Rational Software Corporation®

RATIONAL® CLEARCASE® MAINFRAME CONNECTORS

User's Guide

VERSION: 2002.05.20

Rational®

the software development company

support@rational.com
http://www.rational.com

Technical Acknowledgments
This software and documentation is based in part on BSD Networking Software Release 2, licensed from the Regents of the University of California. We acknowledge the role of the Computer Systems Research Group and the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley and the Other Contributors in its development.
This product includes software developed by Greg Stein <gstein@lyra.org> for use in the mod_dav module for Apache (http://www.webdav.org/mod_dav/).

Preface

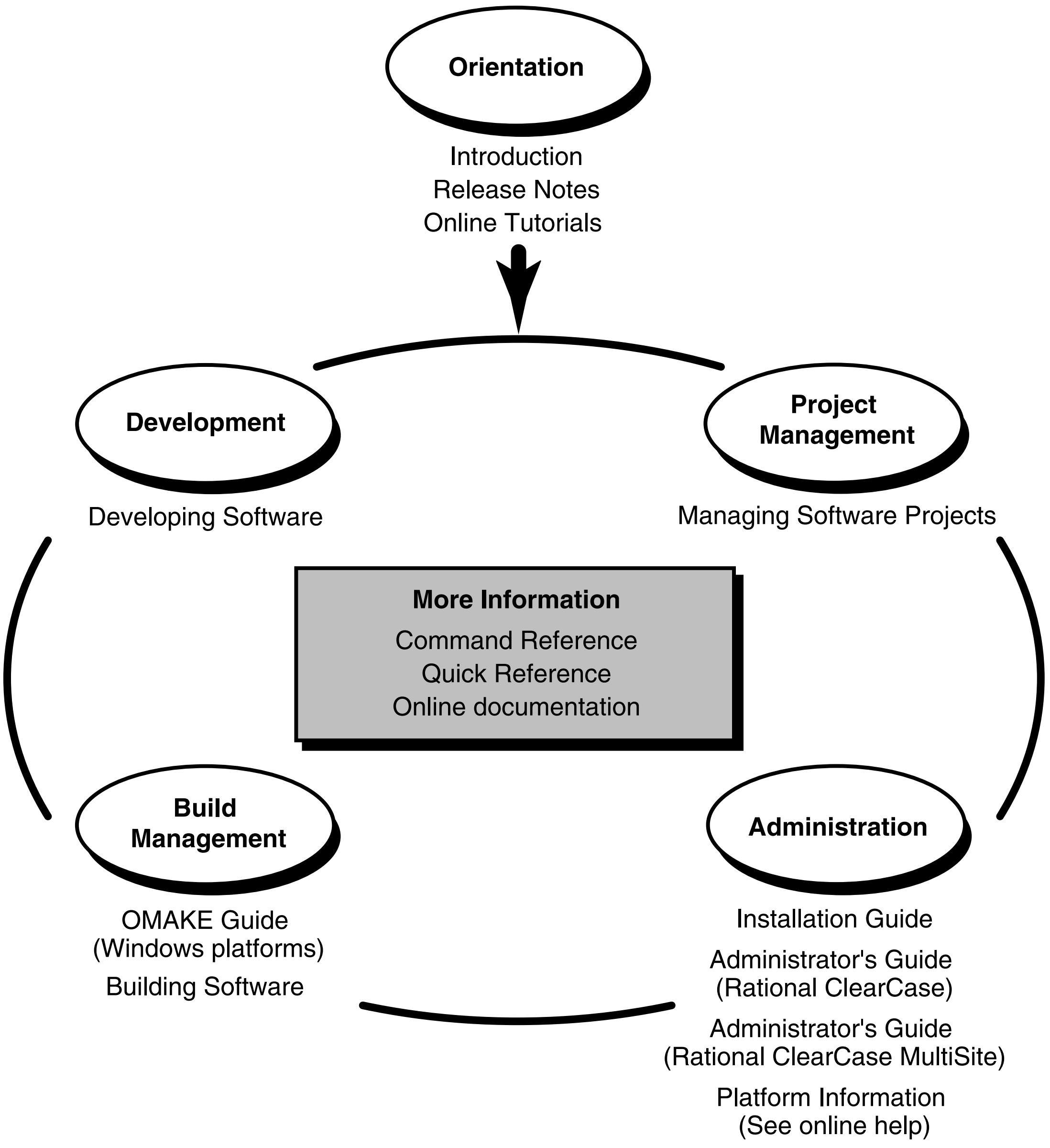
Rational ClearCase is a comprehensive configuration management (CM) system that manages multiple variants of evolving software systems and tracks changes. ClearCase maintains a complete version history of all software development artifacts, including code, requirements, models, scripts, test assets, and directory structures.

About This Manual

This manual describes how to install and configure Remote Build components, create JCL build scripts, and submit build requests.

Using Remote Build, programmers who write COBOL and other mainframe applications on client workstations can submit remote build requests to the mainframe.

ClearCase Documentation Roadmap



Typographical Conventions

This manual uses the following typographical conventions:

- *ccase-home-dir* represents the directory into which the ClearCase Product Family has been installed. By default, this directory is /opt/rational/clearcase on UNIX and C:\Program Files\Rational\ClearCase on Windows.
- *cquest-home-dir* represents the directory into which Rational ClearQuest has been installed. By default, this directory is /opt/rational/clearquest on UNIX and C:\Program Files\Rational\ClearQuest on Windows.
- **Bold** is used for names the user can enter; for example, command names and branch names.
- A sans-serif font is used for file names, directory names, and file extensions.
- **A sans-serif bold font** is used for GUI elements; for example, menu names and names of check boxes.
- *Italic* is used for variables, document titles, glossary terms, and emphasis.
- `A monospaced font` is used for examples. Where user input needs to be distinguished from program output, **bold** is used for user input.
- Nonprinting characters appear as follows: <EOF>, <NL>.
- Key names and key combinations are capitalized and appear as follows: SHIFT, CTRL+G.
- [ ]   Brackets enclose optional items in format and syntax descriptions.
- { }   Braces enclose a list from which you must choose an item in format and syntax descriptions.
- | A vertical bar separates items in a list of choices.
- ...   In a syntax description, an ellipsis indicates you can repeat the preceding item or line one or more times. Otherwise, it can indicate omitted information.

  Note: In certain contexts, you can use "**...**" within a pathname as a wildcard, similar to "*" or "?". For more information, see the **wildcards_ccase** reference page.
- If a command or option name has a short form, a "medial dot" (·) character indicates the shortest legal abbreviation. For example:

  lsc·heckout

Technical Support

If you have any problems with the software or documentation, please contact Rational Technical Support by telephone, fax, or electronic mail as described below. For information regarding support hours, languages spoken, or other support information, click the **Technical Support** link on the Rational Web site at **www.rational.com**.

Your Location	Telephone	Facsimile	Electronic Mail
North America	800-433-5444 toll free or 408-863-4000 Cupertino, CA	408-863-4194 Cupertino, CA 781-676-2460 Lexington, MA	**support@rational.com**
Europe, Middle East, and Africa	+31-(0)20-4546-200 Netherlands	+31-(0)20-4546-201 Netherlands	**support@europe.rational.com**
Asia Pacific	61-2-9419-0111 Australia	61-2-9419-0123 Australia	**support@apac.rational.com**

Figures

Figure 1 Processing an MVS Build Request 14
Figure 2 Processing a USS Build Request 15
Figure 3 Handling Multiple MVS Build Requests 16
Figure 4 Handling Multiple USS Build Requests 17
Figure 5 Build Files in ClearCase Explorer 56
Figure 6 Derived Objects in ClearCase Explorer 57

Tables

Table 1 Authentication Modes and Run-time Job Names 19
Table 2 USS Trace File Location 22
Table 3 Sample Files 59

Contents

Preface iii

About This Manual iii

ClearCase Documentation Roadmap iv

Typographical Conventions v

Technical Support vi

. . . . vi

Overview 1

Platforms and Components 1

About the Remote Build Server 1

Starting a Remote Build Request 2

Hardware and Software Requirements for Remote Build 2

Client Requirements 2

Connectivity 2

Supported Hardware and Operating Systems 2

Server Requirements 3

Connectivity 3

Supported Hardware and Operating Systems 3

Installing Remote Build Client and Server Components 5

Installing the Client Component 5

Setting Up the Server Component 5

About Installing Remote Build Server 5

About Remote Build Server Files 6

MVS Deliverables 6

USS Deliverables 6

Installation Prerequisites 7

Uploading the file RCCOS390 7

Running the RCCOS390 EXEC 7

Setting Up the MVS Server 8

Setting Up Run-time Parameters 8

Setting up SMP/E Libraries 8

Running SMP/E RECEIVE-APPLY-ACCEPT 9

Setting Up the USS Server 9

Setting Up Run-time Parameters 9

Setting Up JCL DD Definitions for SMP/E 10
Running SMP/E RECEIVE-APPLY-ACCEPT 10

Configuring the Remote Build Server 13

About Processing Build Requests. 13
Running a Build Server in MVS 13
Running a Build Server in USS 13
Processing Multiple Requests 16
Queuing Requests 17
Setting Queue Size 18
Authenticating Users. 18
Understanding Server Authentication Modes. 18
Making MVS Users Owners of Their Remote Build Jobs 19
Overriding the Default Job Name 19
Returning MVS Output Files to the Client Machine. 20
Returning USS Output Files to the Client Machine 20
Logging Server Messages and Traces 20
Activating Server Tracing 21
Activating Build Request Tracing 21
MVS Builds. 21
USS Builds. 21
Determining the USS Trace File Location 22
Configuring the Server Under MVS. 22
Modifying JCL. 22
Editing the RCCRUNM Member 23
Editing the RCCMVS Member 24
Starting the Server 24
Stopping the Server 25
Configuring the Server Under USS. 25
Modifying JCL. 25
Editing the RCCRUNU Member. 25
Editing the RCCUSS Member 26
Starting and Stopping the Server 26
Verifying Client/Server Communication (MVS) 26
Verifying Client/Server Communication (USS) 27
Running the Sample Executable 28

Sending a Build Request 29

Using the Client Command (rccbuild) 29

Synopsis 29

DESCRIPTION 31

Repeating Command Options 31

EBCDIC Translation (MVS Only) 31

Sending User IDs and Passwords 31

OPTIONS AND ARGUMENTS 31

Obtaining the Remote Build Client Version 31

Specifying a Remote Build Server 32

Pinging a Remote Build Server 32

Specifying a Local Build Script 32

Specifying a Server-Side Build Script 33

Specifying a Server-Side Build Script in a Nondefault PDS 33

Specifying Client-Based Source Files 34

Returning Output Files to Client Machine 35

Keeping Output Files on the Server 35

Specifying the Directory for a USS Build 35

Specifying TSO Login Details 36

Specifying Codepages for ASCII to EBCDIC Conversion 37

Setting Message Verbosity 37

Setting the Condition for Valid Return Codes 37

Specifying a Time-out Factor 38

Setting Environment Variables 38

Passing Variables to Build Scripts 39

Specifying Prefix for Messages Returned to Client Log File 39

Using the –i, –o and –d Options with USS Builds 40

Specifying Input and Dependent Files 40

Input File Examples 40

Specifying Output Files 41

Output File Examples 41

Working with Build Scripts 43

Identifying Build Scripts at Run Time 43

Understanding JCL Build Scripts 43

Understanding Coding Requirements 43

Testing Scripts on the Mainframe 44

Identifying Build Files 44

Sample Scenarios 45

Input File on Client Machine 45

Dependent File on Client Machine 45
Output File, Link-Edit Step 45
Coding the EXEC Statement. 46
Coding the DD Statement 47
Identifying Files Using RCCEXT DD Parameters. 48
Identifying Input Files 48
Identifying Dependent Files 48
Sending Output Messages to a Client File. 49
Sending Output Messages to the Client's Screen and a File. 49
Using Variables 50
Using Predefined Variables 50
Using User-Defined Variables 52
Setting Defaults for User-Defined Variables 52
File Name Conversions for MVS 53
Conversion Examples 53

Using Remote Build with clearmake 55

Creating a makefile for a Remote Build 55
Running the makefile 56
Returning Derived Objects to the Client. 56
About the Sample Files. 59
Submitting the COBOL Build Request 59
Editing the Batch File 60
Understanding the User-Defined Variables in the Build Script. 60
Running the Batch File 60
Running the COBOL Load Module 61
Editing the Batch File 61
Running the Batch File 62

Sample rccbuild Commands. 63

Sample Commands. 63

Index 65

Overview 1

Using the Remote Build feature of Mainframe Connectors, you can submit build requests from Windows and UNIX client platforms for ClearCase to OS/390 MVS and USS. You can configure Remote Build to return the derived objects to the client platforms where you can version them in ClearCase. In addition, you can audit the builds using the **clearmake** facility.

Platforms and Components

Remote Build runs on the following platforms:

Client platforms	AIX, Linux/390, Windows NT, Windows 2000, Windows XP
Server platforms	OS/390 and UNIX System Services (USS)

Remote Build has the following major components:

Client executable	**rccbuild**
Control statements	Job Control Language statements (JCL)
Mainframe executables	Load modules, such as **RCCBLDW**
USS deliverables	Executables (.exe and .dll)
ClearCase	clearmake utility and client-based VOBs
Mainframe connectivity	TCP/IP

About the Remote Build Server

The Remote Build server is multithreaded and starts a new job for each request. Builds run concurrently and are limited only by system resources, such as MVS JES initiators, and by a server option.

Remote Build supports multiple server instances, which you set up through different OS/390 ports.

Starting a Remote Build Request

You can start a build request at the operating system prompt, or through a script or makefile. In addition, you can point to build scripts and input files on client or server machines.

Hardware and Software Requirements for Remote Build

This section describes software requirements for the client and server components.

Client Requirements

Connectivity

TCP/IP

Supported Hardware and Operating Systems

Hardware Platform	**Operating System**
IBM RISC System/6000	AIX 4.3.2, 4.3.3, 5.1 (32–bit support)
IBM PC-compatibles	Windows NT 4.0 SP5 and higher Windows 2000 SP1 and higher Windows XP Pro
IBM System/390	SuSE Linux Enterprise Server 7 (2.4.7 kernel)
IBM zSeries	SuSE Linux Enterprise Server 7 (2.4.7 kernel)

The zSeries platform refers to "Linux S/390" running on the IBM zSeries hardware in the 31-bit run-time environment. "Linux for zSeries" in a 64-bit run-time environment is not supported.

This release has not been tested on the IBM S/390 under VIF.

Server Requirements

Connectivity

TCP/IP

Supported Hardware and Operating Systems

Hardware Platform	**Operating System**
IBM System/390	OS/390 2.9 and higher, including UNIX System Services (USS)
IBM zSeries	OS/390 2.9 and higher, including USS

2 Installing Remote Build Client and Server Components

This chapter describes how to install the Remote Build client and server components.

Installing the Client Component

Install all versions of the Remote Build client as a patch to ClearCase. Individual patches are available through this password-protected Rational Software site:

http://www.rational.com/support/downloadcenter/patches/clearcase/index.jsp

Setting Up the Server Component

This section describes how to install the Remote Build server component on the mainframe. This component is included in the same ClearCase patch with the Remote Build client component

About Installing Remote Build Server

Using ISPF panels and the ISPF Editor, you set up both the MVS and USS Remote Build servers. From the main panel, you pick MVS or USS. You must install the MVS server before installing the USS server.

The installation process accomplishes the following tasks:

- Allocates PDSs, including:
  - JCL library, which contains JCL that starts the Remote Build server load modules.
  - Load library, which contains server load module.
  - Object library, which contains server object code.
  - Procedures library, which can contain production build scripts that you write.
  - Samples library, which can contain sample JCL build scripts.
- Creates the JCL required to generate the SMP/E control file (CSI).

- Customizes run-time JCL using the high-level qualifier that you specify.
- Prompts for the location of the Language Environment library (**SCEEKLED**).
- Prompts for the **VOLSER** for all Remote Build PDSs.
- Creates the JCL to receive, apply, and accept the SMP/E installation files.
- Link-edits the Remote Build object modules.
- Installs USS executables.
- Removes all work files when the installation completes successfully.
- Sets execute and read permissions on the USS executables and shell script.

About Remote Build Server Files

MVS Deliverables

These load modules are created during installation:

RCCBLDS	Main executable that accepts MVS and USS build requests.
RCCBLDW	Executable that processes MVS build requests.
RCCDLL	Dynamic link library for MVS.
RCCINIT	Wrapper executable that calls RACF and the **RCCBLDS** module.
RCCMSENU	English-language messages.
RCCMSG	Executable that formats messages.

These JCL members are used by the MVS server.

RCCMVS	Calls the RCCBLDW load module.
RCCRUNM	Calls the RCCINIT load module.

USS Deliverables

The USS deliverables include:

- **JCL**
  - **RCCRUNU –** JCL that calls the **RCCBLDS** load module.

    - **RCCUSS** – JCL that calls the USS shell script **rccbldw.sh**.
- **Executables**
    - **rccbldw** – Executable that processes USS build requests.
    - **rccbldw.sh** – Shell script that calls the **rccbldw** executable.
    - **rccdll** – Dynamic link library.

Installation Prerequisites

Before installing the servers, you need to take the following steps:

1. Upload the REXX exec **RCCOS390** in binary mode to a partitioned data set.
2. Using IBM's RACF, define **RATIONAL** as a valid high-level qualifier.

Uploading the file RCCOS390

To upload **RCCOS390** from a Remote Build client workstation:

1. Open an FTP connection to OS/390 MVS.

Warning: When using the FTP **lcd** command, enter a local directory path in double quotes. For example, **lcd "C:\Program Files\dir\subdir"**

2. Change to binary transfer mode: **binary**
3. Change the destination to the desired partitioned data set.

   cd '*pds***'**
4. Upload the file:

   put RCCOS390
5. Quit the FTP session.

Running the RCCOS390 EXEC

1. From the ISPF Command Shell panel, run the command

   ex '*pds***(RCCOS390)'**

   where *pds* is the destination PDS for the REXX exec.

   The SMP/E INSTALLATION menu opens.

Setting Up the MVS Server

Type **1**, for INSTALLING MVS SERVER, and press ENTER. The MVS REMOTE BUILD SERVER menu opens.

Warning: During the installation process, do not exit from the SMP/E panels until the install completes successfully.

Setting Up Run-time Parameters

To set up run-time parameters:

1 Type **1**, for SET UP INSTALLATION OPTIONS, and press ENTER.

  A panel for entering run-time parameters opens.

2 Add a **JOB** statement that includes this information:

  - Job Name that ends with a numeral, if the total length is less than eight characters.
  - Region size of 48 MB: **REGION=48M**

3 In the **SCEELKED LIBRARY** field, specify the location of the Language Environment library.

4 In the **DISK VOLSER** field, special a volume serial number.

5 In the **HIGH-LEVEL QUALIFIERS** field, specify one or more high-level qualifiers.

6 Press ENTER to save these values as defaults.

7 Press F3 to return to the MVS REMOTE BUILD SERVER menu.

Setting up SMP/E Libraries

To set up the SMP/E libraries:

1 From the MVS REMOTE BUILD SERVER menu, type **2**, for SET UP SMP/E LIBRARIES, and press ENTER.

  An ISPF Editor session opens, in Edit mode.

2 Edit the JCL as needed.

3 On the Command line, type **SUB** and press ENTER.

  On successful completion, a JES2 message like the following one appears:

```
userid ENDED AT N1  MAXCC=0 CN(INTERNAL)
```

4 In the ISPF editor, press F3 to return to the SMP/E INSTALLATION menu.

Running SMP/E RECEIVE-APPLY-ACCEPT

To set up the JCL for RECEIVE-APPLY-ACCEPT processing:

1 From the MVS REMOTE BUILD SERVER menu, type **3**, for RUN SMP/E RECEIVE, APPLY, AND ACCEPT, and press ENTER.

   An ISPF Editor session opens, in Edit mode.

2 Edit the JCL as needed.

3 On the Command line, type **SUB** and press ENTER.

   On successful completion, a JES2 message like the following one appears:

```
userid ENDED AT N1  MAXCC=4 CN(INTERNAL)
```

Setting Up the USS Server

To set up a USS server, you need to do the following:

1 Install the MVS server, whose discussion starts on page 8.

2 Run the USS SMP/E installation, as described *Setting Up Run-time Parameters* on page 9.

Setting Up Run-time Parameters

To set up run-time parameters:

1 From the SMP/E INSTALLATION menu, type **2**, for INSTALLING USS SERVER, and press ENTER.

   The USS REMOTE BUILD SERVER menu opens.

2 Type **1**, for SET UP INSTALLATION OPTIONS, and press ENTER.

   A panel for entering run-time parameters opens.

3 Add a **JOB** statement that includes this information:

   - Job name that ends with a numeral, if the total length is less than eight characters.

- Region size of 48 MB: **REGION=48M**

4 The DISK VOLSER default is the value from the corresponding MVS Server step. Change this value, as needed.

5 In the HFS TARGET DIRECTORY field, specify an existing USS directory. This is the destination for Remote Build server executables and a shell script.

6 Press ENTER to save these values as defaults.

The installation exec displays the following additional parameters that were used in the MVS server installation:

- TARGET CSI DATASET value
- HIGH LEVEL QUALIFIERS values

7 Press F3 to return to the USS REMOTE BUILD SERVER menu.

Setting Up JCL DD Definitions for SMP/E

To set up DD definitions for SMP/E:

1 From the USS REMOTE BUILD SERVER menu, type **2**, for SET UP SMP/E DDDEF, and press ENTER.

An ISPF Editor session opens, in Edit mode.

2 Edit the JCL as needed.

3 On the Command line, type **SUB** and press ENTER.

4 Press F3 to return to the USS REMOTE BUILD SERVER menu.

Running SMP/E RECEIVE-APPLY-ACCEPT

To set up the JCL for RECEIVE-APPLY-ACCEPT processing:

1 From the USS REMOTE BUILD SERVER menu, type **3**, for RUN SMP/E RECEIVE, APPLY, AND ACCEPT, and press ENTER.

An ISPF Editor session opens, in Edit mode.

2 Edit the JCL as needed.

3 On the Command line, type **SUB** and press ENTER.

4 Press F3 to return to the USS REMOTE BUILD SERVER menu.

On successful completion, a JES2 message like the following one appears:

```
userid ENDED AT N1  MAXCC=0 CN(INTERNAL)
```

Configuring the Remote Build Server 3

This chapter describes how to configure and run a Remote Build server. It also explains how to verify client/server communication.

About Processing Build Requests

The Remote Build server performs the following tasks:

- Receives build requests and files from the client.
- Performs character conversions (MVS only).
- Runs builds within its environment.
- Optionally collects and returns results to the client.

Running a Build Server in MVS

In MVS, the server load module **RCCBLDS** receives client build requests. **RCCBLDS** triggers the JCL member **RCCMVS**, which executes the **RCCBLDW** module. **RCCBLDW** processes your build scripts (Figure 1 on page 14).

Running a Build Server in USS

For USS operations, the server load module **RCCINIT** and **RCCBLDS** run in MVS. **RCCBLDS** triggers the JCL member **RCCUSS**, which starts the USS shell script **rccbldw.sh**. This script starts the executable **rccbldw**, which processes build requests (Figure 2 on page 15).

Figure 1 Processing an MVS Build Request

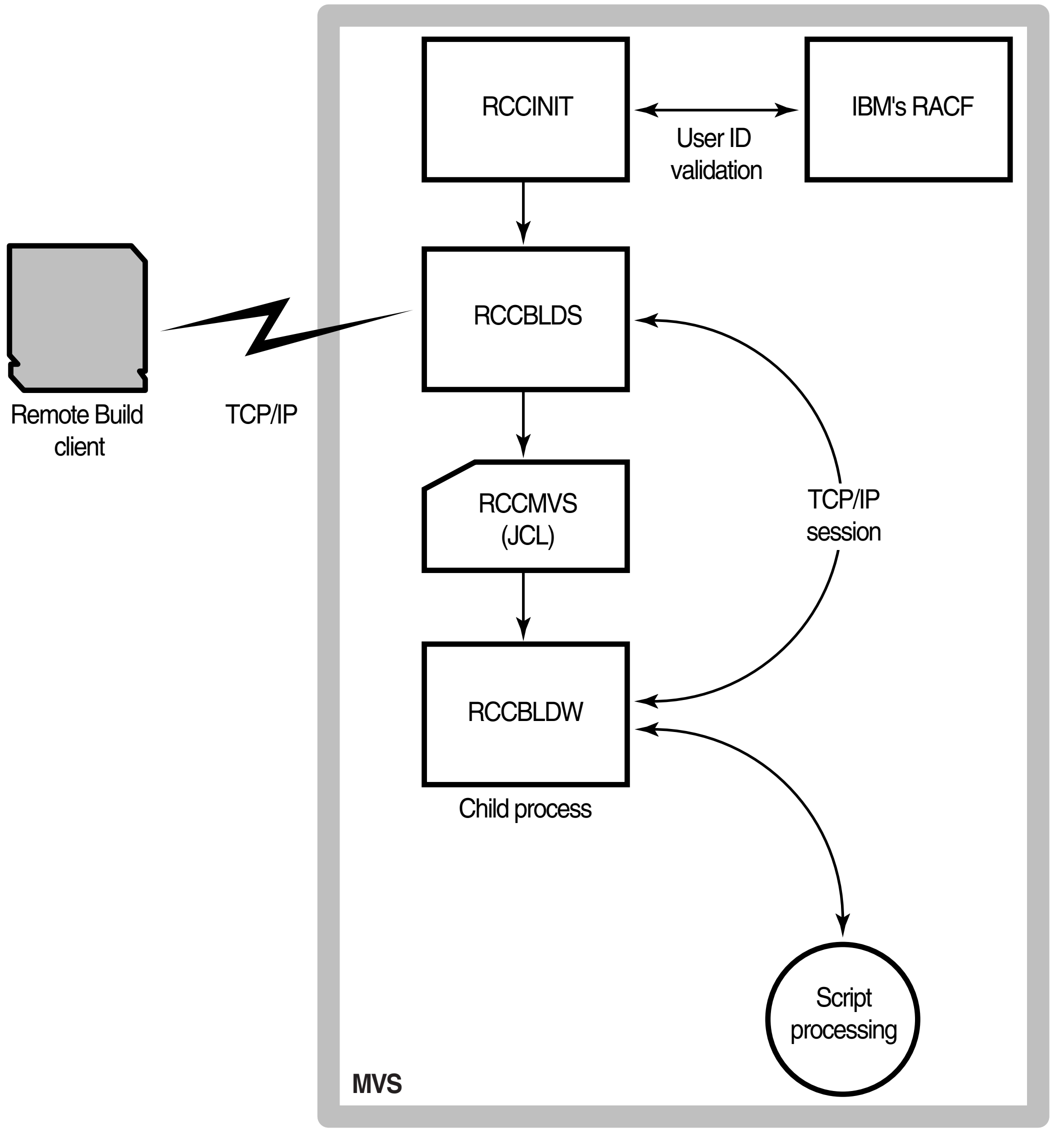
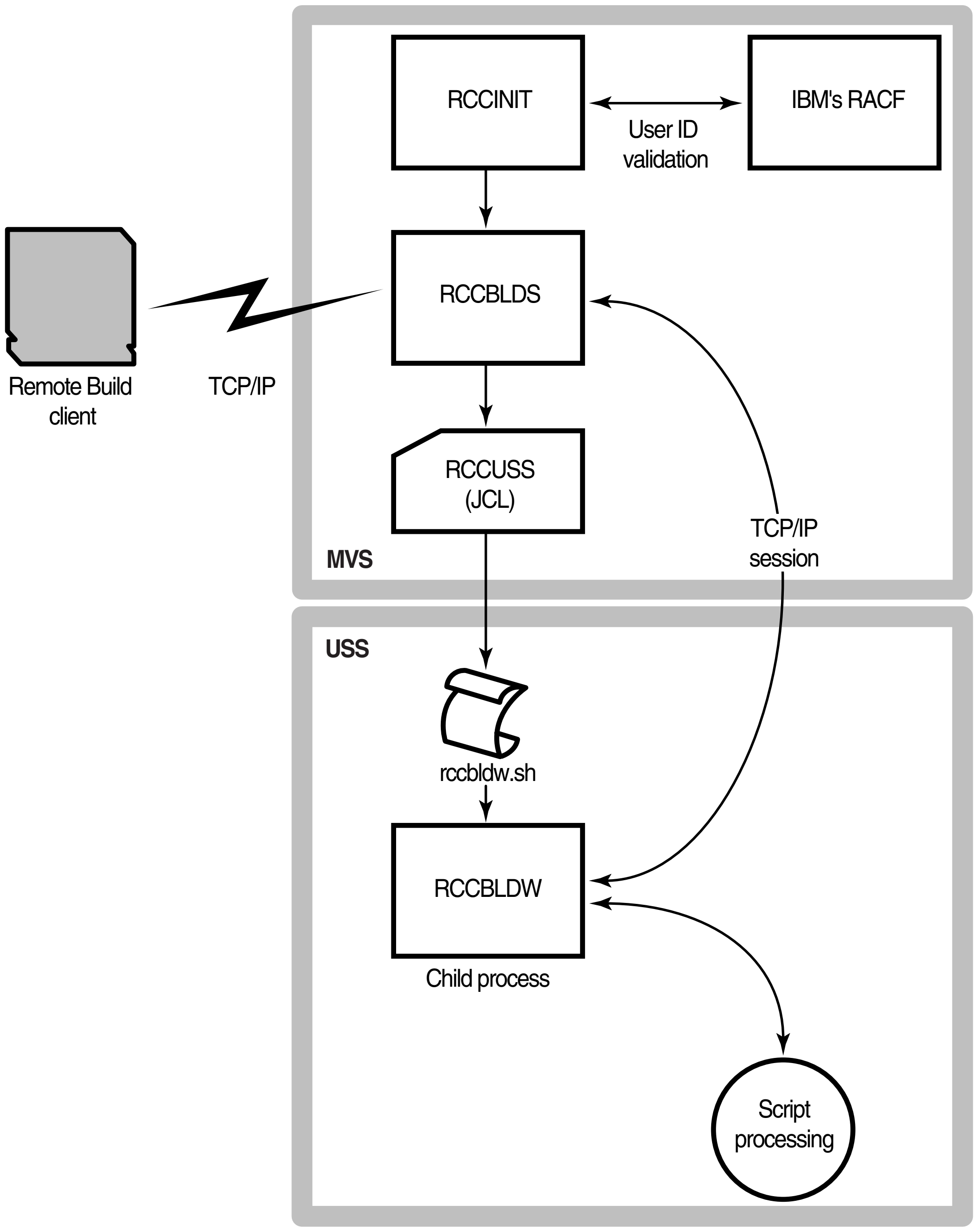


Figure 2 **Processing a USS Build Request**



Processing Multiple Requests

The server is multithreaded. Each build request starts a new process to handle the build transaction. You control the number of concurrent jobs using the **–n** server option. Concurrency is limited by system resources (such as JES initiators) and workload policies.

Figure 3 illustrates the spawning process for multiple MVS build requests.

Figure 3 Handling Multiple MVS Build Requests

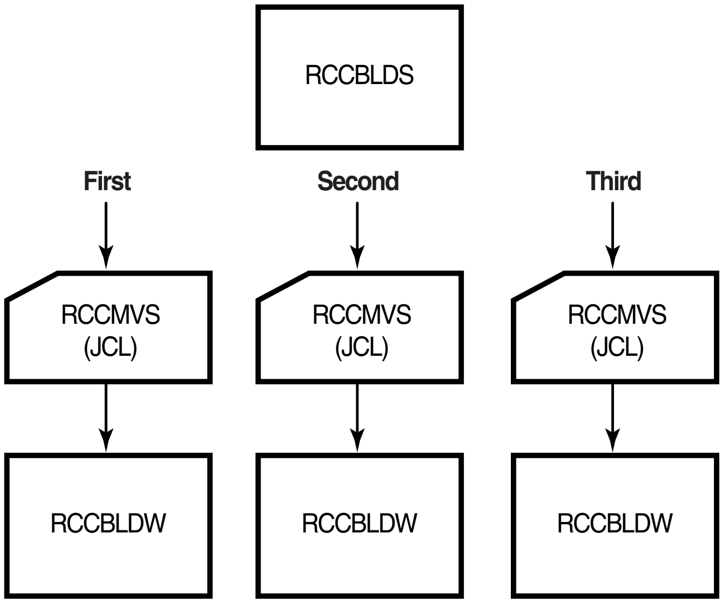
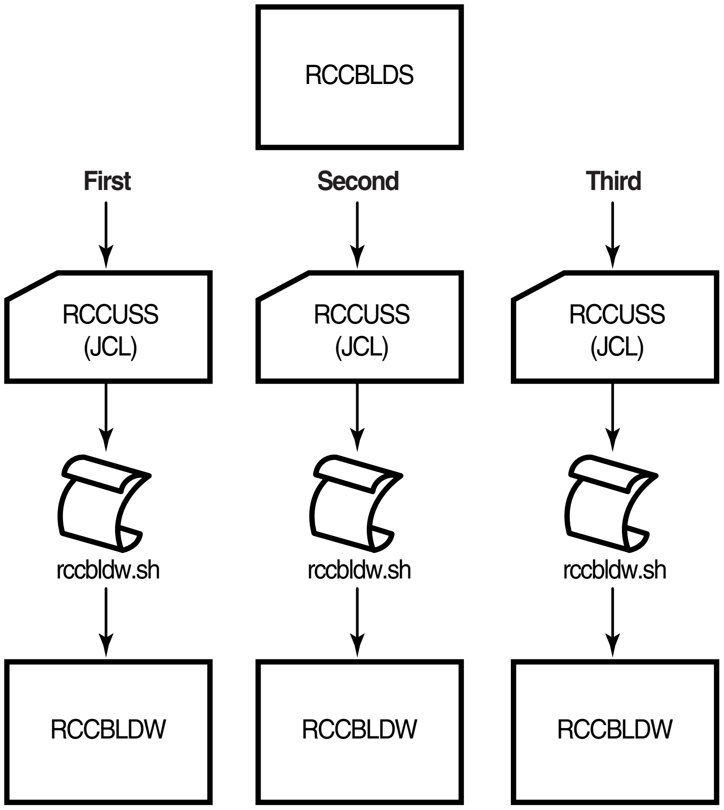


Figure 4 illustrates the spawning process for multiple USS build requests.

Figure 4 Handling Multiple USS Build Requests



Queuing Requests

When the concurrency limit is reached, the server queues any additional requests and submits them on a first-come-first-served basis. Each queued request uses a TCP/IP socket in a finite pool. The default queue size is 10. You control the queue size with the server option **–q**.

When the queue is full, the client waits 10 seconds and retries indefinitely. Retries are recorded in the client log file (**rccbuild.log**). The queue size must not exceed the pool size.

Setting Queue Size

We recommend that the sum of queue size and number of concurrent builds be less than the number of sockets that the server can keep active at a time:

```
queue_size + concurrent_builds < number_sockets
```

Authenticating Users

Remote Build server interfaces with IBM's RACF to perform the following tests:

- Validate TSO user IDs and passwords that are passed by the client command.
- Check user privileges for accessing MVS libraries and USS directories accessed during a build request.

A user ID that passes these tests becomes the owner of the remote build process.

To enable user authentication:

1. Start the Remote Build server with authentication mode 1 or 2.
2. Store the **RCCINIT** module in an APF-authorized library.

Understanding Server Authentication Modes

There are three authentication modes, as described below.

Mode	**Description**
0	No user authentication. The user ID that starts the Remote Build server becomes the owner for build processes requested by all users.
1	TSO user ID and password, passed by the client, are optional. If supplied, RACF validates them.
2	TSO user ID and password, passed by the client, are required. RACF validates them.

For information about setting the authentication mode in MVS, see *Editing the RCCRUNM Member* on page 23.

For information about setting the authentication mode in USS, see *Editing the RCCRUNU Member* on page 25.

Making MVS Users Owners of Their Remote Build Jobs

When you use a **JOB** statement in your **RCCMVS** JCL that specifies a hardcoded job name, Remote Build Server generates a job name as follows:

your_job_name + n

where *n* is a number from 0 through 9. For example, the first job that is named **ACPRUN** becomes **ACPRUN0**.

Overriding the Default Job Name

You can override the job name in **RCCMVS** with the TSO user ID of the build requestor.

To override the job name:

1 Substitute the job name value with the user-defined parameter **&USERID**. For example:

//&USERID JOB (ACCT#),'DEFINE TSO ID',CLASS=A

2 Specify a valid TSO user ID as the **rccbuild –au** parameter. For example:

rccbuild –h…–au RBUSER…

3 Start the Remote Build server using authentication mode **–a1** or **–a2**.

Table 1 describes the requirements for substituting a TSO user ID as a remote build job name.

Table 1 Authentication Modes and Run-time Job Names

Server authentication mode	**User ID supplied in rccbuild command**	**&USERID value in RCCMVS JCL Member**	**Run-time job name**
–a0	No	Replaced by RACF user ID that starts server	Server job ID plus suffix
–a0	Yes	Replaced by RACF user ID that starts server	Server job ID plus suffix
–a1	No	Replaced by RACF user ID that starts server	Server job ID plus suffix
–a1	Yes	Replaced by the **–au** supplied name	**–au** name plus suffix
–a2	Yes	Replaced by the **–au** supplied name	**–au** name plus suffix

Returning MVS Output Files to the Client Machine

To send output files from an MVS build to the client machine:

1 Identify the file extension of the output file in your JCL build script using a DD statement with the **RCCEXT** extension parameter.

For example:

```
//SYSOUT DD  RCCEXT=PRO,DISP=(NEW,DELETE),
//    UNIT=VIO,SPACE=(TRK,(10,10)),
//     DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
```

For more information about using the extension parameters to identify output files, see *Identifying Files Using RCCEXT DD Parameters* on page 48.

2 Specify the output file, using the **rccbuild –o** option.

For example:

rccbuild ... –o c:\builds\banner.pro

Returning USS Output Files to the Client Machine

To send output from a USS build to the client machine:

1 Specify a build directory, using the **rccbuild –l** or **–la** options. For example:

rccbuild ... –la /accounts/q3

2 Include an instruction in your build script or program that copies the output files to the current build directory. For example:

cp myoutput.exe .

3 Specify the output file on the command line using the **rccbuild –o** option. For example:

rccbuild ... –o c:\builds\myoutput.exe

Logging Server Messages and Traces

Remote Build server logs the trace for the server and for build runs in separate data sets.

- Server (**RCCINIT**) messages

  Server messages are captured in the **RCCBLOG** data set. Trace entries are captured in a data set called **SYS***n.* The default location for both data sets is the JES2 output queue.

- Build Run (**RCCBLDW**) messages

  Messages for a specific build run are recorded in a data set pointed to by the **RCCMVS** JCL.

Activating Server Tracing

To specify the **RCCBLOG** location, modify the **RCCBLOG** DD statements in the **RCCRUNM** and **RCCMVS** JCL.

To activate tracing, add the **–t** option to the **PARM EXEC** parameter in the **RCCRUNM** JCL.

Activating Build Request Tracing

Activating build tracing varies by build platform.

MVS Builds

To activate tracing for build requests, make the following changes to **RCCMVS** JCL:

1. Add this directive to the PARM clause within the **EXEC** statement that calls the **RCCBLDW** load module.

   PARM='ENVAR("_CEE_ENVFILE=DD:EDCENV")'

2. Add an **EDCENV** statement, in which you point to a sequential data set or a PDS member. For example:

   //EDCENV DD DSN=*sequential.dataset***,DISP=SHR**

3. In the sequential data set that the DSN parameter points to insert this line only:

   RCC_TRACE=*

USS Builds

To activate tracing, modify the shell script **rccbldw.sh**:

1. Add the following **EXPORT** commands:

EXPORT RCC_TRACE=*
EXPORT RCC_TRACEFILE=~*/filename*

where */filename* specifies the trace file.

Determining the USS Trace File Location

The location of the trace file depends on the following factors:

- Authentication mode of the Remote Build server.
- **rccbuild –au** value.
- Directory where **rccbldw** is running.

Table 2 describes the effect of these factors.

Table 2 USS Trace File Location

Server authentication mode	–au value	Directory running rccbldw	Trace file location
–a0	*any value*	**/rational/smith**	**/rational/smith**
–a1	**acp**	**/rational/smith**	**/rational/acp**
–a1	*no value*	**/rational/smith**	**/rational/smith**
–a2	**gls**	**/rational/journal**	**/rational/gls**

Configuring the Server Under MVS

This section describes how to customize the JCL that is used in running the server.

Modifying JCL

Customize the following JCL members:

- **RCCRUNM**, which executes the **RCCINIT** module.
- **RCCMVS**, which executes the **RCCBLDW** module.

Editing the RCCRUNM Member

1 Customize the **JOB** statement, as needed.

2 The **RSERVER** PROC contains default values for user-defined variables in the PARM EXEC parameter. Modify the **RSERVER** parameters as follows:

- **HLQ=’***hlq***’**: Replace *hlq* with the high-level qualifier of the library that contains the Remote Build load modules and **RCCMVS** JCL.
- **PORTNO=***portno*: Replace *portno* with the server listening port.
- **AUTH=***number*: Replace *number* with the server authentication mode. Valid values are 0, 1, 2. For more information about authentication modes, see *Understanding Server Authentication Modes* on page 18.
- **MAXBUILD=***number*: Replace *number* with the maximum number of concurrent builds. The default is 1. For more information about concurrency, see *Processing Multiple Requests* on page 16.

3 Modify the **STEPLIB** DD statement to point to the load library that contains the server modules.

4 Modify the **RCCWJCL** DD statement to point to the library that contains the RCCMVS JCL member.

5 Specify the following run-time parameters by adding options to the **PARM EXEC** parameter:

–t	Activates tracing. Trace entries are captured in the **RCCBLOG** data set.
–q *number*	Specifies the size of the queue for client requests. The default is 10.
–V	Optional. Specifies the verbosity level of server messaging (1, 2, or 3). The first instance sets the level at 1. Specify up to three instances. There is no default verbosity level. The following string sets the verbosity level at 2. **PARM='... –V –V'**

For example:

```
// PARM=('ENVAR("_CEE_ENVFILE=DD:EDCENV")/
//             -p &PORTNO -a &AUTH -n &MAXBUILD -q 5 -t -V -V')
```

Editing the RCCMVS Member

Following are required modifications for **RCCMVS**. Do not make any other modifications. The maximum number of JCL statements is 25.

1. Customize the **JOB** statement. To use the remote build requestor name (TSO ID) as the job name, insert **&USERID** in the job name field. For example:

   //&USERID JOB (ACCT#),'DEFINE TSO ID',CLASS=A

   For more information about using the requestor name, see *Overriding the Default Job Name* on page 19.

2. Edit the **STEPLIB** DD statement to point to the load library that contains the server modules. You can concatenate it with load libraries of other applications.

3. Modify the **RCCPROC** DD statement to point to the data set that contains your MVS build scripts.

   This modification is needed only if your build scripts reside on the MVS system. If your build script resides on the client, you must pass it to the server as part of the build transaction. For more information about build scripts, see Chapter 5, *Working with Build Scripts*.

4. If you want to activate tracing for build requests, modify the **EXEC** statement that calls the **RCCBLDW** load module. For more information, see *Activating Build Request Tracing* on page 21.

Starting the Server

You can start the server in two ways:

- As a started task
- As a batch job

To enable Remote Build to run as a started task:

1. Modify the **RCCRUNM** JCL, as needed.

   a. Delete the **JOB** statement.

   b. Delete all lines starting from the **PEND** statement.

2. Copy the modified **RCCRUNM** JCL to the library **SYS1.PROCLIB**.

To start the Remote Build server as a batch job, submit the **RCCRUNM** JCL.

Stopping the Server

To stop the Remote Build server, cancel the job that was used to start it.

Configuring the Server Under USS

This section describes how to customize the JCL that is used in running the server.

Modifying JCL

Customize the following JCL:

- **RCCRUNU**, which executes the **RCCINIT** module.
- **RCCUSS**, which calls the **BPXBATCH** utility to run the **rccbldw** shell script.

Editing the RCCRUNU Member

1 Customize the **JOB** statement.

2 Modify the **PARM** EXEC parameter:

// PARM='–p *portno* **–a** *number* **–n** *number* **–q** *number* **–t –V '**

where:

–p *portno*	Required. Specifies the server listening port.
–a *number*	Specifies the authentication mode of the server. The default mode is **2**. You can use authentication modes **1** and **2** only if the RCCINIT module is run from an APF-authorized library.
–n *number*	Specifies the number of concurrent builds. The default is **1**. When this limit is reached, the server queues any additional requests and submits them on a first come, first serve basis.
–q *number*	Specifies the size of the queue for client requests. The default is 10.
–t	Activates tracing. Trace entries are captured in the **RCCBLOG** data set.

–V	Specifies the verbosity level of the server (1, 2, or 3). The first instance sets the level at 1. Specify up to three instances. There is no default verbosity level. The following string sets the verbosity level at 2. **PARM='… –V –V'**

3 Modify the **RCCWJCL** DD statement to point to the library that contains the **RCCUSS** JCL member.

Editing the RCCUSS Member

1 Customize the **JOB** statement, as needed.

2 Modify the **PARM EXEC** to point to the shell script **rccbldw.sh**.

Starting and Stopping the Server

To start the Remote Build server, submit the **RCCRUNU** JCL.

To stop the server, determine the server process ID, and use the system **kill** command.

Verifying Client/Server Communication (MVS)

This section describes how to verify the connection between an MVS server and a client workstation by processing a sample text file.

1 On the client machine, run the following **rccbuild** command:

rccbuild –h *servermachine@portno* **–ft sample.jcl –b sample –it sample.inp –ot sample.out –k IBM-850 –r IBM-037**

where:

servermachine	Specifies the server machine.
portno	Specifies the listening port on the server machine. The port number must match the number in the **RCCRUNM** member. AIX and Linux/390 users, remove these codepage parameters: **–k IBM-850 –r IBM-037**

Sample JCL and an input file (**sample.inp**) are sent to the server. The input file is copied to the file **sample.out** and returned to the client machine.

Messages, like the following ones, appear on the client screen:

```
02/03/15 12:18:31 *** Success ***
02/03/15 12:18:31
RCCI-003
Program Name : 'IEBGENER'.
PARM         : ''.
RCCI-004
The MVS step 'TEST1' return code is '000000'.
02/03/15 12:18:31 Message files from build:
02/03/15 12:18:31    1:TEST1.SYSPRINT
02/03/15 12:18:32
*----------------------------------------------------------
```

2 In the directory that contains the **rccbuild** executable, browse the file **sample.out** for the following messages:

```
The Remote Build server and client components are communicating.
To see the server output messages, view the file RCCBLDC.LOG.
```

Verifying Client/Server Communication (USS)

This section describes how to verify the connection between an MVS server and a client workstation by compiling a C-language program and returning the output executable to the client machine.

1 On the client machine, run the following **rccbuild** command:

rccbuild –h *servermachine@portno* **–b cc –it rcopy.c –p --o rcopy.ob rcopy.c –o rcopy.ob –V**

where:

servermachine	Specifies the server machine.
portno	Specifies the listening port on the server machine. The port number must match the number in the **RCCRUNU** member. AIX and Linux/390 users, remove these codepage parameters: **–k IBM-850 –r IBM-037**

The C-language source file **rcopy.c** is sent to the server and compiled. The output file **rcopy.ob** is returned to the client machine.

Messages, like the following ones, appear on the client screen:

```
The build job has been queued by the server. Position is 1.

02/05/06 13:33:55
RCCI-014
Job  'BUILD000.' has been started by the server.
...
02/05/06 13:33:58 *** Success ***

02/05/06 13:33:58

Input Files:  rcopy.c

Output Files:  rcopy.ob
```

2 In the directory from which you ran **rccbuild**, browse for the file **rcopy.ob**.

Running the Sample Executable

The **rcopy** executable copies the list of files in a specified directory and their associated permissions to the client screen.

On the client machine, run the following **rccbuild** command:

rccbuild –h *servermachine@portno* **–b rcopy.ob –fb rcopy.ob –p** *path* **–V**

where:

servermachine	Specifies the server machine.
portno	Specifies the listening port on the server machine. The port number must match the number in the **RCCRUNU** member. AIX and Linux/390 users, remove these codepage parameters: **–k IBM-850 –r IBM-037**
path	Specifies an existing directory path on the server machine.

The server runs the **rcopy** executable and returns a list of files and associated permissions to the client machine.

Sending a Build Request

4

This chapter describes how to configure and send a build request.

The server creates a build job when you run the client program **rccbuild**. The client then waits for completion of the build while the server runs the build script. After running the build script, the server returns the results of the build to the client along with a return code of 0 (success) or 1 (failure).

If both the **rccbuild –o** option and the appropriate server options are used, build results are sent to the remote workstation's file system. These results include return codes, messages and any files that are returned to the client.

Using the Client Command (rccbuild)

This section describes **rccbuild** options and processing.

Synopsis

- Find out the version of the Remote Build client:

  rccbuild –version

- Specify a Remote Build Server:

  rccbuild –h *servermachine@portno…*

- Find out whether a Remote Build server is running on a specific port:

  rccbuild –h *servermachine@portno* **–testServer**

- Specify a build script that resides on the client machine:

  rccbuild –h *servermachine@portno* **–f [t|b]** *client_build_script* **–b** *copy_to_name…*

- Specify a build script that resides on the server:

  rccbuild –h *servermachine@portno* **–b** *server_build_script…*

- Specify a build script that resides on the server in a nondefault PDS:

rccbuild –h *servermachine@portno* **–b** *server_build_script*
–**proclib** *mvs_buildscript_library…*

- Specify client-based input and dependent files to the build process:

  rccbuild –h *servermachine@portno* **–i [t|b|n]** *input_file* **–d [t|b|n]** *dependent_file*

- Return output files, such as compiled objects, to the client machine.

  rccbuild –h *servermachine@portno* **–o [t|b]** *output_file…*

- Keep derived files on the server:

  rccbuild –h *servermachine@portno* **–on** *output_file…*

- Specify the directory for a USS build:

  rccbuild –h *servermachine@portno* **–l[a][c]** *build_directory…*

- Specify TSO login details:

  rccbuild –h *servermachine@portno…***–au** *userid* **–ap** *password*

- Specify codepages for ASCII to EBCDIC conversion (MVS server only):

  rccbuild –h *servermachine@portno…***–k** *client_codepage* **–r** *server_codepage*

- Set message verbosity level:

  rccbuild –h *servermachine@portno…***[–V|–V–V|–V–V–V]**

- Set the condition for valid return codes:

  rccbuild –h *servermachine@portno…***–c** *condition* **–n** *good_rc*

- Specify a time-out factor, in minutes:

  rccbuild –h *servermachine@portno…***–T** *timeout*

- Set environment variables:

  rccbuild –h *servermachine@portno…***–v** *var1=value var2=value2***…**

- Pass run-time variables to the build script:

  rccbuild –h *servermachine@portno…***–p** *build_parameters*

- Specify a prefix for server messages returned to the client log file **rccbuild.log**:

  rccbuild –h *servermachine@portno…***–P** *message_prefix*

DESCRIPTION

Use the **rccbuild** executable to submit a build request to an OS/390 server.

Repeating Command Options

You can repeat command options. The effect varies, as follows:

- For the following options, when there are conflicts in option values, the last value overrides other instances.
  - **–ap, –au, –b, –c, –f, –h, –k, –l, –n, –proclib, –P, –r, –T**
- Each instance of the following options supplements the current value:
  - **–db**, **–dn**, **–dt**, **–ib**, **–in**, **–it**, **–ob**, **–on**, **–ot**, **–p**, **–V**, **–v**

EBCDIC Translation (MVS Only)

During a client-to-server transfer, text files are converted to EBCDIC. When server files are transferred to the client, text files are converted to ASCII. Binary files are not converted in either direction.

The **rccbuild** processor cannot handle files that contain both text and binary data. If you have text files with imbedded binary data, transfer these files to the appropriate data sets before issuing the **rccbuild** command.

Sending User IDs and Passwords

Using the **–au** and **–ap** options, specify user IDs and passwords in uppercase. Lowercase and mixed-case names are not converted.

OPTIONS AND ARGUMENTS

You must specify the **–h** option with all **rccbuild** options except the **–version** option, which does not make a server request.

Obtaining the Remote Build Client Version

–version

Returns the following information about the executable. For example:

```
rccbuild Version:1.0.3.5
```

Specifying a Remote Build Server

–h *servermachine@portno*

Required except when **–version** is specified. Specifies the server name and the listening port. For example:

–h os390@2600

Supported server platforms: MVS, USS.

Pinging a Remote Build Server

–testServer

Returns the following information about the server: operating system, Remote Build server version, and authentication mode. The only other required option is **–h**.

Supported server platforms: MVS.

Sample output:

```
Operating System: OS/390 MVS
Version:              2002.05.20
Authentication Mode: 2
```

Specifying a Local Build Script

–f [**t** | **b**] *client_build_script* **–b** *copy_to_name*

where:

–f [**t** | **b**] *client_build_script*

Specifies a build script file that resides on the client machine, which is transferred to the server for processing. The **t** option (default) specifies that the build script file is a text file. Specify the **b** option if the file is binary. Note that the MVS server only accepts build files in text format.

–b *copy_to_name*

Specifies a copy-to name for the build script. Remote Build script copies the local script to the server under the copy-to name.

Supported server platforms: MVS, USS.

Examples:

This MVS example identifies a local JCL file on Windows, which is in text format:

–f D:\MYCOMP.JCL –b MYCOMP

Specifying a Server-Side Build Script

–b *server_build_script*

Without the **–f** option, the **–b** option specifies that the build script resides on the server.

In MVS, the server looks for the script in the partitioned data set that is pointed to by the **RCCPROC** DD statement in the **RCCMVS** JCL. The **RCCMVS** JCL is stored in the JCL installation library. To override this partitioned data set, use the **–proclib** option.

Supported server platforms: MVS, USS.

Example:

In the following example, the server looks for the script **MYSCRIPT** in the default PDS.

rccbuild…–b MYSCRIPT

Specifying a Server-Side Build Script in a Nondefault PDS

–proclib *mvs_buildscript_library*

Specifies an override to the default partitioned data set (PDS) that contains JCL build scripts. Use a fully qualified PDS name, and also specify the **–b** option. For information about the default PDS, see the **–b** option. The **–proclib** option is ignored when you use the **–f** option.

Supported server platforms: MVS.

Example:

rccbuild… –proclib REMOTE.BUILD.SCRIPTS

Specifying Client-Based Source Files

–i [t | b | n] *input_file…*

Specifies the names of one or more input files (separated by blanks) or a file that contains a space-delimited or comma-delimited list of files. Precede the name of a file that contains a file list with an at sign (@). For example: **@mylist.txt**

Supported server platforms: MVS, USS.

To indicate that the files are in text format, specify the **t** option. This is the default.

To indicate that the files are in binary format, specify the **b** option.

To indicate that the input files already exist on the server and are not transferred to the server, specify the **n** option. Use a DD statement in your JCL build script to indicate the location.

For more information about specifying files for USS builds, see *Using the –i, –o and –d Options with USS Builds* on page 40.

–d [t | b | n] *dependent_file…*

Specifies the names of one or more dependent files (separated by blanks) or a file that contains a list of files. Precede the name of a file that contains a file list with an ampersand (@). For example: **@mylist.txt**

Supported server platforms: MVS, USS.

To indicate that the files are in text format, specify the **t** option. This is the default.

To indicate that the files are in binary format, specify the **b** option.

To indicate that the input files already exist on the server and are not transferred to the server, specify the **n** option.

For more information about specifying files for USS builds, see *Using the –i, –o and –d Options with USS Builds* on page 40.

Returning Output Files to Client Machine

–o [t | b] *output_file…*

Specifies the names of one or more output files (separated by blanks).

Supported server platforms: MVS, USS.

To indicate that the files to be transferred to the client are in text format, specify the **t** option.

To indicate that the files are in binary format, specify the **b** option. This is the default.

For more information about specifying files for USS builds, see *Using the –i, –o and –d Options with USS Builds* on page 40.

Keeping Output Files on the Server

–on *output_file…*

Keeps a copy of the derived files on the server. To prevent transfer of the specified files to the client, specify the **n** option. After a successful build, the client creates the files specified after the **n** option as empty files.

The actual build output remains on the server, and an empty file is returned to the client. This provides a record on the client (with a time stamp) that the build was done. This file can be used to prevent unnecessary builds when used in conjunction with a make file.

Supported server platforms: MVS, USS.

Specifying the Directory for a USS Build

–l [a] [c] *build_location*

Specifies the path for the build location. To identify a relative path, omit the **a** option. To identify an absolute path, specify the **a** option and a fully qualified path. The server creates any directories that do not exist.

If you use the **–la** options, copying the output file to the current directory (.) is not needed.

Supported server platforms: USS.

To delete new directories when the build completes, specify the **c** option.

Examples:

The following example creates, if not present, the directory **Driver01** and compiles **hello.c** in that directory. Because the **c** option is not specified (**–l** instead of **–lc**), the directory **Driver01** is not deleted, and the object file **hello.o** is left in the directory.

rccbuild... –l Driver01 ... –b cc –p --c --o hello hello.c –i hello.c

The following example builds the **hello** object using **hello.o** (from the previous example). Because the **c** option is specified (**–lc** instead of **–l**), the directory **Driver01** is deleted after the build is complete.

rccbuild... –lc Driver01 ... –b cc –p --c --o hello hello.c –i hello.c

The following example builds the **hello** object using **hello.o** (from the previous example). Because the **a** option is specified (**–la** instead of **–l**) therefore, the server creates the directory **/Driver01** and makes the directory **/Driver01** the current directory for the build transaction.

Because the **c** option is not specified (**–la** instead of **–lac**), **Driver01** is not deleted, and the object file **hello.o** is left in the directory.

rccbuild... –la /Driver01 ... –b cc –p --hello hello.o –o hello

The following example builds the **hello** object using **hello.o** (from the previous example). Because the **c** option is specified (**–lac** instead of **–la**), the directory **Driver01** is deleted after the build is complete.

rccbuild... –lac /Driver01 ... –b cc –p --hello hello.o –o hello

Specifying TSO Login Details

–au *userid*

Specifies a TSO ID.

Supported server platforms: MVS and USS.

The server authentication mode determines whether a TSO ID is required. This is specified in the **RCCRUNM** JCL. For more information about authentication modes, see *Editing the RCCRUNM Member* on page 23.

–ap *password*

Specifies a TSO password.

Supported server platforms: MVS and USS.

The server authentication mode determines whether a TSO password is required. This is specified in the **RCCRUNM** JCL. For more information about authentication modes, see *Editing the RCCRUNM Member* on page 23.

Specifying Codepages for ASCII to EBCDIC Conversion

–k *client_codepage*

Specifies the codepage for the input, output, and build script files on the client. Codepage conversion occurs only on text files. The default codepage for the Windows NT client is IBM-850. The AIX and Linux/390 default is ISO-8859-1.

Supported server platforms: MVS, USS.

–r *server_codepage*

Specifies the codepage used on the server. The default codepage is IBM-1047.

Supported server platforms: MVS, USS.

Setting Message Verbosity

[–V | –V–V | –V–V–V]

Specifies the verbosity level of the server (1, 2, or 3). The first instance sets the level at 1. Specify up to three instances.

Supported server platforms: MVS, USS.

Example:

The following command sets the verbosity level at 2: **rccbuild… –V –V**

Setting the Condition for Valid Return Codes

–n *good_rc*

Specifies a comparison value for determining whether the return code from a build run signals success. The default value is 0. The **–n** option works in conjunction with the **–c** option.

Supported server platforms: MVS, USS.

–c *condition*

Specifies the comparison operator for determining whether the return code from a build run signals success. The **–c** option works in conjunction with the **–n** option.

The comparison operators include the following:

LT (less than)
LE (less than or equal to)
GT (greater than)
GE (greater than or equal to)
EQ (equal to)
NE (not equal to)

Supported server platforms: MVS, USS.

Examples:

rccbuild options	**Return code**	**Success?**
–n 4 –c LT	4	No
–n 4 –c LE	4	Yes
–n 4 –c GT	3	No

Specifying a Time-out Factor

–T *timeout*

Specifies the number of minutes that the server waits for an invoked build script to return before stopping the build event. The minimum time-out interval is 5 minutes.

Supported server platforms: MVS, USS.

Setting Environment Variables

–v var=*value…*

Specifies the list of variables and their values that are used to modify the build environment. Variable names are limited to 30 characters.

Supported server platforms: MVS, USS.

Build environment variables are used differently in MVS and USS.

In MVS, the **–v** option works in conjunction with user-defined variables on a DD statement. For more information about the MVS implementation, see *Using User-Defined Variables* on page 52.

In USS, the **–v** option changes or sets an environment variable. It is the equivalent of using the C-language command **putenv()**.

Passing Variables to Build Scripts

–p *build_parameters*

Specifies parameters that are passed to the build script. Build parameters are used differently by MVS and USS servers.

Supported server platforms: MVS, USS.

In MVS, the **–p** option works in conjunction with the PARM parameter on a DD statement. For more information about the MVS implementation, see *Using Predefined Variables* on page 50.

For UNIX platforms, specify two hyphens (- -) instead of one (-) when you need to pass a hyphen to your build script. This enables the server to distinguish between **rccbuild** parameters and your build script parameters.

Example:

The passed values of the **–p** option are **–o hello hello.c,** which are preceded by an extra hyphen. If the hyphen is omitted, the values are interpreted by the **rccbuild** command.

rccbuild –b cc –i hello.c –o hello -p --o hello hello.c

Specifying Prefix for Messages Returned to Client Log File

–P *prefix*

Specifies a prefix that is added to messages that the server returns to the client log file.

Supported server platforms: MVS, USS.

Example:

Prefix	Server message file	Client message file
122500	`COMPILE.SYSPRINT`	`122500COMPILE.SYSPRINT`

Using the –i, –o and –d Options with USS Builds

The file specifications on the **–o** and **–d** options are interpreted differently by the client and server.

Specifying Input and Dependent Files

The client treats file locations specified with the **–i** and **–d** options as absolute or relative to the client's current directory. The server places input and dependent files in a subdirectory relative to where the server is running. The server has a concept of build location, which is the directory the server uses as the current working directory.

- To use the default location, which is a subdirectory relative to the directory in which the Remote Build server starts, omit the **–l** and **–la** options. The subdirectory is deleted after the build request completes.
- To force Remote Build server to create a subdirectory relative to a specified directory, use the **–l** option. The relative is not deleted unless the **–lc** options are specified.
- To specify an absolute path, use the **–la** options.

Input File Examples

command	Description
rccbuild… –it foo.c	The client reads the file **foo.c** from its current directory. The server creates a temporary directory (typically named **tb***nnnn*) and creates **foo.c** there. At the end of the build, the server deletes the directory.
rccbuild… –it foo.c –l MyDir	The server creates or reuses the subdirectory **MyDir** as the build location, and does not delete it at the end of the build.
rccbuild… –it foo.c –la /u/server/test	The server uses the absolute directory **/u/server/test** as the build location. The directory is not deleted after the build.
rccbuild… –it foo.c –lc MyDir	The server deletes the directory **MyDir** after the build.

command	Description
rccbuild… –it temp/temp1/foo.c	When you omit the **–l** option, the server creates a temporary directory path that is relative to the directory where you start the server. The name of the temporary directory varies. For example, you start the server in the directory **RemoteBuild**. The server creates a subdirectory beneath it, such as **tmp0001**. Given the example **rccbuild** command, the server then creates the directory path **temp/temp1** beneath **tmp0001**. The file foo.c is copied to the directory **temp1**. The full path is: **/RemoteBuild/tmp0001/temp/temp1/foo.c**
rccbuild… –it temp/temp1(foo.c	The client interprets the left parenthesis as a slash (/) and finds the appropriate directory. You can use the left parenthesis in place of a slash anywhere in the path. On the server, the left parenthesis forces the file **foo.c** to be created in the current directory, not in a subdirectory.

Specifying Output Files

For output files, the client and server work similarly.

Output File Examples

command	Description
rccbuild… –o foo.obj	The server reads the file from the build location and returns it to the client. The client places the file in its current directory.
rccbuild… –o c:\output\foo.obj	The server reads the output **\foo.obj** relative to the build location. The client creates the file **C:\output\foo.obj**.
rccbuild… –o c:\output(foo.obj	The server reads the file **foo.obj** relative to the build location. The client creates the file **C:\output\foo.obj**.
rccbuild… –o output(foo.obj	The server reads **foo.obj** relative to the build location, and the client creates the file output**\foo.obj** in the current directory.

Working with Build Scripts 5

This chapter describes how to create **JCL** build scripts for MVS builds.

For builds on the USS platform, you can use a makefile or an executable on USS.

Identifying Build Scripts at Run Time

You identify the build script with the client command **rccbuild**.

- To specify a build script that is stored on the client machine:

  rccbuild –ft *build_script* **–b** *server_filename*

  where *build_script* is the local file and *server_filename* is the copy-to name when the file is transferred to the server.

- To specify a build script that is stored on the server machine:

  rccbuild –b *server_script*

  In MVS, the script must be stored in the PDS that is associated with the **RCCPROC** DD statement of the RCCMVS member. This PDS must have the following attributes: **RECFM=FB**, **LRECL=80**.

Understanding JCL Build Scripts

You must write JCL build scripts using pseudo JCL for Remote Build. Each file must have an LRECL of 80 characters. Any lines with more than 80 characters are truncated during the transfer process. Pseudo JCL syntax is similar to standard JCL, with some extensions and restrictions. We recommend starting with an existing JCL fragment.

Understanding Coding Requirements

The key coding requirements include the following items:

- Omit a JOB statement.
- Start all statements, except in-stream data, with two slashes — //.

- Identify the following build files using DD statements with either **RCCEXT** parameter:
  - Input source files
  - Dependent files
  - Output files
  - Output listings

Testing Scripts on the Mainframe

We strongly recommend testing JCL build scripts directly on your mainframe before submitting them remotely using Remote Build

Identifying Build Files

To identify files that Remote Build processes, you need to customize your build script and, depending on the type and location of a file, specify a client command option. This coordination is required to send output files, such as object modules and executables, to the client machine where they can be checked in to a ClearCase view.

The DD statement parameter **RCCEXT** identifies the file extension of a build file used in a specific I/O operation.

The following table describes JCL script and client command requirements.

Files	**DD statement syntax**	**rccbuild option**
Input files that reside on the client machine	**//SYSIN DD RCCEXT=***ext…*	**–i**
Dependent files that reside on the client machine	**//***ddname* **DD RCCEXT=***ext*	**–d**
Output files to be sent to the client machine	After a compile step: **//SYSLIN DD RCCEXT=***ext…* After a link-edit step: **//SYSLMOD DD RCCEXT=***ext…*	**–o** (Required)
Output listings to be sent to the client machine	**//SYSPRINT DD RCCEXT=RCCOUT**	(Not applicable)

Sample Scenarios

This section describes several build scenarios.

Input File on Client Machine

The input file **BANNER.CBL** resides in a Windows directory.

Sample rccbuild Command

Use the **–i** option to specify the input file.

rccbuild -h… –i C:\MYCOBOL\BANNER.CBL –b… –f…

Sample DD Statement

```
//SYSIN    DD  RCCEXT=CBL,DISP=(NEW,DELETE),
//     UNIT=VIO,SPACE=(TRK,(10,10)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
```

Dependent File on Client Machine

The dependent file **BANNER.LED** resides in a Windows directory.

Sample rccbuild Command

Use the **–d** option to specify the dependent file.

rccbuild -h… –d C:\MYHEADERS\BANNER.LED –b… –f…

Sample DD Statement

```
//SYSLIN    DD  RCCEXT=LED,DISP=(NEW,DELETE),
//     UNIT=VIO,SPACE=(TRK,(10,10)),
//     DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
```

Output File, Link-Edit Step

The generated executable **BANNER** is sent to the client as **BANNER.LOD**.

Sample rccbuild Command

Use the **–o** option to specify the output file.

rccbuild -h… –i C:\MYCOBOL\BANNER.CBL –o BANNER.LOD –b… –f…

Sample DD Statement

```
//SYSLMOD    DD  RCCEXT=LOD,DISP=(NEW,DELETE),
//     UNIT=VIO,SPACE=(CYL,(10,10)),
//      DCB=(RECFM=U,LRECL=0,BLKSIZE=6233)
```

Coding the EXEC Statement

Use the **EXEC** statement for these purposes:

- To define a new job step
- To specify the name of a load module or build script
- To define parameters whose values you pass from the client

Syntax:

//stepname **EXEC** **[PGM**=*program_name* | *proc_name*] **[PARM**=’*parm_string* |
COND=(*code*,*operator*[,*stepname*])

where:

program_name

Specifies a load module.

proc_name

Specifies a build script whose location is identified by the **RCCPROC** DD statement in the **RCCMVS** JCL member.

PARM='*parm_string*'

Specifies a parameter string or variable. To pass a value for a user-defined variable from the client, use the **rccbuild –v** option.

A parameter string can contain imbedded blanks and quotes. To imbed a single quote, concatenate two single quotes.

Imbedded Quote Example

Your build script has the parameter:

PARM='&X''s'

You enter the following client command:

rccbuild ... -v X=it

The script value expands to:

PARM='it's'

Variables Example

To specify a variable, type an ampersand (&) followed by the variable name.

For example:

'&X'

COND=(*code*,*operator*[,*stepname*]))	Specifies a condition to test before executing the current step. You can code multiple conditions per EXEC statement.
	The parameter *code* is the value to test against the return code from a previous job step.
	The parameter *operator* is the comparison operator.
	The parameter *stepname* identifies the job step that issues the return code.
	For example:
	
	//STEP1 EXEC PGM=ONE
	...
	//STEP2 EXEC PGM=TWO
	...
	//STEP3 EXEC PGM=THRE,COND=(4,LE,STEP1 COND=(4,LE, STEP2)

Coding the DD Statement

Use the DD statement to describe data sets, including source, dependent, and output files.

Syntax:

//ddname **DD**
[**DISP** | **DISP**=(*status*,*disposition*=) | **DCB**=*value* | **DSN**=*value* | **DSNAME**=*dataset* | **DSORG**=*dataset_organization* | **SPACE**=*value* | **UNIT**=*type* | **VOL**=**SER**=*volume*] | [**RCCEXT=***ext* | **RCCEXT=(***ext1* , *ext2* , ...**)** | **RCCEXT=RCCOUT** | **RCCEXT=RCCSTD** | **RCCEXT=RCCERR**] | *

where the following standard JCL variables must be adapted for use with Remote Build:

ddname	Specifies the DD name.
status	Valid Remote Build values include: NEW, OLD, DELETE, SHR

disposition	Valid Remote Build values: DELETE, KEEP, CATLG, UNCATLG.
unit_type	The default is VIO.

For more information about the **RCCEXT** parameter, see *Identifying Files Using RCCEXT DD Parameters* on page 48.

Identifying Files Using RCCEXT DD Parameters

You must identify input files, dependent files, output files, and output listings with a DD statement and a **RCCEXT** parameter.

Identifying Input Files

Include a **SYSIN** DD statement for each input file that you pass using the **–i** option.

Change

//SYSIN DD DSN=

to

//SYSIN DD RCCEXT=*ext*

where *ext* is the file extension for the input file, such as CBL or C.

Identifying Dependent Files

Include one DD statement for one or more dependent files, such as header files and COBOL copybooks. Use one of the following formats:

- *//ddname* **DD RCCEXT=***ext*
- *//ddname* **DD RCCEXT=(***ext1*, *ext2*, ...*extN***)**

The following DD statement specifies that all dependent files with extension **.h** and **.hpp** (case- insensitive) are placed in the datas et allocated to the ddname **USERLIB**. The same extension can appear only once in the JCL script.

```
//USERLIB DD DSN=MY.HEADERS,DISP=SHR,RCCEXT=(H,HPP)
```

Sending Output Messages to a Client File

Use the RCCOUT extension to send output messages to the client, in a file called *prefix.stepname.ddname*.

where:

prefix	The value, if any, specified with the **rccbuild –P** option.
stepname	The step name in the EXEC statement.
ddname	The DD name in the DD statement.

In a build script run that omits the **–P** option, Remote Build overwrites an existing SYSOUT file called *stepname.ddname*. By using the **–P** option, you can create and keep message files from multiple build script runs. This is useful when more than one source program (**-i**) uses the same build script. For example, you might use the program name as the **–P** option.

When you use the **RCCOUT** extension with the **SYSOUT** ddname, COBOL DISPLAY messages are included.

In the following example, after the program CBCDRVR executes, the contents of SYSOUT are transferred to the client as file COMPILE.SYSOUT. This assumes that the **–P** option is omitted.

```
//COMPILE EXEC PGM=CBCDRVR,..
//SYSIN…
//SYSOUT DD RCCEXT=RCCOUT,DISP=(NEW,DELETE),SPACE=(32000,(30,30)),
//  DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
```

Sending Output Messages to the Client’s Screen and a File

Use the **RCCERR** or **RCCSTD** extension to send output messages to the client console and in a file called *prefix.stepname.ddname*.

where:

prefix	The value, if any, specified with the **rccbuild –P** option.
stepname	The step name in the EXEC statement.
ddname	The DD name in the DD statement.

In a build script run that omits the **–P** option, Remote Build overwrites an existing SYSOUT file called *stepname.ddname*. By using the **–P** option, you can create and keep message files from multiple build script runs. This is useful when more than one source program (**-i**) uses the same build script. For example, you might use the program name as the **–P** option.

In the following example, after the program CBCDRVR executes, the contents of SYSOUT are transferred to the client as file COMPILE.SYSOUT. This assumes that the **rccbuild –P** option is omitted.

```
//COMPILE EXEC PGM=CBCDRVR,..
//SYSIN…
//SYSOUT DD RCCEXT=RCCSTD,DISP=(NEW,DELETE),SPACE=(32000,(30,30)),
//  DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
```

Using Variables

The parameters of a DD statement can have variables similar to standard JCL. Variable names must start with an ampersand (&) and contain alphanumeric characters. They are terminated by a nonalphanumeric character or a period (.), if needed. Variable names are limited to 30 characters.

The pseudo JCL supports predefined and user-defined variables.

Using Predefined Variables

The following variables are predefined:

&INPUT

Returns the names of input files, passed by the **rccbuild –i** option. The path and file extensions are discarded, and names are converted to MVS-compatible names.

Example:

If the input files include **src/hello.obj hello1.obj**, the **&INPUT** variable returns **HELLO HELLO1**.

&OUTPUT

Returns the names of output files, passed by the **rccbuild –o** option.

Example:

If the output file is **src/hello.obj,** the **&OUTPUT** variable returns **HELLO**.

&DEP

Returns the names of dependent files, passed by the **rccbuild –d** option.

Example:

If the dependent file is **header/stdout.h,** the **&DEP** variable returns STDOUT.

&PARM

Returns the value of a parameter, passed by the **rccbuild –p** option. This string is passed as is (without folding). In the **EXEC** statement, the **&PARM** variable must be enclosed in single quotes.

Example:

The command issued on the client machine:

rccbuild ... –p TYPERUN=DEBUG

The corresponding command located in the build script:

```
//COMPILE EXEC PGM=COMPILER,PARM='&PARM'
```

The server performs the variable substitution and changes the EXEC statement:

```
//COMPILE EXEC PGM=COMPILER,PARM='TYPERUN=DEBUG'
```

&COMMA

Returns a comma.

&SP

Returns a single space.

The following behavior is associated with in-stream statements that contain predefined variables **&INPUT**, **&OUTPUT,** and **&DEP:**

The statement that contains the variable is repeated for each file associated with the variable.

Example:

Your build script has this DD statement:

```
//SYSLIN DD *
INCLUDE OBJ(&INPUT)
/*
```

You enter the following client command:

rccbuild ... –i hello.obj hello1.obj

The server expands the input stream to this:

```
//SYSLIN DD *
INCLUDE OBJ(HELLO)
INCLUDE OBJ(HELLO1)
/*
```

Using User-Defined Variables

To pass user-defined variables, use the **rccbuild –v** option.

Example:

Your build script has the DD statement:

```
//OBJ  DD DISP=SHR,DSN=&USERID..OBJ
```

You enter the following client command:

rccbuild ... –v USERID=QEORD1

The script value expands to:

```
//OBJ  DD DISP=SHR,DSN=QUEORD1.OBJ
```

Setting Defaults for User-Defined Variables

Using a **VARS** statement, you can set default values for user-defined variables. The **VARS** statement defines a comma-delimited list of name-value pairs.

//*label* **VARS** *name1***=***value1***,…***nameN***=***valueN*

where:

label	Specifies a label for the statement. The label has the same constraints as a DD name.
name1	Specifies the name of a user-defined variable.
value1	Specifies the default value of the variable.

Example:

In the following example, default values are set for two variables.

```
//PRODVAR  VARS USER=USER01,HLQ=V40021
```

File Name Conversions for MVS

Client file names must conform to these rules:

- Names must contain the following valid MVS characters:

  0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ$@#

- Names must begin with an alphabetic character.

The server makes the following transformations:

- 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 discarded.
- Lowercase characters are converted to uppercase characters.
- The file extension is stripped from the right, up to and including the separating period. The extension, minus the period, is used by the MVS server to direct the file to particular data sets according to **RCCEXT** parameters in the JCL build scripts.
- The remaining name is truncated from the left, to a maximum of eight characters.
- Underscore characters (_) in a file name are converted to at signs (@).

Conversion Examples

The following examples demonstrate file name conversions:

- File name **src\build\fhbldobj.C** converts to FHBLDOBJ.
- File name **src/build/fhbtruncate.c** converts to FHBTRUNC.

Using Remote Build with clearmake

6

The **clearmake** utility is the Rational ClearCase variant of the UNIX **make** utility. Using **clearmake**, you can audit remote builds and trigger future build events.

During the process of building executables and load modules, ClearCase tracks the following actions:

- One or more source files that are under source control in a VOB are opened, read, and sent to the Remote Build server.
- Other files are created or updated as a result of the processes running.

An audit record indicates that the updated files are dependent upon files that were read. When source files change and you reprocess the makefile, **clearmake** knows which derived objects need to be recompiled.

Creating a makefile for a Remote Build

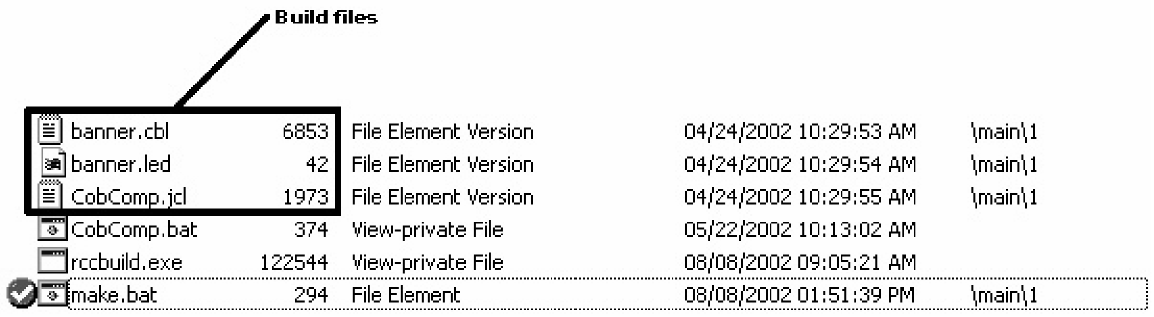
To create a makefile that integrates with Remote Build, replace build script commands with a **rccbuild** command string.

In the following example, the file **banner.cbl** is compiled to generate the object module **BANNER**. The object file is link-edited, and the generated load module returns to the client as **banner.pro**. Only the link-edit step needs to be reflected in the first statement.

```
banner.pro: banner.cbl
rccbuild -h os390@3604 -b cobcomp -ft cobcomp.jcl -k IBM-850 \
-r IBM-037 -it banner.cbl -dt banner.led -o banner.pro -v MBR=BANNER \
COBCOMP=IGY210.SIGYCOMP LERUN=CEE150.SCEELKED HLQ=SMITH \
SYSTEM=MVSCICS -V -V -V
```

The following Windows example shows the versioned files that are used to generate the load module **BANNER**.

Figure 5 Build Files in ClearCase Explorer



banner.cbl	6853	File Element Version	04/24/2002 10:29:53 AM	\main\1
banner.led	42	File Element Version	04/24/2002 10:29:54 AM	\main\1
CobComp.jcl	1973	File Element Version	04/24/2002 10:29:55 AM	\main\1
CobComp.bat	374	View-private File	05/22/2002 10:13:02 AM	
rccbuild.exe	122544	View-private File	08/08/2002 09:05:21 AM	
make.bat	294	File Element	08/08/2002 01:51:39 PM	\main\1

Running the makefile

To run the makefile, use a **clearmake** command. For example:

clearmake –f *makefile*

After the makefile is run, the clearmake utility creates an audit record that indicates that **banner.pro** depends upon the three files read: **banner.cbl**, **banner.jcl**, and **banner.led**. When you rerun the makefile, the build is executed again only if one or more dependent files have changed. If all of them remain unchanged, the build request is not submitted to the mainframe.

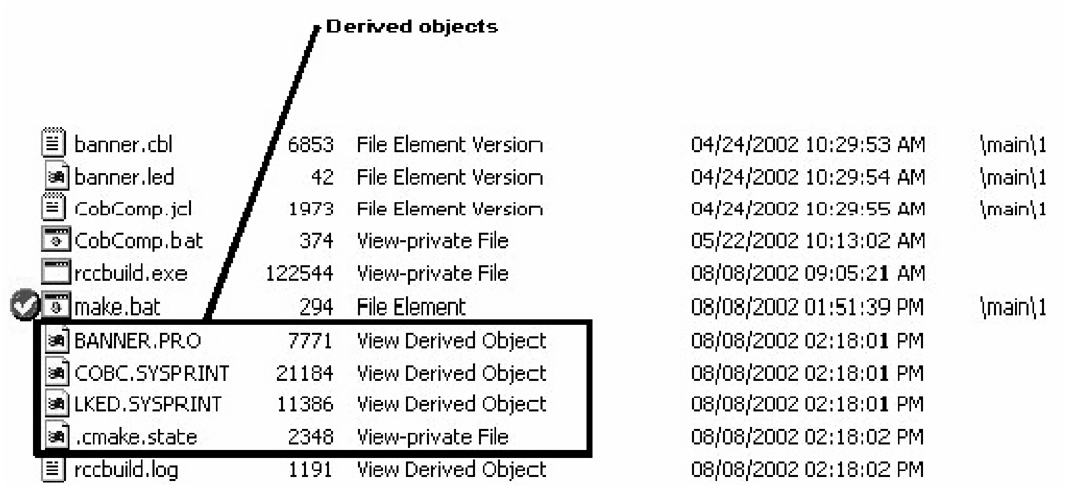
Returning Derived Objects to the Client

To return a derived object to the client:

1. Specify the derived object using the **rccbuild –o** option. The default format is binary. If the file is in text format, specific the **–ot** option.
2. Include the file extension of the derived object using the **RCCEXT** extension parameter in your build script. For more information about extension parameters, see *Identifying Build Files* on page 44.

The following Windows example shows the derived object, load module **BANNER.PRO**, and three other files: log file **rccbuild.log**, and two **SYSPRINT** message listings.

Figure 6 Derived Objects in ClearCase Explorer



banner.cbl	6853	File Element Version	04/24/2002 10:29:53 AM	\main\1
banner.led	42	File Element Version	04/24/2002 10:29:54 AM	\main\1
CobComp.jcl	1973	File Element Version	04/24/2002 10:29:55 AM	\main\1
CobComp.bat	374	View-private File	05/22/2002 10:13:02 AM	
rccbuild.exe	122544	View-private File	08/08/2002 09:05:21 AM	
make.bat	294	File Element	08/08/2002 01:51:39 PM	\main\1
BANNER.PRO	7771	View Derived Object	08/08/2002 02:18:01 PM	
COBC.SYSPRINT	21184	View Derived Object	08/08/2002 02:18:01 PM	
LKED.SYSPRINT	11386	View Derived Object	08/08/2002 02:18:01 PM	
.cmake.state	2348	View-private File	08/08/2002 02:18:02 PM	
rccbuild.log	1191	View Derived Object	08/08/2002 02:18:02 PM	

The **SYSPRINT** and log files are sent to the directory from which you run the **rccbuild** command. If you run the **rccbuild** command from a directory other than the view that contains the source files, direct the derived object and other output files to the view by specifying an output path (**–o** *path*).

Sample Build Files

A

This appendix demonstrates how to submit two types of remote requests using the client command **rccbuild**:

- Building a COBOL load module.
- Running the COBOL load module.

Each process generates output files that are returned to the client machine.

About the Sample Files

Table 3 describes the sample files.

Table 3 Sample Files

Sample File	Description
banner.cbl	Source code for a sample COBOL program that displays the Rational logo.
banner.led	Linkage Editor control statements. This file is passed as a dependent file to the **rccbuild** command.
cobcomp.bat	Batch file that runs the **rccbuild** command to submit a build request to MVS.
cobcomp.jcl	JCL script that invokes the COBOL compiler and Linkage Editor in MVS.
runscr.bat	Batch file that runs the **rccbuild** command to submit the JCL file **runscr.jcl**.
runscr.jcl	JCL script that executes the BANNER load module.

Submitting the COBOL Build Request

This section describes the following:

- Editing the **rccbuild** command within the batch file **cobcomp.bat**.
- Running the batch file on the client machine.

Editing the Batch File

The batch file **cobcomp.bat** contains the following **rccbuild** command:

rccbuild –h *servername@portno* **–b cobcomp –ft cobcomp.jcl –k IBM-850 –r IBM-037 –it banner.cbl –dt banner.led –v MBR=BANNER COBCOMP=***cobol_lib* **LERUN=***langenv_lib* **HLQ=***hlqname* **SYSTEM=***hlq2name* **–V –V –V**

This **rccbuild** command passes a JCL script, COBOL source, Linkage Editor statements, and values for user-defined variables to the server.

Edit the batch file:

Change	**To**
servername@portno	The MVS server name, followed by the at sign and the listening port for the Remote Build server.
cob_lib	Your COBOL library name.
langenv_lib	Your Language Environment library name.
hlqname	The high-level qualifier for your object and load libraries.
hlq2name	The second-level qualifier for your object and load libraries.

AIX and Linux/390 users, remove these codepage parameters:

–k IBM-850 –r IBM-037

Understanding the User-Defined Variables in the Build Script

The **–v** option in the sample **rccbuild** command supplies values for user-defined variables that are declared in the build script **cobcomp.jcl**. The variables are highlighted in the following example.

```
//COBC   EXEC PGM=IGYCRCTL,REGION=4096K,
...
//STEPLIB  DD  DISP=SHR,DSN=&COBCOMP
...
//SYSLIN   DD  DISP=SHR,DSN=&HLQ..&SYSTEM..OBJECT(&MBR)
...
//          DD  DISP=SHR,DSN=&LERUN
...
```

Running the Batch File

To run the batch file on the client:

1 Update your system search path, if needed, to include the directory that contains the executable **rccbuild**.

2 Make the directory that contains the sample files the current directory.

3 At the command prompt, enter the following:

cobcomp.bat

The build server returns two output files, whose names are derived from information in the COBCOMP JCL:

- **COBC.SYSPRINT** contains COBOL compiler messages. **COBC** is the step name on the **EXEC** statement that calls the COBOL compiler.
- **LKED.SYSPRINT** contains Linkage Editor messages. **LKED** is the step name on the **EXEC** statement that calls the Linkage Editor.

These files are returned to the client because **COBCOMP JCL** also has **SYSPRINT DD** statements that include the extension parameter **RCCEXT=RCCOUT**.

Running the COBOL Load Module

To execute the COBOL module in MVS, you can run the **rccbuild** command.This section describes the following:

- Editing the **rccbuild** command within the batch file **runscr.bat**.
- Running the batch file on the client machine.

Editing the Batch File

The batch file **runscr.bat** contains the following **rccbuild** command:

rccbuild –h *servername@port* **–b runscr –ft runscr.jcl –k IBM-850 –r IBM-037 –v HLQ=***hlqname* **SYSTEM=***hlq2name*

This **rccbuild** command passes JCL and values for user-defined variables to the server.

Edit the batch file:

Change	**To**
servername@portno	The MVS server name, followed by the at sign and the listening port for the Remote Build server.

hlqname	The high-level qualifier for your object and load libraries.
hlq2name	The second-level qualifier for your object and load libraries.

AIX and Linux/390 users, remove these codepage parameters:

–k IBM-850 –r IBM-037

Running the Batch File

To run the batch file on the client:

1 Update your system's search path, if needed, to include the directory that contains the executable **rccbuild**.

2 Make the directory that contains the sample files the current directory.

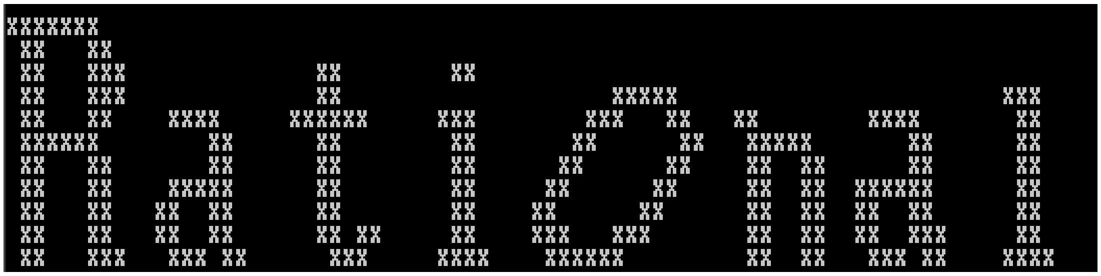
3 At the command prompt, enter the following:

runscr.bat

The build server returns two output files, whose names are derived from information in the RUNSCR JCL:

- **RUNLOG.SYSPRINT** is an empty file. **RUNLOG** is the stepname on the **EXEC** statement that calls **BANNER**, the COBOL load module.
- **RUNLOG.SYSOUT** contains the Rational logo, as shown below.

These files are returned to the client because **RUNSCR JCL** also has **SYSPRINT** and **SYSOUT DD** statements that include the extension parameter **RCCEXT=RCCOUT**.



```
XXXXXXX
 XX   XX
 XX   XXX              XX        XX
 XX   XXX              XX                    XXXXX                        XXX
 XX   XX    XXXX     XXXXXX     XXX        XXX   XX   XX        XXXX       XX
 XXXXXX        XX      XX        XX       XX      XX   XXXXX       XX      XX
 XX   XX       XX      XX        XX      XX      XX    XX  XX      XX      XX
 XX   XX    XXXXX      XX        XX     XX      XX     XX  XX  XXXXXX      XX
 XX   XX   XX  XX      XX        XX    XX      XX      XX  XX  XX  XX      XX
 XX   XX   XX  XX      XX XX     XX    XXX   XXX       XX  XX  XX  XXX     XX
 XX   XXX   XXX XX      XXX     XXXX    XXXXXX         XX  XX   XXX XX    XXXX
```

Sample rccbuild Commands

B

This appendix describes several sample **rccbuild** commands. It also demonstrates that command options are not positional. For more information on **rccbuild** command options, see Chapter 4, *Sending a Build Request*.

Sample Commands

- Ping a Remote Build server called **os390**. No build request is passed.

  rccbuild –testServer –h os390@42310

- Return the version of the client executable **rccbuild**.

  rccbuild –version

- Send local files (JCL and COBOL program) to the server for processing.

  rccbuild –b rcccomp –ft rcccomp.jcl –i banner.cob –h os390@42310

- Set a time-out factor of one minute for starting the previous build request.

  rccbuild –h os390@42310 –T 1 –ft rcccomp.jcl –i banner.cob –b rcccomp

- Send TSO login details to the server. If the login details are valid and the user has access to the required libraries and directories, the makefile is processed.

  rccbuild –h prodserv@24434 –au BOSMA01 –ap DEL34 –ft helpux.mak –b hlp

- Point to an MVS-based library that contains thebuild script **INVMAIN**.

  rccbuild –h prodserv@24434 –proclib ACPDEV.LONDON.JCL –b INVMAIN

- Pass values for user-defined variables for a script through the **–v** option. The command also identifies input (**–i**) and dependent (**–d**) files.

  rccbuild –h os390@55323 –b cobcomp –ft cobcomp.jcl –it banner.cbl –dt banner.led –v MBR=BANNER COBCOMP=MYCOB.LIB LERUN=MY.LE.LIB HLQ=HAZLTON SYSTEM=INVENT

- Set up a line prefix for messages that are recorded in the client log (**rccbuild.log**) during an MVS build run.

rccbuild –h os390@47123 –b rcccomp –ft rcccomp.jcl –i banner.cob –P MONDAYJSMITH

- Set the maximum verbosity for messages recorded in the file **rccbuild.log**.

  rccbuild –h prodserv@24434 –proclib ACPDEV.LONDON.JCL –b INVMAIN –V –V –V

- Use the **–c** and **–n** options to set a test for continuing processing. It also demonstrates overlaying the two options with new values.

  rccbuild –h os390@4602 –b cob –ft cobcomp.jcl –it banner.cbl –dt banner.led –c GT –c EQ –n 10 –n 0

Index

&COMMA variable 51
&DEP variable 50
&INPUT variable 50
&OUTPUT variable 50
&PARM variable 51
&SP variable 51
@ signs in filenames 53

A

absolute directory paths 40
APF-authorized libraries 18
authenticating users
  in TSO 18
authentication users
  setting level for 18

B

BPXBATCH utility
  running Remote Build script with 25
build environment
  passing variables to 38
build requests
  handling multiple MVS 16
  handling multiple USS 16
  queuing 17
build scripts
  indicating build files in 44
  passing parameters to 39
  remote 32
  specifying 43
  specifying local 32
build server
  passing login information 36
  pinging 32
builds 35
  codepages for 37
  dependent files for 34
  input files for 34
  setting timeout factor 38
  working directory for (USS) 35

C

ccase-home-dir directory v
clearmake utility
  makefile example 55
  overview 55
client command
  repeating options 31
  submitting 43

client command (rccbuild) 29
client command examples 63
client software 5
client version 31
COBOL build example 59
codepages 37
command syntax, rccbuild 29
concurrency considerations 17
configuring MVS server 22
connectivity requirements 2
conventions, typographical v

D

DD statements
    user-defined variables for 38
deliverables
    MVS only 6
    USS 6
dependent files 34
    USS 40
derived objects, sample output 27

E

EBCDIC conversions 31
EXEC statement
    build requirements 46

I

imbedding rccbuild in makefile 55
input files 34
    MVS 34
    USS 40
installing 5
    Remote Build client 5
    Remote Build server 5
Installing Remote Build server
    overview 5
IVP 26

J

JCL build scripts
    overview 43
JCL members
    customization requirements 22
JOB statement
    in build scripts 43

L

Log files
    rccbldc 17
log files
    ccubldc 56
logging server messages 21
logging trace entries 21
login details 36
LRECL consideration, build scripts 43

M

makefile
    running 56

makefile example 55
message line prefix 39
multithreaded server 1
MVS deliverables 6
MVS server
  running 13

N

national language messages 6

O

operating systems, supported 2
output files
  returning to client 35
  USS 40
output messages 49
overriding JCL PDS 33

P

passing build script parameters 39
passing options to script 39
passing variables to build server 38
PDS locations 33
pinging server 32
port number
  selecting 25
  setting 26–28
predefined variables
  &COMMA 51
  &DEP 50
  &INPUT variable 50
  &OUTPUT 50
  &SP 51
prefix, message line prefix 39
prefixing server messages 39
prerequisites, server installation 7
proclib location, for JCL scripts 33

Q

queue size, setting 17
queuing build requests 17

R

RACF authentication 18
RCCBLDS module 6
RCCBLDW module 6
rccbldw script 25
rccbuild command options
  -ap 37
  -au 36
  -b 32
  -c 38
  -d 34
  -f 32
  -h 32
  -i 34
  -k 37
  -l 35
  -n 37
  -o 35
  -P 39
  -p 39
  -proclib 33
  -r 37
  -T 38

-testServer 32
-V 37
-v 38
-version 31
rccbuild command syntax 29
rccbuild commands, samples 63
RCCDLL module 6
RCCEXT extension parameters
examples 44
overview 20
RCCMAIN module 6
RCCMSENU module 6
RCCMSG module 6
RCCMVS member 22
RCCRUNM member 22
RCCUSS member
editing 26
region size requirement (MVS) 10
relative directory paths 40
Remote Build
processing overview 13
Remote Build client
requirements 2
remote build script 32
Remote Build server
installing 5
requirements 3
repeating rccbuild options 31
required rccbuild options 29
requirements
client 2
Remote Build server 3
return codes
logic for processing 37
returning derived objects to client 56
returning output to client 35
running MVS server 13
running USS server 13

S

sample input file 27
sample output file 27
sample rccbuild commands 63
security
for MVS objects 18
server machine 32
server messages
logging 21
server port 32
server timeout 38
SMP/E for MVS installation
RECEIVE-APPLY-ACCEPT 9
runtime parameters 8
SMP/E libraries 8
SMP/E for USS installation
DD definitions 10
RECEIVE-APPLY-ACCEPT 10
runtime parameters 9
spawning build processes 16
starting the MVS server 24
starting the USS server 26
stopping
MVS server 25
USS server 26

submitting local build scripts 32
submitting the startup job (USS) 26
Synopsis (rccbuild command) 29

T

TCP/IP
    sockets 17
TCP/IP requirement 2–3
technical support vi
timeout factor for builds 38
trace entries
    logging 21
typecase conversions 31
typographical conventions v

U

underscores in filenames 53
UNIX working directories 40
USS deliverables 6
USS server
    installing 9
    running 13

V

verbosity (messaging) 37
verifying client/server communication 26
    MVS 26
    USS 27
version, client 31

W

working directories (UNIX) 40