executable file for a build script 117
 - following a no-track process 47
 - following a tracking process 69
 - listing work areas in a release 89
 - retrieving past part versions 87
 - rules file 212
 - serial development 47
 - showing part/release/component relationship 8
 - starting make import 210
 - teamcpak command for Gather 167
 - teamcpak command for NVBridge 176, 196
 - writing a build script 116
- expand keywords 17
- extract action 6
- Extract Parts window 56
- extracting parts
 - an example of 55
 - authority needed 30
 - previous versions of 89

extracting parts (*continued*)
 resulting build executable 152
 versus checking out 30

F

family 6
family administrator responsibilities 12
family server 101
features
 definition of 9
 states of
 canceled state 39
 closed state 39
 design state 39
 open state 39
 return state 39
 review state 39
 size state 39
 verify state 39
 working state 39
 working with 34
fhbuild, build directory 103
fhbcbatch, cache directory 103
fhbhag.\$\$\$, listing of files in cache directory 103
fhbopars.cmd, sample command file 139
fhpicat utility 189
fhpscatt utility 189
fhpmcat utility 191
fhpobdel utility 187
fhpobdif utility 188
fhpobmon utility 187
fhprqmon utility 192
fhprqpur utility 192
fhpstat utility 187
fhptrpur utility 194
fhptrver utility 193
fhpuccat utility 190
fhpverif utility 191
files
 fhbuild 103
 fhbcbatch 103
 fhbhag.\$\$\$ 103
 teamc.log 19
filter windows for parts 25
finding objects 25
fix command
 accepting fix records 78
 completing fix records 78
 reactivate 80
fix records
 accepting 77
 completing 77
 moving back to active state 80
 reactivating 80
 when created 72
fix state 40
Fix Work Areas 79
Freeze Work Areas window 59
freezing work areas
 examples of 58, 75

freezing work areas (*continued*)
 explanation of 26
full part tree 43

G

Gather tool
 explanation of 165
 packaging file
 example of syntax 168
 keywords for 169
 specifying 166, 195
 syntax rules for 168
 writing 168
 teamcpak command 166
GUI
 client
 accepting a defect 48
 accepting fix records 77
 accepting test records 85
 accessing online help 18
 adding driver member 78
 approving the fix 73
 building a driver 82
 building parts 54
 checking in part 53
 checking out a part 50
 closing a defect 62
 committing driver changes into release 84
 completing a driver 85
 connecting builder to its parts 119
 connecting parser to parts 137
 creating a work area 49
 creating parsers 135
 extracting parts 55
 fast path 17
 freezing a work area 58
 integrating a work area, concurrent development 64
 integrating a work area, serial development 61
 moving fix records back to active state 80
 reactivating fix records 80
 reassigning defect ownership 70
 reconciling collisions 66
 refreshing a driver 81
 refreshing a work area, concurrent development 64
 refreshing a work area, serial development 60
 removing builder connection from parts 120
 removing parser connection from parts 138
 restricting a driver 83
 returning work area to fix state 79
 searching for parts 51
 Settings notebook 17
 starting 15
 stopping 16
 Tasks window 16
 using 15
 verifying a defect 62
family administrator
 to start family server 101
 to start notification server 101

H

help

- diagram push button 18
- how do I 18
- how to access 18

hierarchy

- component example 7
- displaying component structure 23

how do I help 18

I

importing makefile information 209

information model 3

input part 28

integrate state

- of drivers 43
- of work areas 41

Integrate Work Areas window 61

integrating

- commit versus integrate 86

interfaces

- becoming familiar with 15
- description of 5

J

job queue

- adding a job 151
- definition of 95

K

keyword

- ATTEMPTS 175, 183
- CLIENTINTERVAL 175, 182
- CORPID 179
- DATA 169, 177, 197
- ENTRY 184
- EXITPOST 171
- EXITPRIOR 171
- EXITREPLACE 171
- for Gather utility 168, 169
- in text files 241
- INSTALLDIR 180
- INSTALLPGM 180
- INSTALLS 175, 176, 184, 196
- IPARMS 181
- MAIL 174, 178, 195
- NVGLOBALS 178, 198, 199, 200
- package file 168
- PACKAGEFORMAT 169, 178, 197
- RULE 169
- SENDINTERVAL 175, 183
- SENDS 175, 185, 196
- SOURCE 170
- TARGET 171
- TARGETROOT 169
- TEAMCSERV 179
- TEST 175, 176, 184
- UNINSTALLPGM 182

L

LANG 203

M

mail exit routines 101

mail facility 101

makefile

- creating rules file 210
- example of starting import 210
- importing information 209

merging differences 67

Modify Defect Owner window 71

Modify Part Properties 119

MVS

build cache data set 104

build script

- definition of 121
- file name conversions 125
- for a compile 129
- for a link 132

builder 121

modifying RUNPGM JCL 104

starting build processor 104

stopping build processor 109

supported JCL syntax 127, 128

syntax for builds 127

N

naming

builders 112

parts 27

work areas 26

network 4

NLSPATH 203

no-track process, scenario 47

notification

setting up mail facility 101

notification server 101

nvbridge.cmd 173

NVBridge tool

checking if object exists in catalog 189

checking install history for object 187

checking status of NetView DM/2 components 187

cross-referencing differences 188

explanation of 173

NetView DM/2 output files 173

packaging file

example of syntax 177

keywords for 177

specifying 174

syntax rules for 177

writing 176

problem determination 185

removing information about a cataloged object 187

sample build script 173

teamcpak command 174

used as a builder for packaging 173

utilities 186

O

- online help
 - diagram push button 18
 - how to access 18
- open state 39
- OS_AS_SIZE 203
- OS_CACHE_SIZE 203
- OS_NETWORK 203
- OS_ROOTDIR 203
- OS_TMPDIR 203
- output part 28

P

- package file
 - for Gather tool
 - keywords for 169
 - specifying 166, 195
 - syntax rules for 168
 - writing 168
 - for NVBridgetool
 - example of syntax 177
 - keywords for 177
 - specifying 174
 - syntax rules for 177
 - writing 176
 - for Tivoli Courier tool
 - example of syntax 197
 - keywords for 197
 - syntax rules for 197
 - writing 197
- packaging
 - definition of 11
 - explanation of Gather tool 165
 - NVBridge tool 173
 - sample files shipped 263
 - setting up build tree
 - for other distribution tools 163
 - for the gather tool 160
 - for the NVBridge tool 162
 - tasks involved in 159, 195
- parameters
 - passing to a build script 115
 - where specified
 - attributes of a builder 115
 - attributes of part in a build tree 115
 - parameters of part -build command 116
- parser command
 - connecting parser to parts 138
 - creating a parser 136
- parsers
 - command file
 - fhbopars.cmd, sample shipped 139
 - specifying 136
 - writing 139
 - creating 135
 - definition of 95
 - explanation of 135
 - removing from a part 138
 - samples shipped 263
- part command
 - building a driver 83

- part command (*continued*)
 - building parts 55
 - canceling a build 154
 - checking in parts 54
 - checking out parts 50, 52
 - extracting build executable 152
 - extracting parts 56
 - listing parts that will be built 154
 - monitoring progress of build 152
 - removing builder connection 120
 - removing parser connection 139
 - starting a build 148
 - touching a part 150
 - viewing all version parts in a work area 87
- partFull action 25
- parts
 - authority needed to check in 31
 - authority needed to check out and extract 30
 - checking in 31
 - checking in, an example 53
 - checking in versus integrating 31
 - checking out, an example 50
 - checking out versus extracting 30
 - common
 - between releases 29
 - between work areas 29
 - breaking link 29
 - definition of 29
 - connecting builder 119
 - connecting parser 137
 - creating 27
 - definition of 6
 - dependent 28
 - editing parts 30
 - empty 27
 - extracting 30
 - extracting, an example 55
 - finding
 - using BuildView action 25
 - using Filter window 25
 - using PartFull action 25
 - using Parts action 25
 - using report command 25
 - input 28
 - linking 29
 - linking between releases 29
 - linking between work areas 29
 - locked 28
 - making changes visible to a release 84
 - making changes visible without driver subprocess 87
 - naming 27
 - preparing for build 28
 - removing builder from 120
 - removing parser from 138
 - retrieving past versions 87, 89
 - searching for 25
 - searching for, an example 51
 - testing updates to 54, 74
 - versioning 34
 - viewing versions in a work area 87

- parts (*continued*)
 - where placed on workstation 30
- parts action 25
- Parts Filter window 51
- PATH 203
- placeholder parts 27
- problem information, reporting 34
- processes
 - definition of 9
 - example of using a configured process 86
 - relating to defects and features 34

R

- reassigning ownership of a defect 70
- Reconcile Collision Record window 66
- reconciling differences 65, 79
- Refresh Drivers 82
- Refresh Work Areas window 60
- refreshing
 - a driver 81
 - a work area 59
- relative flag 30
- release management 3
- releases
 - committing driver changes to 84
 - common parts 29
 - definition of 7
 - example of relationship with other objects 8
 - linking parts between releases 29
 - parts common to more than one 29
 - using a configured process 86
 - versioning 32
- report command
 - listing work areas in a release 89
 - searching for parts 52
 - to find parts 25
 - used to view differences 66
- Restrict Drivers 83
- restrict state
 - of drivers 43, 83
 - of work areas 42
- retrieving past versions of objects 31
- return codes 186
- return state 39
- review state 39
- REXX execs 259
- rules file
 - creating 210
 - example of 212
- RUNPGM JCL 104

S

- sample files shipped
 - build script for NVBridge tool 173, 195
 - build scripts 262
 - for package function 263
 - mail exit routines 101
 - parsers 263
 - REXX execs 259
- Save to Task List push button 51

- scenarios
 - concurrent development 63
 - no-track process
 - accepting a defect 48
 - checking in a part 53
 - checking out a part 50
 - closing a defect 62
 - creating a work area 49
 - extracting a part 55
 - freezing the work area 58, 59
 - integrating a work area 61
 - reconciling differences 65
 - searching for a part 51
 - testing part updates 54
 - verifying a defect 62
 - tracking process
 - accepting a defect 71
 - accepting fix records 77
 - accepting test records 85
 - adding driver member to a driver 78
 - approving the fix 72
 - building a driver 82
 - checking out a part 73
 - committing driver changes into release 84
 - completing a driver 85
 - completing fix records 77
 - designing a defect 70
 - freezing the work area 75
 - reactivating fix record 80
 - reassigning defect ownership 70
 - refreshing a driver 81, 83
 - returning work area to fix state 79
 - reviewing a defect 70
 - sizing a defect 70
 - testing part updates 74
- searching for objects 25
- sendmail command 101
- serial development
 - definition of 47
 - example of 47
 - how to work in 28
- servers
 - definition 5
 - family server
 - definition 5
 - starting 101
 - notification server 101
- Settings notebook
 - for client 17
 - list of variables 17
- size state 39
- sizing records 70
- socket port
 - format for specifying for build agent 106
- starting
 - a build on the client 147
 - family server 101
 - GUI client 15
 - MVS build processor 104
 - notification server 101
- startup files 107

- states of objects
 - defects 37
 - drivers 42
 - features 37
 - work areas 40
- stopping
 - build agent 108
 - MVS build processor 109
 - OS/2 build processor 108
- superuser 6
- syntax
 - for MVS builds 127
 - how to view online 19
 - of NVBridge utilities 186
 - supported JCL syntax for build 127, 128
- system administrator, responsibilities 12

T

- tasks
 - authority to perform 24
 - preparing to build an application 99
 - understanding the basics 23
 - when following a no-track process 47, 69
- Tasks window 16
- TC_BATCH_TXNS 203
- TC_BECOME 203, 259
- TC_BUILD_AGENT_NOISY 203
- TC_BUILD_AGENTS_FILE 107, 203
- TC_BUILD_NOKEY 203
- TC_BUILD_PROC_NOISY 203
- TC_BUILD_PROCESSORS_FILE 107, 203
- TC_BUILDPPOOL 203
- TC_BULKCHUNKSIZE 203
- TC_CACHEPRUNEMETHOD 103, 105, 203
- TC_CACHESIZELIMIT 103, 105, 203
- TC_CASESENSE 203
- TC_COMPONENT 19, 203, 211
- TC_DBPATH 102, 107, 203
- TC_DEADLOCK_DEBUG 203
- TC_DEADLOCKBEEP 203
- TC_DECACHE_COUNT 203
- TC_DECACHE_WINDOW 203
- TC_FAMILY 19, 47, 115, 142, 203, 207, 209, 210, 259
- TC_FAMILY_SERVER_NOISY 203
- TC_INPUT 115, 117, 118
- TC_INPUTTYPE 115
- TC_JOB_QUEUE_INTERVAL 203
- TC_LOCATION 115
- TC_MAKEIMPORTRULES 203, 209, 212
- TC_MAKEIMPORTTOP 203, 210
- TC_MAKEIMPORTVERBOSE 203, 210
- TC_MIGRATERULES 203
- TC_NOTIFY_DAEMON 203
- TC_OUTPUT 117, 118
- TC_OUTPUTTYPE 115
- TC_RECYCLE_DEADLOCK 203
- TC_RECYCLE_SERVICE_COUNT 203
- TC_RELATIVE 74
- TC_RELEASE 115, 203, 209, 210, 259
- TC_STACK_TRACE 203
- TC_SYSTEM_LOG 203

- TC_TMP 203
- TC_TOP 30, 203
- TC_TRACE 203
- TC_TRACEATTEMPTS 203
- TC_TRACEDELAY 203
- TC_TRACEFILE 203
- TC_TRACESIZE 203
- TC_USER 203
- TC_WORKAREA 115, 203, 209
- tclogin command 19
- tcmerge 67
- TCP/IP
 - sendmail command 101
 - specifying socket port for build agent 106
- teamagt command 106
- teamcgui command 15
- TeamConnection
 - concepts of 4
 - diagram showing physical structure of 93
 - introducing 3
 - the basics of using 23
- teamcpak command
 - for Gather tool
 - command line flags 166
 - examples of 167
 - starting 166
 - syntax of 166
 - for NVBridge tool
 - command line flags 175
 - examples of 176, 196
 - starting 174
 - syntax of 174
 - using NVBridge utilities instead 186
- templates 120
- test command
 - accepting test records 86
- test records
 - accepting 85
 - when created 85
- test state
 - moving work area to 85
 - of work areas 42
- Tivoli Courier tool
 - explanation of 195
 - packaging file
 - example of syntax 197
 - keywords for 197
 - syntax rules for 197
 - writing 197
 - problem determination 200
 - sample build script 195
 - teamcpak command 195
 - Tivoli Courier output files 195
 - used as a builder for packaging 195
- TMP 203
- touch action 150
- tracking
 - following a no-track process 47
 - following a tracking process 69

U

utilities

NVBridge

- displaying syntax of 186
- fhpicat 189
- fhpiscat 189
- fhpmcat 191
- fhpobdel 187
- fhpobdif 188
- fhpobmon 187
- fhprqmon 192
- fhprqpur 192
- fhpstat 187
- fhptrpur 194
- fhptrver 193
- fhpuccat 190
- fhpverif 191

V

- verification record 39
- Verify Defects window 62
- verify state 39
- version control 3, 31
- versions 31
 - builders 112
 - of build trees 98
 - of drivers 33
 - of parts 34
 - of releases 32
 - of work areas 32
 - retrieving past part versions 87
 - viewing all part versions in a work area 87
- VisualAge C ++ 120

W

window examples

- Accept Defects 48
- Accept Test Records 86
- Activate Fix Records 80
- Add Driver Members 78
- Approval Records 73
- Build Parts 54
- BuildView 25
- Check In Parts 53
- Check Out Parts 50
- Commit Drivers 84
- Complete Drivers 85
- Complete Fix Records 77
- Components window 23
- Create Builder 111
- Create Parser 135
- Create Parts 143
- Create Work Areas 49
- Edit Task List 51
- Extract Parts 56
- Fix Work Areas 79
- Freeze Work Areas 59
- Integrate Work Areas 61
- Modify Defect Owner 71

window examples *(continued)*

- Modify Part Properties 119
- PartFull 25
- Parts Filter 25, 51
- Reconcile Collision Record 66
- Refresh Drivers 82
- Refresh Work Areas 60
- Restrict Driver 83
- Tasks 16
- Verify Defects 62

work area

- automatic creation of 26
- canceling 26
- creating 49
- definition of 8
- freezing 26
- freezing, an example 58, 75
- integrating, concurrent development 64
- integrating, serial development 61
- linking parts between work areas 29
- moving to complete state 86
- moving to test state 85
- naming 26
- reactivating to fix state 80
- refreshing, concurrent development 63
- refreshing, serial development 59
- returning to fix state 79
- reusing 27
- states of
 - approve state 40
 - commit state 42
 - complete state 42
 - fix state 40
 - integrate state 41
 - restrict state 42
 - test state 42
- things you can do with 8
- using 26
- versioning 32
- viewing all version parts 87
- when parts become visible to others 26
- when to create one 26

workarea command

- creating 49
- freezing work areas 59
- integrating when driver subprocess is not enabled 87
- integrating work areas, concurrent development 65
- integrating work areas, serial development 62
- refreshing work areas, concurrent development 64
- refreshing work areas, serial development 60
- return work area to fix state 80

working state

- of defects and features 39
- of drivers 42



Part Number:



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-4499-02

