A Developer's Guide to the D System v0.6

July 1994

dsystem-info@cs.rice.edu

Rice University
CITI/CRPC - MS 41
6100 South Main Street
Houston, Texas 77005-1892

http://www.cs.rice.edu/fortran-tools/DSystem/DSystem.html

A Developer's Guide to the D System v0.6

This document was prepared by Rice University for the Advanced Research Projects Agency (ARPA) under contract number DABT63-92-C-0038.

Contents

List of Figures

Introduction

1.1 About This Release

The D System v0.6 is a collection of prototype tools aimed at supporting the development of programs written in Fortran D, an abstract, machine-independent parallel programming language. Release v0.6 should be viewed as a snapshot of work in progress. Release v0.6 of the D System includes:

- The Fortran D editor, an intelligent browser for Fortran 77D that provides feedback on the analysis and parallelization performed by the Fortran 77D compiler. The Fortran D editor will support editing in a later release.
- The Fortran 77D compiler, which translates sequential Fortran 77 programs with data decomposition directives into message-passing Fortran 77 programs. This prototype is suitable for experimentation on application kernels and modest benchmarks that use 1D BLOCK data distributions.
- A demo binary of an enhanced D editor that includes a preliminary integration of feedback about dynamic performance into the editor along with a new X/Motif user interface. The dynamic performance analysis work is the result of an ongoing collaboration with the University of Illinois' Pablo project. For more information about the Pablo project, see the Pablo World Wide Web pages (http://www-picasso.cs.uiuc.edu/Projects/Pablo/pablo.html) or contact pablo@guitar.cs.uiuc.edu.
- An integrated framework for demand-driven interprocedural analysis which was designed to support both batch and interactive tools for whole-program analysis.

1.2 About This Document

A Developer's Guide to the D System v0.6 presents an overview of the installed directory, private directories, and the D Developer's Toolkit, plus limited instructions for modifying source code. To successfully modify the D System source code, you will need some knowledge of

- Unix
- C
- C++
- Fortran
- revision control principles
- parallel computation

To help you become more familiar with the D System, we recommend the following related documentation.

Related Documentation

Users with Internet access can obtain more information about the D System via the World Wide Web or by contacting dsystem-info@cs.rice.edu. Online documentation includes

- Technical papers. (http://www.cs.rice.edu/fortran-tools/DSystem)
- *The D System v0.6 Installation Guide*: step-by-step instructions for retrieving and installing the D System. (http://www.cs.rice.edu/fortran-tools/DSystem/v0.6)
- *The D System Hypertext Directory Tour*: an automated, surfable index of the installed world directory structure. (http://www.cs.rice.edu/fortran-tools/DSystem/v0.6)
- D System man pages: an online summary of each of the D System tools provided in this release. (DSystem/man, DSystem/tools/develop/man)
- The Fortran Tools Glossary: a collection of the terms indigenous to parallel computation and the D System. (http://www.cs.rice.edu/fortran-tools)

This document is the result of a collaborative effort between D System developers and their support staff. To "meet" the people behind the D System, see the D System pages on the World Wide Web (http://www.cs.rice.edu/fortran-tools/DSystem) or contact dsystem-info.cs.rice.edu.

1.3 About the CRPC

The Center for Research on Parallel Computation (CRPC), one of 25 NSF-funded Science and Technology Centers, is a consortium that includes Argonne National Laboratory, the California Institute of Technology, Los Alamos National Laboratory, Rice University, Syracuse University, and the University of Tennessee. Affiliated sites include the University of Illinois and the University of Maryland.

The CRPC is committed to a program of research that will make parallel systems truly usable—as usable as sequential machines are today. CRPC research focuses on developing algorithms that are portable across several parallel machines and providing compiler, editing, and debugging tools to make parallel machines easier to program.

Users with Internet access can obtain more information about the CRPC via the World Wide Web (http://softlib.rice.edu/) or by contacting crpc@cs.rice.edu.

The Big Picture

2.1 Introduction

In this section you'll get an overview of the development process at Rice. Section 2.2 briefly describes the relationship between the installed directory structure, a private directory structure, and the D Developer's Toolkit. Sections 2.3 and 2.4 define the installed and private directory structures, and Section 2.5 defines the D Developer's Toolkit, the D makesystem, and the d- and rn- naming convention.

2.2 The Big Picture

The D System source code resides in what is called the *installed directory structure*. However, each developer at Rice uses a separate directory structure to maintain local copies of the source code. This separate structure is called a *private directory structure*, because it is private to that developer. Futhermore, to access the installed directory structure from a private directory, developers at Rice use a set of tools provided in the *D Developer's Toolkit*. The general relationship between a private directory structure, the installed directory structure, and the D Developer's Toolkit is shown in Figure 1.

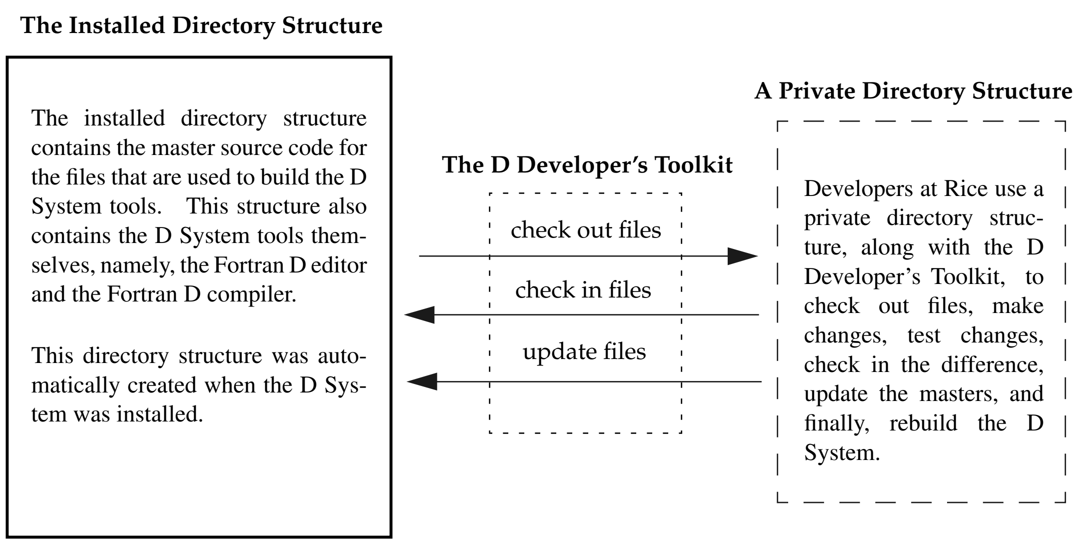

Figure 1: The general relationship between a private directory structure, the installed directory structure, and the D Developer's Toolkit.

2.3 The Installed Directory Structure

The installed directory structure is simply the directory structure that was automatically created when the D System was installed on your system. If you modify the D System, this is where you will ultimately store the most up-to-date versions of the source code. The upper levels of the installed directory structure, or *installed world*, are described in Section 3.

2.4 Private Directory Structures

At Rice, each developer's account contains one or more private directory structures, or *private worlds*. A private world, which partially replicates the directory structure of the installed world, contains copies of the source code files that a developer has created or is modifying. The upper levels of a private directory structure are described in Section 5. Instructions for creating your private world are provided in Section 6.

2.5 The D Developer's Toolkit

To modify the D System source code, you will need to access the installed directory structure with or without some kind of revision control. To make it easy for several developers to work on the D System at the same time, Rice created a set of scripts that run on top of GNU make. These scripts, together with RCS and GNU make, are collectively called the D Developer's Toolkit. Unless you decide to develop your own set of scripts, you will be using the D Developer's Toolkit with or without a private directory to modify the D System source code. The high-level components of the D Developer's Toolkit are discussed in Section 7.

The D Makesystem

The portions of the D Developers Toolkit that are used to build, or *make*, the D System are sometimes referred to as the D makesystem. Examples are dmake and makeall, which Rice developed as a set of scripts wrapped around GNU make. The internal components of the D makesystem are discussed in Section 4.

The d- and rn- Naming Scheme

Throughout the D Developer's Toolkit, you'll find utilities whose names begin with d. One example is dmake, which Rice developed as a set of scripts wrapped around GNU make. In general, the d- prefix indicates an RCS or GNU tool that has been customized or script-wrapped for the D System. Similarly, throughout the D System code, you'll find the prefix rn-. This is becaus the D System infrastructure grew out of the R^n programming environment, which was developed at Rice in the mid-eighties, and many rn- artifacts still remain in the D System code.

The Installed Directory Structure

3.1 Introduction

This section shows you how the D System's 1,000+ files are organized within the installed directory structure. For a more detailed tour of the installed directory structure, see the D System World Wide Web Hypertext Directory Tour (http://fortran-tools/DSystem/v0.6).

3.2 The Installed Directory Structure

The installed directory structure is simply the directory structure that was automatically created when the D System was installed on your system. Currently, the D System can be compiled for

- the Sparc architecture running SunOS 4.1.3
- the Sparc architecture running SunOS 5.3

Throughout the D System, the naming scheme emphasizes the operating system (OS) rather than the architecture, because the operating system determines what include files you're including, what libraries you're linking against, and what compilers you're running.

The upper levels of the installed directory structure are shown in Figure 3, and the upper levels of the DSystem/develop/src directory are shown in Figure 3. Some levels of this structure are briefly discussed in the following sections. For a more detailed tour of the installed directory structure, see the D System World Wide Web Hypertext Directory Tour (http://fortran-tools/DSystem/v0.6).

The DSystem/bin Directory

The DSystem/bin (Figure 3) directory contains one or more OS directories. These directories contain symbolic links to the Fortran D editor and compiler and their related tools for a specific operating system. For example, if the D System was installed under SunOS 5.x, the DSystem/bin/SunOS-5.x directory would contain symbolic links back to the D System executables in the DSystem/develop/obj/SunOS-5.x/execs directory.

The DSystem/tools Directory

The DSystem/tools directory (Figure 3) contains the development tools that you will use to build the D System. These development tools include scripts that were developed at Rice, as well as off-the-shelf tools, like perl and RCS, that were not included in the D System distribution, but were installed separately.

The DSystem/tools/bin directory contains OS-independent scripts that run on top of RCS and GNU make, as well as directories for the SunOS 4.x and SunOS 5.x operating systems. The DSystem/tools/develop/bin/{OS} directories contain symbolic links to the directories that contain GNU make, RCS, and perl. The DSystem/tools/develop/man directory contains the man pages for the tools in the D Developer's Toolkit.

DSystem/

- bin/ - symbolic links to the D System executables
- lib/ - runtime libraries, including fonts
- man/ - man pages for the D System executables
- examples/ - example files for the D editor/compiler
- docs/ - D System documentation
- tools/ - development scripts, GNU tools, and tool man pages
- develop/
  - src/ - master souce code for the D System (see Figure 3)
  - RCS/ - RCS files for DSystem/develop/src/
  - obj/ - compiled library archives

Figure 2: The upper levels of the installed directory structure.

The DSystem/develop/src Directory

The DSystem/develop/src directory (Figure 3) contains the master source code for the D System. If you make any changes to the source code, this is where you will ultimately store the most up-to-date versions. In specific, the DSystem/develop/src/libs directory and its subdirectories contain the main source code for the D System, which is compiled into multiple library archives. The DSystem/develop/src/execs directory and its subdirectories contain the main programs for the Fortran D editor and compiler and their related tools. The DSystem/develop/src/include directory and its subdirectories contain miscellaneous include files.

DSystem/develop/src/

- execs - source code for the D System executables
- include - miscellaneous include files
- libs - main source code for the D System
  - config
  - fileAttrMgmt
  - fortD
  - frontEnd
  - graphicInterface
  - ipAnalysis
  - moduleAnalysis
  - perfAnalysis
  - support

Figure 3: The upper levels of the installed DSystem/develop/src directory structure. For more information on these directories, see the D System Hypertext Directory Tour (http://fortran-tools/DSystem/v0.6).

The DSystem/develop/obj Directory

The DSystem/develop/obj directory (Figure 3) contains one subdirectory for each OS the code is compiled for, for example, SunOS-4.x or SunOS-5.x. Each subdirectory contains the master archives and executables for the D System for the corresponding OS.

The DSystem/develop/obj/{OS} directory structure is identical to the DSystem/develop/src directory structure. For example, if the D System was installed under SunOS 5.x, the DSystem/develop/obj/SunOS-5.x directory would emerge from the installation process with the subdirectories libs/, execs/, and include/.

Finally, the DSystem/develop/obj/{OS}/include directory and its subdirectories contain a few machine-generated files that depend on the specific operating system.

Archive & Executable Names

In general, the DSystem/develop/obj/{OS}/libs directory and its subdirectories contain the compiled library archives created from the DSystem/develop/src/libs source code. Here, archive names have the form

- `{archive name}-{OS}-ng.a`
- `{archive name}-{OS}-g.a`

where -ng.a denotes an archive without debugging information, and -g.a denotes an archive with debugging information. As of this release of the D System, archives with debugging information are built by default, whereas archives without debugging information are not. Furthermore, the OS is determined by the DSYSTEM_OS environment variable, which is discussed in Section 6.

The DSystem/develop/obj/{OS}/execs directory and its subdirectories contain the master executables. Here, executable filenames have the form

- `{executable filename}-{OS}-g`

where the -g denotes an executable with debugging information. As was the case with archive names, the OS is determined by the DSYSTEM_OS environment variable, which is discussed in Section 6.

Makesystem Components

4.1 Introduction

The portions of the D Developers Toolkit that are used to build, or make, the D System are called the D makesystem. This section discusses the internal components of the D makesystem, which exist throughout the installed directory structure. As you're modifying the D System source code, you will encounter the following files:

- Makeinclude
- Makeinclude.config
- Makefile
- Makeinclude.override
- Makeinclude.components

In order for the makesystem to work properly, all include files must be included in angle brackets (<>'s), not quotation marks. This forces the D makesystem to search all paths specified on the compile line, rather than the path of the last include. These makefiles use the environment variables discussed in Section 6.

Related Man Pages

The D makesystem uses the utilities dmake and dmakeall. For more information on these utilities, see their man pages (/DSystem/tools/develop/man/).

4.2 Makeinclude

The file Makeinclude, which contains machine-independent information, is the main driver for the D makesystem. This file resides in DSystem/develop/src and is never modified.

4.3 Makeinclude.config

The file Makeinclude.config, which resides in DSystem/develop/src, contains OS-specific information. When the D System was installed, this file was edited to contain the path of the compiler you're using. You would not need to edit this file unless you change compilers or other utilities that are used by the D makesystem. Makeinclude.config can be divided into five major sections. The first three sections contain machine-specific information, like

- specific variables concerning which compilers you want to use
- locations of X libraries
- specific commands for archives
- various extra compile-line flags

The fourth section contains the locations of scripts and tools used during each make, and the last section contains information about the GNU compilers.

4.4 Makefile

The DSystem/develop/src directory, and every directory below it, contains a file named Makefile. Each Makefile defines a set of Makefile variables, which are discussed in Section 7.3.

As you’re modifying the D System source code, if you need to add or delete a file, you’ll also need to modify one or more of the following files:

- Makefile
- Makeinclude.override
- Makeinclude.components

Instructions for adding and deleting files are provided in Section 7.3.

4.5 Makeinclude.override

Every directory that contains a Makefile also contains a file named Makeinclude.override. This file is used to override information in the Makefile from the current directory down. For example, Makeinclude.override can be used to turn certain options on or off during a make.

4.6 Makeinclude.components

The file Makeinclude.components contains archive names and paths and specifies the linking order for the archives.

4.7 Dependence Files

The D makesystem produces a file named `.d_`*filename* for every compilable file. These dependence files contain both a list of dependences for *filename* and a list of files that would be nearer than the current dependences. The dependence file for a given file resides in either the installed-world or private-world obj directory corresponding to the src directory where the source file lives. So the dependence file for, say, DSystem/develop/src/libs/support/tables/HashTable.C resides in DSystem/develop/obj/{OS}/libs/support/tables/.

When you run the dmake command, dmake uses DependenceVerify to check the creation date on the files listed in `.d_`*filename* and decides whether or not *filename* should be recompiled. The dependences are first determined with respect to DSystem/develop/src and then with respect to your private source directory. For example, if you add to your shadow world a header file that is intended to override its installed counterpart, dmake will remake the file that depends on the new header file.

Dependences are computed by taking the -E output of the compiler and filtering it through a perl script called DependenceFilter, which lives in DSystem/tools/develop/bin. This ensures that the include files are actually the ones that are used during compiles.

To support shadow development, dependence information is stored in `.d_`*filename* rather than appending it to the Makefile itself, and DependenceVerify is used to do the processing required.

Private Directory Structures

5.1 Introduction

Developers at Rice use private directory structures to modify the D System source code. This section discusses the merits of using a private directory and describes the upper levels of a private directory structure. Instructions for setting up a private directory structure are delayed until Section 6.

5.2 Why Use a Private Structure?

You don't have to use a private directory structure to modify the D System source code. However, if there are several developers in your group or you're basing several projects on the D System source code, you may find a private structure convenient. For example, using a private structure and the D Developer's Toolkit, you can compile a complete application in your private world. Even if you change just one library, the D makesystem will link in all the other libraries from the installed world to easily create your own executables. That way, you can test the executables resulting from your changes before you overwrite the actual source code, which is sitting safely in the installed world.

In addition, you can create and link all the executables without having all the private directories. If a certain library doesn't exist in your private world, the D makesystem will use the correct libraries from the installed directory structure. Even if you're the only developer modifying the D System, you still might find a private structure convenient. After all, there's a toolkit in place to facilitate your development, and you can test your changes before you integrate them with the installed source code.

5.3 Private Directory Structures

At Rice, each developer's account contains one or more private directory structures, or *private worlds*. A private world, which partially replicates the directory structure of the installed world, contains copies of the source code files that a developer has created or is modifying. Like the installed directory structure, a private directory structure separates the source and object files into two distinct trees. The upper levels of a private directory structure are shown in Figure 4.

In the installed world, DSystem/develop/src contains a directory named RCS. When the D System was installed, all of the source files were checked into empty RCS directories with their current revision numbers. This approach keeps your source files in synch with Rice files, just in case you should ever need to report a bug fix. In your private world, every directory under src/ and obj/ contains a softlink back to its corresponding RCS directory in the installed world.

Although Figure 4 shows only one private directory structure, in practice there may be more than one private structure for each developer. Furthermore, each developer's private structure might contain a set of subdirectories for each project. This concept is illustrated in Figure 5.

DSystemPrivate/

- src/ - your private copies of the D System source code
  - RCS@ - a link to DSystem/develop/src/RCS/ (see Figure 2)
  - execs/ - source for the D System executables
  - include/ - miscellaneous include files
  - libs/ - main source code for the D System
- obj/ - your private copies of the D System compiled library archives
  - SunOS-4.x/ - may be empty or may not exist
  - SunOS-5.x/ - compiled library archives for SunOS 5.x
    - execs/ - executables
    - include/ - miscellaneous include files
    - libs/ - main source code for the D System

Figure 4: The upper levels of a private directory structure.

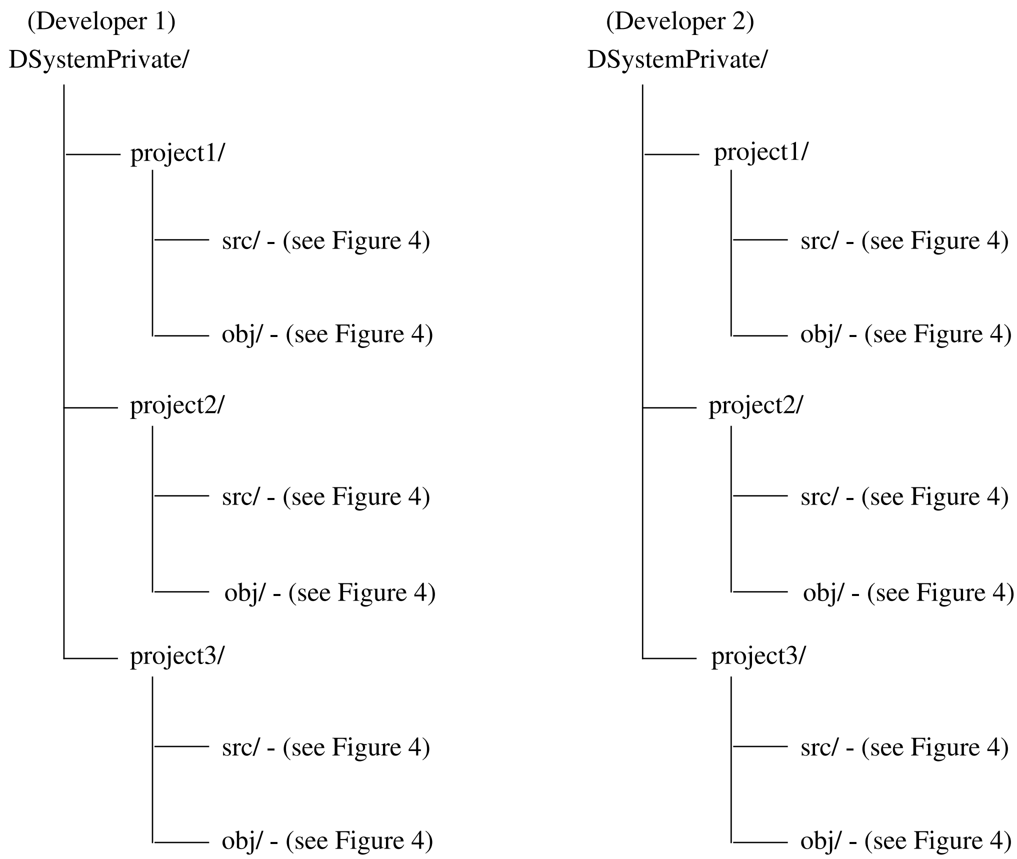

Figure 5: An overview of private directory structures for multiple developers with multiple projects.

Setting Up Your Environment

6.1 Prerequisites

This section provides instructions for setting your environment variables, updating your path, and creating a private world. To use these instructions, you should

- be familiar with the terms *installed world* and *private world*
- know how to modify a shell rc file
- know where the D System was installed

If the terms *installed world* and *private world* are new to you, this would be a good time to look over Section 2.2. If you've never modified a shell rc file, you may want to ask someone to help you interpret these instructions.

Finally, if you're not sure where the D System was installed, you may have to contact the person who performed the installation. If this is not an option, try searching for a directory called DSystem, since this is the default directory where the D System files would have been installed.

6.2 Setting Your Environment Variables

If you plan to modify the D System, you'll need to set the following environment variables within your shell rc file:

- `DSYSTEM`
- `DSYSTEM_OS`
- `DSYSTEM_PVT`

The utilities in the D Developer's Toolkit use these environment variables to control compilation. `DSYSTEM` points to the installed home directory; `DSYSTEM_OS` contains the operating system you want to compile for, based on the directories installed in DSystem/tools/develop/bin/; and `DSYSTEM_PVT` points to your private D System directory.

So, if the D System was installed under /<path>/DSystem and you plan to create your private world under /home/<userid>/DSystemPrivate/, you would edit your shell rc file so that

`DSYSTEM` corresponds to `/<path>/DSystem`
`DSYSTEM_OS` corresponds to `` `/<path>/DSystem/tools/develop/bin/darch` ``
`DSYSTEM_PVT` corresponds to `/home/<userid>/DSystemPrivate`

Instructions for creating a private world are provided in Section 6.4. If you don't plan to use a private world, you would set `DSYSTEM_PVT` to /<path>/DSystem/develop.

A C-Shell Example

If you're running C-Shell, you'll need to modify your .cshrc file. To do this, go to your home directory (cd ~), then use your favorite editor to add the following lines to your .cshrc file:

```
setenv DSYSTEM /<path>/DSystem
setenv DSYSTEM_OS `/<path>/DSystem/tools/develop/bin/darch`
setenv DSYSTEM_PVT /home/<userid>/DSystemPrivate
```

In this example, we have arbitrarily chosen the name DSystemPrivate for your private world. If you don't plan to use a private world, your .cshrc file would contain the line

```
setenv DSYSTEM_PVT /<path>/DSystem/develop
```

6.3 Updating Your Path

Note: The following path updates must follow the definitions of the environment variables `DSYSTEM`, `DSYSTEM_OS`, and `DSYSTEM_PVT` (Section 6.2).

You won't be able to access the various D System tools until they are on your path. To update your path, go to your home directory (`cd ~`), and use your favorite editor to add the following directories to the PATH environment variable in your shell rc file:

- `${DSYSTEM}/bin/${DSYSTEM_OS}`
- `${DSYSTEM}/tools/develop/bin`
- `${DSYSTEM}/tools/develop/bin/${DSYSTEM_OS}/rcs`

${DSYSTEM}/bin/${DSYSTEM_OS} points to the D System end-user tools, like the Fortran D editor and compiler. ${DSYSTEM}/tools/develop/bin points to the various OS-independent scripts, like dmake and makeall; and ${DSYSTEM}/tools/develop/bin/${DSYSTEM_OS}/rcs points to the D System rcs commands, like dco and dci.

Updating Your Man Path

To update your man path, set your `MANPATH` environment variable to include the paths

- `${DSYSTEM}/man`
- `${DSYSTEM}/tools/develop/man`

${DSYSTEM}/man contains the man pages for the D System executables, like the D editor and the D compiler, and ${DSYSTEM}/tools/develop/man contains the man pages for the scripts in the D Developer's Toolkit.

6.4 Creating A Private World

Once you've set your environment variables (Section 6.2) and updated your path (Section 6.3), you're ready to create your private world. Basically, this involves

1. Creating your private root directory
2. Creating your private source directory
3. Creating your private object directory (optional)
4. Setting your private-world environment variables
5. Sourcing your shell rc file
6. Running dmake template

This process is depicted in Figure 6. Detailed instructions follow.

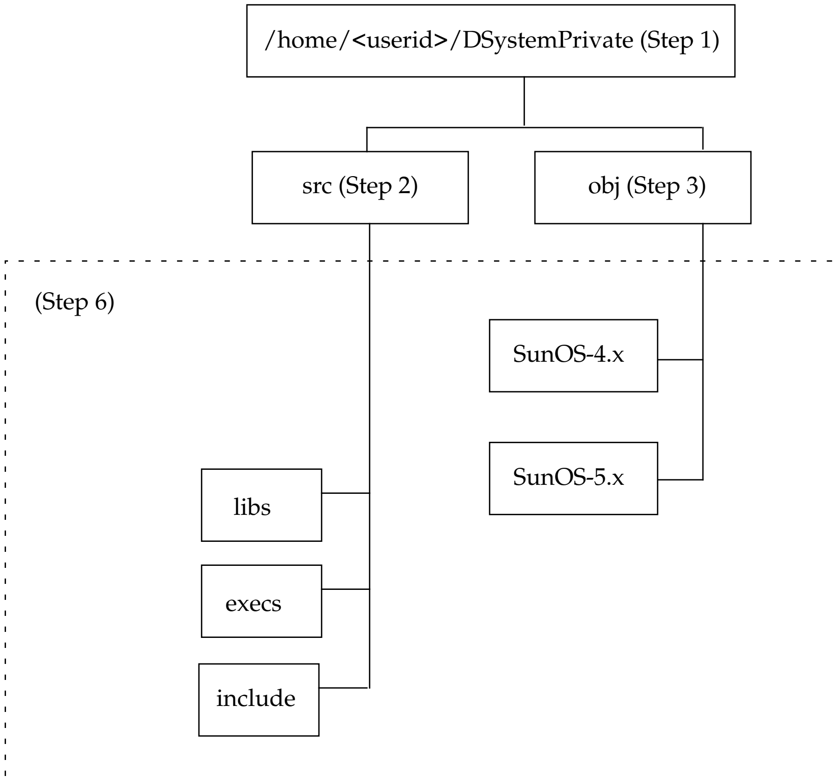

Figure 6: The upper levels of a new private directory structure. Here, the user has chosen to keep the private object directory separate from the private source directory.

Detailed Instructions

Step 1. Within your home directory, create the root directory for your private world (Here, we have arbitrarily chosen the name DSystemPrivate):

```
cd ~
mkdir DSystemPrivate
```

Step 2. Create your private src directory within your private root:

```
cd DSystemPrivate
mkdir src
```

If you have named your private root directory something other than DSystemPrivate, you would cd into that directory and create a src directory there.

Step 3. Create your private obj directory within your private root:

```
mkdir obj
```

If you're only going to be working under one operating system, you don't have to have a separate obj directory; however, we recommend that you create a separate object directory. With a separate object directory, your private source files will be kept separate from your private object files (and hence your private archives).

If you decide to skip this step, all of your private source and object files will reside in the source directory; that is, your private source and object directories will be overlayed.

Step 4. If necessary, go back to your home directory (cd ~) and use your favorite editor to set the environment variable `DSYSTEM_PVT` in your shell rc file. If you're running C-Shell and your private root directory is named DSystemPrivate, your .cshrc file would contain the line

```
setenv DSYSTEM_PVT /home/<userid>/DSystemPrivate
```

Step 5. Source your shell rc file. For example, if you're running C-Shell, source your .cshrc file:

```
source /home/<userid>/.cshrc
```

Step 6. Now you're ready to build your private world. To do this, you will use the dmake utility from within your private source directory:

```
cd DSystemPrivate/src
dmake template
```

First, dmake checks to see if you have a separate private obj directory. If you have a separate private obj directory, dmake automatically creates an empty subdirectory for each OS the D System supports. If you have only a src directory, dmake assumes that you want to overlay your private source and object directories and does not attempt to construct a separate private obj directory.

Note: Directories below each OS directory will not be created until you compile your private-world source code as described in Section 7.

Next, dmake copies the directory structure of the installed-world src directory into your private-world src directory.

Note: The dmake template command does not copy the actual source code files into your private world. To bring the actual source code files into your private world, you will have to use the commands described in Section 7.

Modifying Source Code

7.1 Introduction

Before you start modifying the D System's 1,000+ source files, make sure that you have at least a high-level understanding of

- The general relationship between a private directory structure, the installed directory structure, and the D Developer's Toolkit (Section 2)
- The installed directory structure (Section 3)
- The D makesystem (Section 4)
- Private directory structures (Section 5)
- Environment variables (Section 6)

If you're familiar with all of these concepts and you've set up your private world (Section 6), you're ready to modify the D System source code. To do this, you will use the utilities dco, drcs, ddiff, dmake, and dmakeall, along with some of the other tools listed in Section 7.2, to

- Check out a file
- Lock or unlock a file
- Add a file or directory
- Delete a file or directory
- Compile your changes

For tips on testing your changes, see Section 8. To integrate your changes with the installed code, see Section 9.

Related Man Pages

For more information on the utilities dco, drcs, ddiff, dmake, and dmakeall, see their man pages (${DSYSTEM}/tools/develop/man).

7.2 Development Tools

Developers at Rice use the tools listed below to modify the D System. In Section 7.3, you will use some of the tools to check out and modify a D System file named HashTable.C.

D System Developer's Man Pages

Many of the scripts developed at Rice have accompanying man pages in ${DSYSTEM}/tools/develop/man/. This path was added to your MANPATH when you set up your private world (Section 6.3).

GNU Tools

Developers at Rice have created a set of scripts that run on top of GNU make v.3.70. During the installation, a symbolic link for gmake was added to ${DSYSTEM}/tools/develop/bin/${DSYSTEM_OS}/.

Unfortunately, gcc and g++ v.2.5.8 or earlier do not work with the D System, because they do not handle multiple inheritance correctly. Furthermore, later versions of gcc and g++ may not work with the D System if they do not handle multiple inheritance correctly.

Perl

In addition to the perl scripts that make up the D Developer's Toolkit, developers at Rice have created various perl scripts to make their work easier. Man pages for all of these scripts reside in ${DSYSTEM}/tools/develop/man/. During the installation, a symbolic link for perl v.4.036 was added to ${DSYSTEM}/tools/develop/bin/${DSYSTEM_OS}/.

Purify

At Rice, Purify is used during the linking phase to produce an executable program with runtime checks on memory usage. So, in general, Purify catches and reports on various memory-related errors. During the installation, information about Purify may have been added to a file named Makeinclude.config in the ${DSYSTEM}/develop/src directory. For more information about Purify, contact Pure Software at info@pure.com or phone 408-720-1600.

RCS

Developers at Rice use RCS to move files between the installed world and their private worlds. During the installation, a symbolic link for RCS v.5.6.0.1 may have been added to ${DSYSTEM}/tools/develop/bin/${DSYSTEM_OS}/rcs/. Links in this directory point to the D System rcs commands, like dco and dci.

Sun's C/C++ Compilers

At Rice, developers use the C/C++ compilers from Sun's SC 2.0.1 release for SunOS 4.1.3 and the C/C++ compilers from Sun's SC 3.0 release for Solaris 2.3. All C and C++ source files should go through the C++ compiler under the strictest available checking; currently, however, they do not. Currently, C source files are compiled by the ANSI C compiler. During the installation, information about the compilers you'll be using was added to a file named Makeinclude.config in the ${DSYSTEM}/develop/src directory.

Solaris' Debugger & Source Browser

There are two forms of debuggers under Solaris, dbx and debugger. debugger is a GUI on top of dbx. See the Solaris man pages for more information.

Solaris' Source Browser is good for statically looking at source code. It can find occurrences of variables, functions, etc. and create callgraphs, classgraphs, etc. For more information, see the sbrowser man page or the Answerbook on-line documents, email sunpro-info@sun.com, or phone 1-800-278-6776.

7.3 Modifying Source Code: An Example

In this example, you will use the D Developer's Toolkit to modify the file HashTable.C for a single platform. This procedure is summarized in Figure 7. To make changes for more than one

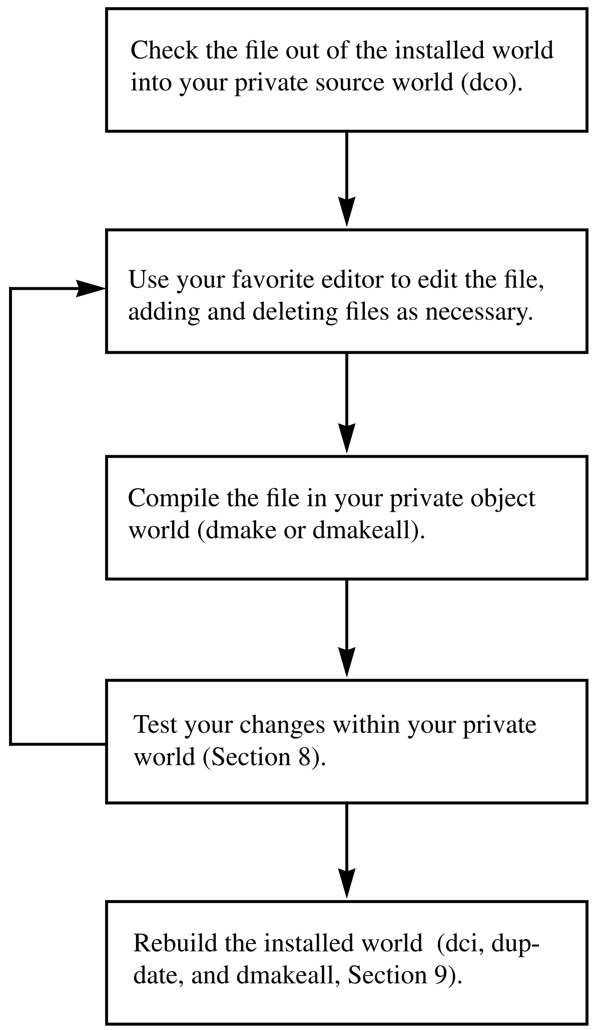

Figure 7: The process used to modify D System source code.

platform, use this procedure to modify the source code and then run a dmakeall instead of a dmake (see "Compiling for More Than One OS," on pg. 24). For a full description of many of the commands used here, including command-line options, see the D System developer's man pages, in ${DSYSTEM}/tools/develop/man/.

Checking Out a File Unlocked

To check out the file HashTable.C, first cd into your private src directory. From there, cd into libs/support/tables, which is the path for HashTable.C relative to ${DSYSTEM}/develop/src (and therefore ${DSYSTEM_PVT}/src). If you just want to view HashTable.C, type

```
cd /home/<userid>/DSystemPrivate/src/libs/support/tables
dco HashTable.C
```

This checks out a read-only, or unlocked, version of HashTable.C into your private world.

Checking Out a File Locked

If you want to be able to modify the file and install the changes, then check the file out locked:

```
cd /home/<userid>/DSystemPrivate/src/libs/support/tables
dco -l HashTable.C
```

This (1) locks the installed version of HashTable.C so that no one else can change it and (2) checks out a read-write version of HashTable.C into your private directory.

Locking & Unlocking a Checked-out a File

Before you can lock or unlock the installed-world version of HashTable.C, you must be in the `${DSYSTEM_PVT}/src` directory that corresponds to the installed-world directory where HashTable.C lives. If you checked out HashTable.C locked but then decided not to modify it, you can unlock it by typing

```
cd /home/<userid>/DSystemPrivate/src/libs/support/tables
drcs -u HashTable.C
```

Likewise, if you checked out HashTable.C unlocked but now want to lock it, type

```
cd /home/<userid>/DSystemPrivate/src/libs/support/tables
drcs -l HashTable.C
```

If another user already has the file locked, drcs will ask you if you want to break the lock. If you answer yes, drcs will send mail to the person who put the lock in place and break the lock.

Editing a File

Once you have checked out HashTable.C locked, you can then use your favorite editor to modify it; for example, most developers at Rice use GNU emacs or vi.

Adding & Deleting Files

As you're modifying the D System source code, if you need to add or delete a file or directory, you'll also need to modify the D makesystem. As explained in Section 4, each directory below ${DSYSTEM}/develop/src contains a makefile named Makefile, which defines the variables

- HERE
- DIRS (optional)
- SRCS (optional)
- ARCHIVE (optional)

HERE contains the present working directory relative to ${DSYSTEM}/develop/src. DIRS, which is optional, contains the subdirectories of the present working directory. SRCS, which is optional, lists the source files in the present working directory that need to be compiled. ARCHIVE, which is also optional, lists the archives where the makesystem will store the compiled source files and object files.

Adding a File

If you add a file to the D System, open the makefile in the directory where you're adding the file, and edit the SRCS variable to contain the new file.

Adding a Directory

To add a directory to the D System, first create the new directory, then copy the makefile from the new directory's parent directory. Next, edit the makefile in the new directory so that HERE contains the present working directory relative to ${DSYSTEM}/develop/src, DIRS contains the subdirectories of the present working directory, SRCS lists the source files in the present working directory, and ARCHIVE lists the archives where the makesystem will store the compiled source files and object files.

Deleting a File

If you delete a file from the D System, open the makefile in the directory where you're deleting the file, and remove the file from the SRCS variable.

Deleting a Directory

To delete a directory from the D System, update the DIRS variable in its parent directory's makefile, then remove the directory and all of its contents.

Comparing Your Changes With the Installed Version

Once you've made your changes, you can diff the modified file with the installed file by typing

```
ddiff HashTable.C
```

A list of differences will be sent to the screen.

Compiling for One OS

Before you install a modified file to the installed world, you will probably want to compile and test your changes within your private world. To do this, you will use the dmake command with or without command line options. You use dmake in your private obj directory to recompile your source code and rebuild your archives for a single operating system. To recompile your source code and rebuild your archives for more than one operating system, use the dmakeall command described in "Compiling for More Than One OS," on pg. 24.

Note: Both dmake and dmakeall create multiple archives throughout your private directory structure. For more information on these archives in their individual directories, take the D System Hypertext Directory Tour (http://www.cs.rice.edu/fortran-tools/DSystem/v0.6/).

Getting Started

Before you invoke dmake, you must first cd into your private obj directory. If you have separate private src and obj directories, you will also need to cd into the appropriate OS directory.

```
cd /home/<userid>/DSystemPrivate/obj
cd SunOS-4.x or cd SunOS-5.x
```

Performing a Dry Run

If you want to see all the commands that dmake will execute without actually executing them, type

```
dmake -n
```

The -n option, which is simply the GNU make -n option, performs a dry run of dmake.

Rebuilding/Creating a Directory Structure

The first time you compile a file for a directory, that directory may not exist. This is okay, because dmake can create missing private subdirectories by looking at the installed directory structure. To rebuild everything in your private obj world, go to your private obj directory and type

```
dmake
```

and dmake will rebuild everything from your private obj down. If you want to build or rebuild the part of your private obj world that should contain your modified HashTable.C file, you can either go to the desired private obj subdirectory, if it exists, and run dmake, or you can type

```
dmake DIRS="libs/support/tables"
```

This creates the libs/support/tables directory tree underneath `${DSYSTEM_PVT}/obj/${DSYSTEM_OS}/` and then compiles HashTable.C.

Compiling for Smaller Executables

By default, dmake builds -g archives, or archives with debugging information. However, the command

```
dmake STRIPPED=yes
```

forces dmake to build an archive without debugging information. This option, which ultimately results in smaller executables, is useful if you want to link an archive without debugging information into a debuggable executable. For more information on archive names, see Section 3.

Using Purify or Source Browser

Furthermore, dmake supports Purify and Source Browser at the command line:

```
dmake PURIFY=yes
dmake SBROWSER=yes
```

For more information on Purify and Source Browser, see Section 7.2.

Compiling for More Than One OS

If you want to recompile your source code and rebuild your archives for more than one operating system, you must use the dmakeall command. Basically, dmakeall runs dmake for multiple operating systems and produces a log file. To recompile your source code and rebuild your archives for a single operating system, use the dmake command described in "Compiling for One OS," on pg. 22.

Note: Both dmake and dmakeall create multiple archives throughout your private directory structure. For more information on these archives in their individual directories, take the D System Hypertext Directory Tour (http://www.cs.rice.edu/fortran-tools/DSystem/v0.6/).

Developers at Rice use the following procedure to update the installed archives and executables for multiple platforms.

0. Edit `${DSYSTEM}/tools/develop/bin/dmakeall`
1. cd into your private obj directory,
2. Run `dmakeall [options]`, then
3. grep `${DSYSTEM_OS}/`*`userid`*`.log` for "Error" and "warning".

These steps are discussed below.

Step 0. Before you run dmakeall for the first time, edit the `servers` section of the file ${DSYSTEM}/tools/develop/bin/dmakeall to contain the names of the machines you wish to compile on.

Step 1. You must run dmakeall within your private obj directory.

Step 2. Running dmakeall without any options performs an unrestricted dmakeall; that is, it compiles for all OS subdirectories. To run an OS-specific dmakeall, use one of the commands

```
dmakeall -SunOS_5x
dmakeall -SunOS_4x
```

As was the case with dmake, you can dmakeall everything from the current directory down, or you can specify a node with the `DIRS` option, as in

```
dmakeall DIRS="libs/fortD execs"
```

First, dmakeall will ask you to delete any `${DSYSTEM_OS}/`*`userid`*`.log` files. After these log files have been deleted, dmakeall rebuilds the installed archives and executables from the current directory down and creates a log file. At Rice, a dmakeall compiling for Solaris running on a Sparc 10 takes about two hours.

Step 3. dmakeall uses your userid, along with `DSYSTEM` and `DSYSTEM_OS`, to create a logfile named *`userid`*`.log`. This logfile is specific to both the user and the operating system, and it is stored in `${DSYSTEM_PVT}/obj/${DSYSTEM_OS}/`.

Testing Your Changes

Instructions for testing your changes are provided in Section 8.

Testing Your Changes

Note: This section is under construction. Please mail your questions or comments about testing and troubleshooting to dsystem-info@cs.rice.edu.

8.1 Introduction

In Section 7, you checked source code out of the installed world and modified it. Now you'll test and troubleshoot your changes within your private world by

- Writing a Test Harness
- Troubleshooting Undefined References
- Generating .cpp Output
- Generating .cfront Files
- Generating .includes Files
- Looking at Dependence Files
- Troubleshooting uninitialized Variables

Related Man Pages

As you're testing your changes, you may find the tools dmake, includes, wif, wincls ("winkles"), Purify, dbx, and debugger helpful. For more information on dmake, includes, wif, and wincls, see their man pages (${DSYSTEM}/tools/develop/man); for more information on Purify, dbx, and debugger, see the Purify and Solaris man pages.

8.2 Testing A Subsystem

Writing a Test Harness

Existing Archive/GUI Application

If the code you want to test is being compiled into an existing archive and you're developing a graphic interface application, use the following instructions to create a test harness.

To produce an executable for a test harness, first create a subdirectory for your test harness in your private execs directory (${DSYSTEM_PVT}/src/execs/). Next copy the makefile ${DSYSTEM}/develop/src/execs/fortDEditor/Makefile into your test harness directory. Finally edit Makefile so that the environment variables HERE, SRCS, and ARCHIVE reflect the current directory, the source files to be compiled, and the name of the archive where you want to store the compiled source files.

Non-existent Archive/Command Line Application

If the code you want to test is being compiled into an existing archive and you're developing a command line application, use the following instructions to create a test harness.

To produce an executable for a test harness, first create a subdirectory for your test harness in your private execs directory (${DSYSTEM_PVT}/src/execs/). Next copy the makefile ${DSYSTEM}/develop/src/execs/fortDCompiler/Makefile into your test harness directory. Finally edit Makefile so that the environment variables HERE, SRCS, and ARCHIVE reflect the current directory, the source files to be compiled, and the name of the archive where you want to store the compiled source files.

Existing Archive/GUI Application

If the code you want to test is not being compiled into an existing archive and you're developing a graphic interface application, use the following instructions to create a test harness.

Create a Makeinclude.override file in ${DSYSTEM_PVT}/src/ that redefines the environment variable LIBS to include the name of the library for your new subsystem. The name of the library you’re adding must appear in the correct order, so that it respects the one-pass link order. (For more information on the environment variable LIBS, see Makeinclude.components in ${DSYSTEM}/develop/src/.)

Non-existent Archive/Command Line Application

If the code you want to test is not being compiled into an existing archive and you're developing a command line application, use the following instructions to create a test harness.

Create a Makeinclude.override file in ${DSYSTEM_PVT}/src/ that redefines the environment variable NLIBS to include the name of the library for your new subsystem. The name of the library you’re adding must appear in the correct order, so that it respects the one-pass link order. (For more information on the environment variable NLIBS, see Makeinclude.components in ${DSYSTEM}/develop/src/.)

Executable Archives (incomplete)

- ${DSYSTEM}/develop/src/execs/fortDEditor/dedX-${DSYSTEM_OS}-g.a

8.3 Troubleshooting Tips: Linking

Troubleshooting Undefined References

Purify invokes the Sun linker and reports the name of the object file where the first undefined reference occurred. When you encounter a reference to undefined symbols, you'll either know where the problem is or you'll use the Unix utilities find, nm, and grep to locate it. (For more information on these utilities, see their man pages.)

For example, say that Purify reports an undefined reference for the function foo in bar.o.
If you think you know which library bar.o came from, look at that library and look in the library in the LIBS or NLIBS path (in the Makeincludes.component file) to note the library order you have; the definition should be after the first use.

If you don't know where the undefined function should have been defined, use the Unix utilities find, nm, and grep:

1. Use find to list all archives (*.a) in the your private obj directory.
2. Run nm with the prepend option on the list of archives created in Step 1.
3. Grep for the undefined function or variable.

4. Run nm again
5. Grep for definitions.
6. Check the link order. The definitions should occur after the first link order.

Generating .cpp Output

To generate preprocessor output for <filename>.C, type

```
dmake <filename>.cpp
```

You can then use the .cpp file to locate include file problems.

Generating .cfront Files

The Sun C++ compiler v.3.0 translates C++ source to C source. To look at the C output produced by the Sun C++ compiler v.3.0 or earlier, type

```
dmake <filename>.cfront
```

This produces a .cfront file, which you can use to track down a problem.

Generating .includes Files

If you need to track down how a particular include file is used, you can either generate the .cpp output and go looking through it, or you can type

```
dmake <filename>.includes
```

This tells you how include files are brought into a particular file during compilation.

Looking at Dependence Files (TBD)

Instructions for viewing .d_* files go here.

8.4 Troubleshooting Tips: Runtime

Troubleshooting uninitialized Variables

Purify Breakpoints & Watchpoints

Developers at Rice use Purify breakpoints and watchpoints to troubleshoot runtime errors. If you want Purify to break at a particular error and ignore other errors, you can create a .purify file that contains "suppression directives." So, for example, if you want Purify to break at a particular error and you don't care about uninitialized memory reads in subsystems that are outside your control, like inside the X-Windows system, your .purify file would contain a line of the form

suppress umr...:<X library function>

Then, as Purify is stepping through your code, if it sees an uninitialized memory read in any function that's called from the X library function that you told it to suppress, Purify will ignore it.

Purify watchpoints tell you when a particular piece of memory has been clobbered, and only slows down the process by a factor of about four, rather than the typical dbX watchpoints, which are much slower.

For more information on Purify breakpoints and watchpoints, see Purify documentation.

Rebuilding the Installed World

9.1 Introduction

In Section 7 and Section 8, you checked source code out of the installed world, modified it, and then tested it. Now you will

- Check your changes into the installed world
- Update the actual source code in the installed world
- Rebuild the installed world

9.2 Rebuilding the Installed World: An Example

In Section 7 and Section 8, you checked a file named HashTable.C into your private world and then modified it and tested your changes. Now you will use the scripts dci, dupdate, dmakeall, and dmake to rebuild the installed world. dci and dupdate add your changes to the installed-world source code. dmake, which is primarily associated with private worlds, rebuilds private archives and executables for a single platform. dmakeall, which is primarily associated with the installed world, rebuilds the installed archives and executables for multiple platforms.

Related Man Pages

For more information on the scripts dci, dupdate, dmakeall, and dmake, see their man pages (${DSYSTEM}/tools/develop/man).

Rebuilding the Installed World

After you've used dci and dupdate to update the installed-world source code, you're ready to rebuild the installed-world archives and executables. To do this, you will use the dmakeall utility. dmakeall, which calls dmake, is an OS-independent script that Rice developed on top of GNU make. Instructions for using dmakeall are summarized below. To learn more about dmakeall, see its man page (${DSYSTEM}/tools/develop/man).

At Rice, developers use the following procedure to rebuild the installed world:

0. If necessary, edit ${DSYSTEM}/tools/develop/bin/dmakeall.
1. Announce your intentions to your group.
2. If necessary, rlogin to the installed world as the installed-world userid.
3. cd into ${DSYSTEM}/develop.
4. source `SourceMe`.
5. cd into ${DSYSTEM}/develop/obj.
6. Use dco to check out `dmakeall.reasons` locked and edit it.
7. Return to your private world, then use dci and dupdate to install your source code.
8. Return to the installed world, then type `dmakeall [options]`
9. grep ${DSYSTEM_OS}/userid.log for the words "Error" and "warning".
10. Check `dmakeall.reasons` back in.

These steps are discussed below.

Step 0. If necessary, edit ${DSYSTEM}/tools/develop/bin/dmakeall. Before you run dmakeall for the first time, edit the `servers` section of the file ${DSYSTEM}/tools/develop/bin/dmakeall to contain the names of the machines you wish to compile on.

Step 1. Announce your intentions to your group. To minimize confusion within your group, we encourage you to make some kind of announcement every time you plan to run a dmakeall. In fact, it is best if only one person in your group has the authority to execute the dmakeall command. If you're the only one modifying the D System source code, this step is not necessary.

Steps 2 and 3. If necessary, rlogin to the installed world as the installed-world userid,then cd into ${DSYSTEM}/develop. If you're using a private directory structure, you'll need to log in to the installed world as the installed-world userid and move to the ${DSYSTEM}/develop directory. Even if you're not using a private structure, you'll still need to move to ${DSYSTEM}/develop before you run a dmakeall.

Step 4. source `SourceMe`. Before you run dmakeall, you must first set your environment variables by sourcing a file named SourceMe. Otherwise, your dmakeall will operate on the wrong dmake.

Step 5. cd into ${DSYSTEM}/develop/obj. You must run dmakeall within the ${DSYSTEM}/develop/obj directory.

Step 6. Check out `dmakeall.reasons` locked and edit it. The file dmakeall.reasons, which lives in ${DSYSTEM}/develop/obj, documents the changes made to the D System and provides a locking mechanism to keep two or more people from running dmakeall at the same time. After you've checked this file out locked, edit it to include your changes.

Caution: If `dmakeall.reasons` has been checked out, someone else is doing a dmakeall; *ergo*, don't do another dmakeall!

Step 7. Use dci and dupdate to install your source code. If you're using a private directory structure, you'll probably want to use the private scripts dci and dupdate to check in and update your revised source code. Otherwise, you can use the revision control tools of your choice. If you're not using revision control, you would skip this step.

Checking In Modified Files

At this point in the development process, you've checked out HashTable.C (Section 7), modified it, and tested it (Section 8) – all from within your private world. When you're sure that your changes are correct, check the modified HashTable.C file back into the installed world by typing

```
dci HashTable.C
```

from within your private world. This changes the HashTable.C,v file in the installed world, but does not alter the installed-world file. Next, you will use the dupdate command to change the installed-world file.

Updating Installed-world Files

Once you've used dci to check your changes into the installed world, update HashTable.C in the installed world by typing

```
dupdate HashTable.C
```

from within your private world.

Caution: If someone is running a dmakeall in the installed world, do not run dupdate. Since dmakeall depends on any dupdates that were run before dmakeall was started, strange side-effects can occur from starting a dupdate while dmakeall is running.

Step 8. Type `dmakeall [options].` First dmakeall will ask you to delete any ${DSYSTEM_OS}/userid.log files. After these log files have been deleted, dmakeall rebuilds the installed archives and executables from the current directory down. To do this, dmakeall invokes the script dmake. Selected ommand line options for dmakeall are provided in the next section.

You can dmakeall everything from the current directory down, or you can specify a node with the command

```
dmakeall DIRS="<libs/fort execs>"
```

as was the case with dmake.

Running dmakeall without any options performs an "unrestricted "dmakeall; that is, it compiles for all OS subdirectories.

To perform an OS-specific dmakeall, type

```
dmakeall -SunOS_5x
```

or

```
dmakeall -SunOS_4x
```

Step 9. grep ${DSYSTEM_OS}/userid.log for the words "Error" and "warning". dmakeall uses your userid, along with the environment variables `DSYSTEM` and `DSYSTEM_OS`, to create a logfile named `userid-g.log`. This logfile is specific to both the user and the operating system, and it is stored in `D_HOME/obj/$(D_ARCH)/`.

Step 10. Check `dmakeall.reasons` back in.

Index

E

F