Data General

Customer Documentation

Porting Applications to the DG/UX™ System

AViiON™
PRODUCT LINE

Porting Applications to the DG/UX™ System

093-701047-00

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-701047

Printed in the United States of America
Revision 00, April 1989

NOTICE

Porting Applications to the DG/UX™ System

093-701047

Revision History:
Original Release – April 1989

Effective with:
DG/UX Release 4.10

Preface

This manual describes how to port UNIX® application programs to the DG/UX™ System.

This manual is directed primarily to experienced C programmers. This means that we assume the reader is fluent in C, but may not be skilled in some of the more esoteric aspects of the language. The manual is for programmers who are porting their software for the first time, as well as for programmer's who are experienced in porting issues.

If you need introductory information about the DG/UX system, please read *Using the DG/UX™ System*. For a list of DG/UX manuals, see "Related Manuals" at the end of this book.

How This Manual Is Organized

This manual is organized as follows:

Chapters 1–2	An overview of the porting process and porting issues. Programmers not experienced in porting issues should read these chapters before beginning to port.
Chapters 3–11	Specific technical information about the major porting issues. These chapters can be read individually.
Appendixes	Examples and tips about the porting process.

Notation Conventions

We use the following notation conventions in syntax:

Element	**Meaning**
Bold string	A literal to be typed just as it appears
Italic string	A place holder representing a literal or other value that you supply
[ ]	Delimiters for an optional argument (not to be confused with the meaning of brackets used in examples)
...	Optional repetition of the preceding argument

We use the following notation conventions in examples:

Element	Meaning
`Bold string`	A literal to be typed just as it appears
`Roman string`	A system response that you will see on your screen
[ ]	Literal brackets to be typed exactly as they appear
<NL>	The New Line or Return key to be pressed on your keyboard

Contacting Data General

If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index.

If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your local Data General sales representative.

Service assistance on Data General software or hardware is available via telephone in the United States and Canada. Ask your Data General representative for the number. Outside North America contact your local Data General office.

End of Preface

Contents

Chapter 6 — Using Software Development Tools

Chapter 7 — Compiling and Linking Programs

Chapter 8 — Using Libraries And System Calls

Chapter 9 — Understanding Hardware Architectural Issues

Chapter 10 — Porting Graphical Applications

Chapter 11 — Building and Running BCS Applications

Appendix A — DG/UX System Calls and Commands

Appendix B — Comparison of C Compilers

Tables

Table

Figures

Figure

Chapter 1
Introduction: Porting and UNIX® Standards

Data General's DG/UX™ product family provides a robust UNIX® environment that complies with many of the existing UNIX standards; nonetheless, your source code may require some changes to run optimally in this environment. This chapter discusses the applicable standards and discusses why porting is necessary. The chapter identifies the major standards activities and organizations and describes how Data General's DG/UX product family relates these standards.

The chapter contains the following sections:

- Overview of Porting and Standards
- Standards Organizations
- Binary Standards and the BCS
- Operating System Standards
- User Interface Standards
- Language Standards
- Communications Standards
- Other Standards
- Summary

Overview of Porting and Standards

To "port" software means to transport it to a new operating environment; that is, to get software currently running on one machine to run on another. This manual outlines the areas of your code and your assumptions about the operating environment that you should examine for compatibility. The conventions described in this manual work on most other UNIX systems.

The DG/UX system fully supports the Binary Compatibility Standard (BCS) for the Motorola 88000 architecture. That means that any program you build within a BCS environment will run on Data General's DG/UX system. Additionally, the DG/UX system provides several extensions beyond the basic BCS functionality. Refer to Chapter 11, "Building and Running BCS Applications," for more information on BCS compatibility.

The UNIX® operating system forms the basis of a highly portable software environment. Data General's DG/UX operating system complies with many of the official and de facto standards for UNIX systems.

When you say UNIX-compatible, you must say what compatibility standard you are using. There are several UNIX compatibility standards, and programs built toward one standard may not run on systems built toward another. For example, one standard may not include all the commands or systems calls specified in the other standard.

The existence of multiple standards complicates the goal of portability. Multiple standards exist because the UNIX system has followed an evolutionary path rather than a planned development cycle. The power of the UNIX operating system stems from its flexibility: the UNIX system was designed to encourage enhancements. Programmers modified the operating system to run on various hardware platforms. Users and vendors enhanced the operating system to meet the needs of their environment, adding features and interfaces as needed.

A major goal of standardizing the UNIX system is to achieve application portability. There have been several approaches to standardizing, as represented by the types of organizations that define and collect UNIX-related standards. Adherence to one or more standards achieves varying levels of functionality. As it has traditionally been implemented, UNIX is standard only in concept; each implementation includes differences that require application code to be customized before it will run on a particular UNIX platform.

Standards Organizations

This section discusses 88Open, OSF, ANSI, IEEE, NIST, and FIPS.

88Open Consortium

The 88Open Consortium is a cooperative group of vendors who are building systems based on Motorola's 88000 chip set, and who are working together to create a compatible family of UNIX systems. Although the hardware and software offered by each vendor may differ, application software that conforms to the BCS standard should run on all platforms. Data General is a member of the 88Open Consortium.

OSF

The Open Systems Foundation is continuing to define a standard operating environment, including the operating system and user interfaces. Data General is a member of this consortium.

ANSI

The American National Standards Institute defines many kinds of standards. ANSI committees that are relevant to the UNIX environment include language standards committees, and communication standards committees.

IEEE

IEEE is a trademark of the Institute of Electrical and Electronic Engineers, Inc. IEEE is defining the standard for Portable Operating System for Computer Environments (POSIX).

NIST and FIPS

The National Institute of Standards and Technology (formerly called National Bureau of Standards) is producing a Federal Information Processing Standard (FIPS) based on IEEE 1003.1. Work is currently underway to bring the FIPS specification and the adopted POSIX specification into agreement.

Binary Standards and the BCS

Conceptually, there are three levels of application portability standards: source, object, and binary (executables). A source level standard is the most flexible of all standards, but it requires that the application code be recompiled and linked for all target systems. An object level standard would allow software vendors to ship their software in compiled, but unlinked form. This would allow the system administrator or end-user to link the code with the appropriate libraries for the target environment. No object level standards currently exist; however, Data General is closely tracking the definition being proposed by the 88Open Consortium. A binary level standard presents a "load and go" environment to applications in which the binary program can be copied from the distribution medium and executed without compiling or linking. Data General adheres to the BCS that has been defined by the 88Open Consortium.

The Binary Compatibility Standard (BCS) that has been defined for Motorola's 88000 architecture enables programmers to create software that can run on all BCS-compliant systems without recompiling and relinking the software. To be portable, the code must be developed on a system that follows the BCS specification, and must not use any extensions provided by that environment. The DG/UX operating system is a BCS-compliant operating system.

The BCS definition includes provisions for referencing non-standard functions and system calls from within a conforming application. To achieve this functionality, a program needs to test its operating environment, and provide an alternate (non-standard) code path if (and only if) that option is available in the host environment. The BCS standard does not currently define the networking or the graphics environment; therefore, applications that require these extensions cannot fully conform to the BCS standard. Compliant programs can access system-specific extensions through the "syslocal" interface. Any application that is BCS compliant can run on the DG/UX system.

The BCS coexists with the Application Binary Interface (ABI) defined for the System V environment. AT&T has endorsed the 88K BCS as a compatible implementation of the ABI.

Operating System Standards

Operating system standards define the features of the development and runtime environment. There are many UNIX operating system "standards" that are in use today: some of these standards conflict, but in many cases they can coexist. Data General has always been committed to standards, and the DG/UX operating system follows the following source-level operating system specifications as closely as possible.

System V as defined by the System V Interface Definition (SVID) and validated with the System V Verification Suite SVVS. The DG/UX system is based on AT&T's release V.3.

BSD Berkeley Software Distribution.

POSIX the Portable Operating System Interface definition produced by IEEE 1003.1

FIPS the Federal Information Processing System specification of POSIX.

OSF the Open Software Foundation does not currently define an operating system standard; however DG is tracking the efforts of this organization.

System V, SVID, and SVVS

AT&T created and licenses this version of UNIX System V. There have been several releases of the System V version of UNIX: these are generally referred to as System V.1, System V.2, System V.3, and System V.4. This version of the software is used as the basis for many UNIX derivatives, including the DG/UX, Xenix™, and AIX™ systems. AT&T has also created the System V Interface Definition (SVID) and System V Verification Suite (SVVS). Before it can be called SVID-compliant, a System V UNIX derivative must pass the SVVS. The suite contains several sections; not all are required for compliance. The DG/UX system passes the verification suite.

Berkeley Software Distribution (BSD)

The BSD version of UNIX offers different functionality than the AT&T System V version, although BSD was originally derived from AT&T System III.

The current version of the DG/UX System provides most of the system calls, commands and features defined in the BSD 4.2 UNIX system, including the Network File System (NFS™), networking and sockets, r-commands, job control, and the C shell (**csh**).

The SunOS™ and Ultrix™ operating systems are based on the BSD UNIX system.

POSIX

The standard for Portable Operating System for Computer Environments (POSIX) is being defined by IEEE committees. In some cases, this standard conflicts with the SVVS. Several related IEEE projects each describe a component of POSIX: 1003.1 is an interface specification for a portable operating system based on the UNIX operating system; 1003.2 describes the command shell and tools; 1003.3 outlines verification test procedures for a POSIX operating system; 1003.4 describes realtime extensions for such a system; 1003.5 describes an Ada Binding for POSIX; 1003.6 is outlining security issues for POSIX. POSIX is no longer a trademark.

FIPS

The National Institute of Standards and Technology (NIST) is producing a Federal Information Processing Standard (FIPS) based on IEEE 1003.1 (POSIX). Although FIPS POSIX and the final IEEE POSIX have some differences, work is currently underway to bring the standards into agreement.

OSF

OSF is incorporated as a non-profit research and development organization that will define specifications and promote an open, portable application environment. The foundation will provide a system that is based on IBM's AIX core technology, and includes features to support current System V- and Berkeley-based applications. Specifications supported by OSF will be publicly available, and a set of verification tests for all appropriate facilities will be identified or created.

User Interface Standards

This section discusses the X/Open organization and the X Window System™.

- X/Open
  X/Open was founded in 1984 as an international non-profit organization whose goal is to facilitate the consolidation of standards into a single comprehensive and coherent set. The consortium adopts existing official or de facto standards and collects them to define the Common Applications Environment (CAE). The CAE currently addresses operating system services, programming languages, and data management. Data General is tracking these standards.

- X Windows
  The X Windows graphics standard is defined by the distribution tape of X software from MIT. The DG/UX system provides a full implementation of the X Window environment, and provides many of the clients. Data General is currently tracking user interface standards, and expects to support the MOTIF standard in a future release.

Language Standards

This section discusses the C language and other languages.

- C Language
  Almost all UNIX systems include the C programming language, since most implementations of the OS are written in C. The three primary C language standards are the definition by Kernighan and Ritchie in their book, *The C Programming Language*, the evolving ANSI standard for the C Language (ANSI Technical Committee X3J11) and the de facto pcc standard. These standards define various compiler defaults, options, and features. Data General supports several C compilers that support these standards and meet a wide variety of needs. Separate sections of this manual describe the classic portability issues of the C language, and compare several of the C compilers available on the DG/UX system.

- Other Languages
  Data General supports other languages, including Green Hills FORTRAN and Pascal. The LPI languages, including COBOL, BASIC, Pascal, FORTRAN, PL/I, and the CodeWatch debugger, will be available in the future. These languages and their standards are discussed in separate documentation. Recognized standards include ANSI FORTRAN-77, ISO Pascal, ANSI COBOL-74, ANSI COBOL-80.

Communication Standards

This section discusses ONC/NFS, RFS, TCP/IP, and IBM communications.

- ONC/NFS - Open Network Computing/Network File System
  The DG/UX system implements the NFS file sharing system licensed by SUN. This product allows different operating systems on a variety of machines in a single network to share file systems by remote access, promoting an efficient distributed environment. The current release includes support for RPC, XDR, netdisk and services. Data General systems can participate fully in an inter-vendor NFS network.

- RFS
  RFS is the AT&T implementation of file sharing. This standard is less widely used than NFS and cannot inter-operate with NFS. Data General does not support RFS on the DG/UX system.

- TCP/IP
  The TCP/IP (Transmission Control Protocol / Internet Protocol) standards situation is far from straightforward. The set of protocols to which TCP/IP refers were originally implemented under Department of Defense funding to standardize the way resources are shared across a network.

  For a complete description of Data General's implementation of TCP/IP, see the TCP/IP manuals listed under "Related Manuals" after this manual's index. Data General's version of TCP/IP is derived from the UC Berkeley implementation, with modifications to support the Military Standard.

  Since the original standards were actually implemented by researchers on the ARPAnet, one of many subnets of the huge Internet network, these protocols are widely known as the Internet protocols. Today there are many different implementations of the TCP/IP protocol set, with the following set of specifications forming the core for development.

  - Official Internet Protocol Documents: Published by SRI International, this three-volume set contains collections of documents outlining implementation specifications for the TCP/IP set of protocols. The documents are revised often. Each specification is referred to as an RFC (Request for Comments).

  - Military Standard: Also referred to as the TCP/IP Mil_Spec, this set of 5 DoD documents are a more formalized rewrite of the Official Internet Protocol Documents. There are some inconsistencies between these documents and the SRI publication.

  - UC Berkeley TCP/IP: Under DARPA (Defense Advanced Research Projects Agency) funding, Berkeley adapted the TCP/IP networking software to run on the UNIX system, within the framework of the Berkeley socket systems calls. The networking

software was released with the BSD 4.2 (Berkeley Software Distribution) version of UNIX for use in local area networks. This implementation has now become a de facto standard and is in widespread use.

- Streams TCP/IP: Several companies have recently modified TCP/IP to use the streams I/O interfaces provided by AT&T's UNIX System V.3.

- IBM Communications
  SNA is a de facto communication standard in commercial environments. The DG/UX system supports SNA communications with the DG/UX SNA/3270 package. Refer to separate documentation for a full description of Data General's implementation of SNA.

Other Standards

Following are some other computer standards:

- Object file format
  The object file format used by the DG/UX system is defined by the BCS model. This format extends the standard COFF (the Common Object File Format) definition.

- International Support
  The international character set requires 8-bit support from both the operating system and from application software. The DG/UX system preserves 8-bit support in the kernel and most commands, consistent with AT&T V.3.

- ABI
  The Application Binary Interface, as defined by AT&T, recognizes the BCS as a compatible implementation.

- SAA
  IBM's Systems Application Architecture (SAA) provides users with a Common Programmer Interface (CPI) layer for required services. This concept allows functions to be performed similarly in a number of specifically targeted software/hardware environments. IBM has not currently targeted AIX, its implementation of UNIX, as an SAA platform.

Summary

Standards simplify the porting process, but do not solve all porting issues. Programmers can help ensure the portability of software by following recognized coding standards and practices. Programmers cannot control the compatibility of operating systems, and instead should design code that isolates any operating system or machine dependencies as much as possible.

Standards-based computer platforms promote compatibility, which eases the task of moving software between systems. Many programs are compatible at the source code level. That means that by compiling and linking the software on the target machine, you can create executable code on the target machine that is equivalent (but not identical) to the executable code on the source machine. This process is generally called "porting." The complexity of the task varies with the complexity of the code and with the compatibility of the source and target systems. This manual outlines the major components of porting, at both an overview and a technical reference level.

Application software that conforms to the BCS standard fully can be run on the DG/UX system with no changes, although software vendors may want to use the BCS extension features to allow the software to take advantage of DG features. Application software that conforms fully to operating system and language standards can generally be ported to the DG/UX system with a simple recompilation and relink process.

End of Chapter

Chapter 2
Getting Started: Outlining the Porting Process

Porting issues generally divide into three categories: operating system and user interface issues, compiler issues, and architecture issues. Moving software to a new machine often requires a careful review of both the code and the environment. This chapter outlines the porting process. Chapters 3 through 11 of this manual looks at the specific issues of porting in detail.

A Porting Checklist

The following checklist of activities will help ensure that your porting process proceeds smoothly. Vendors who have not ported their software before will want to review each step in this list carefully. Those vendors who have ported their software to many systems may want to refer directly to Chapters 3-11 of this manual.

1) Load files and verify the file format. (See Chapter 3 if you have any issues with this step.)

2) Determine the language that you will use. The DG/UX system supports a broad range of compilers.

3) Review the environment requirements. The DG/UX system supports both the System V and the BSD environments. Some commands may have different implementations in each of these environments. Refer to the man pages for details on each command. Your shell scripts should be written to specify the shell. The DG/UX system supports a full range of terminals, and uses both the termcap and **curses/terminfo** interfaces. Refer to Chapter 5 of this manual for more details.

4) Review the code.
   Look for dependencies on header files, redefinition of system functions, or dependencies on machine architecture. If your code makes assumptions about the filing system or directory format, or tries to access inodes directly, you will want to review Chapters 5, 8, and 9 in this manual. If your code is derived from a BSD system, refer to Chapter 8 to determine if the system files and libraries are in the "standard" location. If your code uses **malloc**(3C) routines, review Chapter 9 in this manual.

5) Run the **lint** utility program on the files. Use the **make** script, if it contains a **lint** entry point. Chapter 6 highlights the classic porting issues and compatibility problems that are flagged by the **lint** utility. Missing definitions may be related to how your program references libraries.

6) Try to compile and run your program(s). This step should go smoothly if all **lint** errors have been resolved, and the correct libraries are accessible. Refer to Chapters 7 & 8 for discussions of library and compiler issues.

7) Use a debugger to help isolate and resolve problems. The system supports the sdb debugger. Refer to Chapter 6 for an overview of debugger options.

Understanding User Interface Issues

The DG/UX system supports both the System V and the BSD environments. That means that you have your choice of shells when running on this system. The different shells may cause commands and user interaction to behave slightly differently than expected. You should ensure that all shell scripts indicate the shell in which they should run. Refer to Chapter 4 for a discussion of the general environment options.

If you have dependencies on terminal features, refer to Chapter 5.

The program's **make** file will typically define the development environment needs. The AT&T **make** utility provides a way to define dependencies and relationships between programs and system utilities. Specifically, make files should define the compiler arguments and options, and the libraries that are linked into your program. The DG/UX **make** utility conforms to the standard make and dependency rules. Chapter 4 of this document provides additional suggestions on using the make utility in the porting process. The DG/UX tools manual provides a full description of all **make** options.

Understanding Compiler Issues

Most coding issues can be resolved by rigid adherence to the C standards as described by Kernighan and Ritchie, and by the emerging ANSI standards. Most potential problems can be identified and resolved easily; some of the more common issues are noted below.

- Differences between ANSI features and pcc features can generally be resolved by selecting the appropriate compiler switches.

- Structure alignment requirements may introduce padding, which may result in unintended data formats.

- Explicit type casting of function pointers and return values is recommended to ensure that appropriate values are passed between routines.

- Calling conventions should conform to the varargs semantics.
- Character and integers may be signed or unsigned. Explicitly specify your preference with the appropriate compiler switches to avoid unexpected definitions and runtime results.

Understanding Architecture Issues

The DG/UX system is based on Motorola's 88K architecture. You will need to review your code to ensure that it meets the following architectural requirements:

- Floating point calculations will use the IEEE-754 format.
- Stack growth is generally downward (from higher addresses to lower addresses), but this may vary based on build options. Directly manipulating the stack is not recommended.
- The DG/UX system handles signals based on the BCS model.
- The DG/UX system expects "big-endian" byte ordering of the data.
- The DG/UX system uses 4K page size calculations.

Tips and Techniques for Creating Portable Code

The general tips for creating portable software are:

- Use specified interfaces.
  Standards define user interfaces and the functionality of an operation; however, implementation specifics may differ among vendors. Code that conforms to the described user interface rather than the implementation behavior is generally portable to other host UNIX systems, even if their implementation differs. For example, Data General's DG/UX system uses a proprietary file system structure, but still supports all standard filing system interfaces.
- Use **curses** and terminal libraries.
  The DG/UX system includes robust definitions of Data General terminals.
- Use **lint**.
  The **lint** program will help identify potential problems with your code. Code that runs through lint cleanly should not raise porting problems.
- Use type casting with pointers.
  Although many systems will allow you to use mis-matched pointers, this

coding practice may introduce bugs that are difficult to identify as you port your code across various architectures.

- Use care when defining and passing structures.
  The semantics for passing structures do not follow a set standard. If your code passes structures rather than pointers to structures, you should review the appropriate language reference manual, and ensure that your code conforms to the language requirements. Also, many compilers will introduce padding into structures as required by the machine architecture. These alignment requirements may produce unexpected results at runtime if your code uses implicit unions (references an entity differently than it is declared).

- Use rigor in processing signals.
  Review code that handles signals to ensure that appropriate signals are trapped and processed. The numbering scheme for signals is changing to allow merging BSD and System V support. The system header files define the signal names and numbers that are recognized by the DG/UX system.

- Use IFDEFS to identify code changes.
  This technique will allow you to maintain a common code source that can be compiled and run on many different environments.

- Isolate modules that interact with hardware.
  Code may need to depend on its operating environment. When this is necessary, the code is easier to identify if it is contained in an independent code module.

End of Chapter

Chapter 3
Using Data Formats and Transfer Media

This chapter explains how to transfer your program to the DG/UX system using various media and data formats.

Loading Source Code and Data

The first step in porting a program is to get your program source loaded onto the system. Usually, you will load either from tape or from network.

Loading from Tape

The QIC 150 MB QIC cartridge tape drive is a fully supported SCSI device on the DG/UX system. You may also read and write QIC 120 MB cartridge tapes; QIC 40 MB and 60 MB cartridge tapes may be read only. The different tape densities are selected by entering the appropriate device name in the read/write command line.

In addition to the QIC cartridge tape, the 1600 BPI 1/2 inch reel to reel SCSI tape drive is supported by the DG/UX system. It is possible to connect other SCSI tape drives, however, the device must be compatible with the DG/UX SCSI device drivers. The DG/UX SCSI device drivers require two optional commands described in the ANSI SCSI-1 specification: inquiry and space. The optional commands supported but not required are ready, mode sense/select, and erase.

The DG/UX system supports the standard **cpio**(1), **tar**(1), and **dd**(1) utilities used by most UNIX systems to write to a tape. To write tapes, use the appropriate utility along with the device name to specify the tape drive/density.

Note: Because they depend on the structure of the file system, **finc**, **frec**, and **volcopy** should not be used when porting applications; using these functions will make your code LESS portable to other systems.

You can use three commands to load from a tape:

cpio The format of **cpio** tapes depends on the architecture of the system on which they were written. The **cpio** compatibility option (**-c**) lets you read the headers of a tape from any system, and is compatible with the POSIX

extended cpio format. The tape must have been written using **−c** and must be read using **−c**. While **−c** lets you read the header, it does not help you read the body of the tape. In particular, the byte order of information in the body of the tape may be swapped compared to the DG/UX system. If the bytes on your tape are swapped, use the **cpio -s** option to swap the bytes back as they are read in. The default byte order for **cpio** on DG/UX is "Big-Endian", where the most significant byte is stored in the lowest address. The default blocking factor is 512 bytes. You can use the **−B** option to select a blocking factor of 5120 bytes, and the **−C** option to specify the size of the blocking factor in bytes.

The following command line will usually work for reading **cpio** tapes:

cpio -icdvB < *device_name*

The following command line will copy files to a tape:

ls *.c | cpio -oB >/dev/rmt/0n

All files in the working directory with the .c extension would be copied to the tape on **/dev/rmt/0n**. The **−B** option blocks the tape at 5120 bytes/record, and writing the file to /dev/rmt/0n leaves the tape positioned at end-of-file (instead of rewinding it). Using the no-rewind tape feature, lets you make multivolume tapes. If you want the tape to rewind, specify **/dev/rmt/0** in place of **/dev/rmt/0n**.

tar We recommend that you use the following **tar** command line for reading **tar** tapes:

tar -xvf *device_name*

The **−x** option extracts the named files from the tape and lets **tar** read the tape in the same blocking factor used to write the tape. **tar** is compatible with the POSIX extended tar format. Note: The POSIX implementation of **tar** will not be available in the early software release.

dd **dd** lets you read and write tapes with no effect on file format. It is possible to do conversions, such as ASCII to EBCDIC or blocked to unblocked; and the files=n option allows concatenation of multiple input files for tape. Cases when **dd** would be used, include:

- Reading non-standard formats or tapes from non-UNIX systems.
- Reading ANSI tapes.
- Writing system-boot tapes.

Loading from Network

The DG/UX system supports the TCP/IP communication network and the Network File System (NFS) network file sharing system.

The File Transfer Program (**ftp**) can be used to transfer files from one host to another using TCP/IP. On a diskless workstation, you will be mounted on disk(s) with the Network File System. Use a remote tape device to load your tape onto a disk mounted on your system.

Another mechanism for transferring files from one UNIX system to another is the **rcp** program. **rcp** can copy directory trees as well as files.

The DG/UX system also supports **uucp** with the Honey-DanBer enhancements for UNIX to UNIX copies. Refer to the *User's Reference for the DG/UX System* manual for more information on **uucp.**

When porting from non-UNIX environments, you may also use the DG/UX system serial port to load your sources. An asynchronous communications protocol package such as **kermit**, would be required.

See the *Installing and Managing DG TCP/IP (DG/UX)* for details on TCP/IP setup; and see *Managing NFS and Its Facilities on the DG/UX System* for details on NFS setup.

Understanding How Bytes Are Stored

Byte ordering refers to the order in which bytes are stored in a computer's memory. The computer's hardware architecture determines the byte ordering.

Consider a 32-bit word made up of four bytes: b0 b1 b2 b3, where b0 is most significant and b3 is least significant. In "Big-Endian" systems, the bytes are stored with b0 in the lowest address position: b0 b1 b2 b3. In "Little-Endian" systems, the bytes are stored in reverse order: b3 b2 b1 b0.

The AViiON system, based on the MC 88000 architecture, stores data with the most significant value at the lowest address, i.e. Big-Endian byte ordering. Other Big-Endian systems include Data General's MV/Family, MC 68000, SPARC, and the IBM 370 architectures. Little-Endian systems include Digital's VAX and Intel's 8086, 80286, and 80386 architectures.

As you can see, byte ordering varies with the different hardware architectures. Portable programs do not depend on a specific architecture or byte order.

End of Chapter

Chapter 4
Setting Up the Environment

This chapter describes how to set up the programming environment. Major topics are as follows:

- Using the Bourne and C shells
- Manipulating the line discipline
- Handling file system issues

Using the Bourne and C Shells

The DG/UX system supports two shells: the Bourne shell (sh) and the C shell (csh). The Bourne shell is part of AT&T UNIX System V. The C shell is part of the Berkeley UNIX System.

This section describes the following topics:

- Differences between sh and csh
- Switching between sh and csh
- Shell scripts in sh and csh
- Using Editread

For more information on shells, shell scripts, and editread see *Using the DG/UX system*.

Using sh and csh—the Differences

Both the Bourne shell and the C shell perform basically the same functions, but there are a few differences:

- The C shell has a history facility and an aliasing facility; the Bourne shell does not. However, DG/UX does provide editread, a history and line editing facility that can be used in both shells.
- The C shell has more features for job control. In csh you can run a job in background mode and bring it into foreground mode. In sh, you can also run a job in background mode, but you cannot bring it into foreground mode. In csh, if a job running in the background tries to do terminal I/O the job will be stopped. The job may be restarted by bringing it into the foreground. In sh, if a job running in the background attempts to do terminal I/O it will not be stopped. This can cause problems because the foreground job may also be doing terminal I/O.
- There are some differences in shell programming language and syntax.
- The Bourne shell uses the System V line discipline and the C shell uses the BSD line discipline.

Switching Between sh and csh

Regardless of your login shell, you can change shells during a work session by issuing the appropriate command.

To change to the Bourne shell:

```
% sh <NL>
```

To change to the C shell:

```
$ csh <NL>
```

To terminate the current shell and return to the shell that invoked it, use this command:

```
<Ctrl-D>
```

NOTE: In the C shell only, if you set the ignoreeof variable, using <Ctrl-D> will not terminate the shell. You will have to use the **logout** or **exit** command instead.

Writing Shell Scripts

Before you write a shell script, you will have to decide which shell to use. It will determine some of the commands, options, and programming constructs used.

You can execute a script written in the Bourne shell language in the C shell by inserting this command in the first column of the first line:

```
#!/bin/sh
```

Similarly, you can execute a script written in the C shell language in the Bourne shell by inserting this command in the first column of the first line:

```
#!/bin/csh
```

It is recommended that you put one of the previous commands at the beginning of your shell script to ensure that it is executed in the proper shell.

Using Editread

The editread facility is a unique DG/UX system interface that you can optionally invoke for editing command lines that you enter at either shell. Additionally, editread offers a history facility that saves your previously typed commands for later recall and execution. The history facility provides the same basic function as the C shell history with some differences. For both shells, editread offers a broad range of cursor control and line editing keys. The editread facility can also be used with other DG/UX programs such as **mxdb** and **crash**.

Editread is initially off. To enable it and accept the default settings, follow these procedures:

1) In your home directory, create an empty file and name it .editreadrc.

2) Log out and log back in.

3) To read a quick summary of the current editread default values, type this command from the shell: <Ctrl-R>

The editread facility uses the TERM variable setting to determine some of the characteristics of your terminal so it is important that this is set correctly. By default, the process control keys defined in the editread facility are copied from your terminal line discipline. As an alternative you may choose to redefine some keys in your **.profile** file (or **.login** file for C shell users) and those key definitions will be exported to editread. If you redefine a function in the editread facility, you should make a corresponding change to your line discipline. Editread and your line discipline should be in agreement.

NOTE: Editread may be slow over heavily loaded networks.

Manipulating the Line Discipline

The line discipline facility acts as the kernel's interface to the terminal. The line discipline intercepts data coming from the terminal, buffers it, and does some initial processing of special keys. For example, if your program has requested a line of data from the terminal, the line discipline will buffer this data as the data comes in, interpret certain special keys, and when complete, pass the buffer back to the kernel and thus to your program. On input or output to the terminal, the line discipline can also be set to map certain functions to a particular control sequence.

In general, the line discipline's operations are transparent to your program. However, if terminal-control issues do arise, there are several areas related to the line discipline that you may need to check. The following paragraphs are provided in case you encounter terminal-control problems.

The DG/UX system supports both the System V and BSD terminal line disciplines. Although these two line disciplines are largely the same, your program may use some of the unique features of its source system's line discipline. Therefore, you may need to select the line discipline you need. You can use the **ioctl**(2) system call or the **stty**(1) command to set which line discipline is active for your terminal (see the **ioctl**(2) manual page in the *Programmer's Reference for the DG/UX System (Volume 1)* or the **stty**(1) manual page in the *DG/UX System User's Reference Manual*). Also notice that if you log in directly to the Bourne shell (**sh**), the System V line discipline will be active. If you log in directly to the C shell (**csh**), the BSD line discipline will be active. Whenever **csh** is executed on the DG/UX system, it activates the BSD line discipline and restores the previous line discipline on exit. Executing **sh** after initial login does not affect the line discipline.

If you are using a terminal interface library, your program must include a header file that defines all line discipline characteristics so your program can refer to them by name. **termio.h** defines the System V line discipline characteristics. **sgtty.h** defines the BSD line discipline characteristics. For programs using BSD and System V line disciplines, you use **berk_sgtty.h** in addition to **termio.h**. These include files are described in **termio**(7), **tty**(7), **ioctl**(2), and **modemap**(7) (for more information on **termio(7), tty(7),** and **modemap**(7) see the *System Manager's Reference for the DG/UX System*).

Switching Between the System V and BSD Line Disciplines

Regardless of your initial login shell and line discipline, you can change from one line discipline to the other by issuing the appropriate command at the shell prompt:

To change from the System V to the BSD line discipline: stty line 1

To change from the BSD to the System V line discipline: stty old

Switching Between 7-bit and 8-bit Line Disciplines

From either line discipline you can switch between 7-bit and 8-bit mode by using the appropriate command at the shell prompt.

Under the System V line discipline -

To enable 7-bit mode: stty cs7 parenb parmrk

To enable 8-bit mode: stty cs8 -parenb -parmrk

Under the BSD line discipline -

To enable 7-bit mode: stty -pass8

To enable 8-bit mode: stty even odd pass8

Handling File System Issues

The DG/UX system uses a file system structure that is, in most ways, compatible with both System V and BSD. You will find the following System V features in the DG/UX file system:

- First-in-first-out (FIFO) special files (also called named pipes).
- Newly created files receive the process's effective group ID. On BSD, newly created files get the group ID of the parent directory.

You will find the following BSD features on the DG/UX file system:

- Symbolic links
- Sockets as defined for the UNIX domain under TCP/IP
- Long filenames and pathnames

Recognizing Differences Between the DG/UX and the System V File Systems

In general, as long as you follow standard language practices, the file system structure will be transparent to your program. However, the following System V file system features might cause problems:

- A 14-character maximum filename size and 100-character maximum pathname size
- No direct access to System V-specific structure of directory entries
- The use of streams

The next three paragraphs will address these potential problems.

As on BSD, filenames on the DG/UX system can be up to 256 characters long, and pathnames may be up to 1024 characters long. The structure of directory entries reflects this additional length. Programs that presume a limited-length filename or pathname or that presume a particular directory structure may run into trouble. For example, a program based on System V file systems might allocate a fixed-length, 14-character variable for a filename. Because DG/UX filenames can be much longer, they may overflow this fixed-length variable. Similarly, a program may attempt to read a directory expecting the fixed, 14-character filename entries.

Another common problem that may arise in System V programs relates to how some system calls access the contents of a directory file. Specifically, some programs expect to be able to read a directory using the **open**(2), **read**(2), and **close**(2) system calls. These calls rely on implementation-specific information and may not work

across different UNIX systems or even across different releases of the same system. Programs that use these calls should be changed to use the **opendir**(3), **readdir**(3), and **closedir**(3) library routines, which make the current directory structure transparent to programs.

The DG/UX system does support the System V streams system calls but it does not provide the Transport Layer Interface (TLI) library. If this will cause problems then **socket**(2) may be substituted for stream functionality.

Working with the DG/UX /dev Directory

The /dev directory on the DG/UX system has three major differences from other UNIX systems:

- Many device names are unique to the DG/UX system.
- No regular files may be created in this directory, only nodes and links.
- Files created in this directory may not be present after a reboot. In order to have needed files created in /dev at boot time, the init scripts should be edited.

End of Chapter

Chapter 5
Working with Terminals and Keyboards

This chapter discusses issues involving terminals and keyboards. Major topics are as follows:

- Using Terminal Interfaces
- Working with Supported Terminals
- Working with Character Sets
- Using International Keyboards
- Using a Graphics Console on a Workstation

Using Terminal Interfaces

Terminal interfaces available on DG/UX are **curses**, **terminfo**, and **termcap**. The following sections discuss what header files are required in your programs and how to compile using the specified interface. We would like to stress that the **curses** library can emulate the **termcap** library. And, because **terminfo** is intended to replace **termcap**, we recommend that you link your programs with the **curses** library rather than the **termcap** library.

Working with curses Routines

Curses is a library of routines that you may use to write screen management programs on the DG/UX system. **curses** routines write to a screen, read from a screen, control character attributes, and build windows. They also have advanced features that draw line graphics and provide soft labels for function keys. **curses** on DG/UX accesses information about the terminal from the terminfo database. Terminfo definitions are located in /usr/src/lib/libcurses/terminfo/*.ti. The compiled files are located in /usr/lib/terminfo/?/*. The curses routines are located in /lib/libcurses.a. To direct the link editor to search this library, you must use the -l option with the cc command. The command line for compiling a curses program is:

```
cc file.c -lcurses
```

The source program must include the header file, **curses.h**. You must also set the TERM variable, a shell environment variable that specifies a name for the terminal you are using. If you are using the Bourne shell, remember to export TERM.

Working with terminfo Routines

Some programs must use lower-level routines than those that **curses** provides. The **terminfo** routines are lower-level routines that do not manage your terminal screen, but rather give you access to strings and capabilities which you can use to manipulate the terminal.

The commands for compiling and running a program with **terminfo** routines are the same as those for compiling and running a **curses** program. The curses.h and term.h header files are required because they contain definitions of the strings, numbers, and flags that the **terminfo** routines use. References to these header files must be included in your source program.

Working with termcap Routines

The **termcap** terminal database is stored in an ASCII file; all the information it contains is readable, unlike **terminfo**, which is compiled. Terminal definitions are located in /etc/termcap. The termcap routines are located in /lib/libtermcap.a. To direct the link editor to search this library, you must use the -l option with the cc command. The command line for compiling a termcap program is:

```
cc file.c -ltermcap
```

You must also set the TERM variable, a shell environment variable that specifies a name for the terminal you are using. If you are using the Bourne shell, remember to export TERM.

NOTE:

If you are having problems with terminal control, check to make sure that the TERM variable is set correctly for your terminal.

Working with Supported Terminals

Data General supports a wide range of terminals. For a complete list of supported terminals, refer to the terminfo and termcap databases. For a partial list that includes all Data General terminals, you can also consult the **term**(5) man page. The following section discusses Data General terminals in emulation modes.

Using Data General Terminals in Emulation Modes

To ensure that your application software runs properly from a particular terminal:

1) Configure your terminal by invoking the terminal configuration menu. Refer to the manual that comes with your terminal for instructions on how to configure your terminal to the emulation mode specified for it below.

2) Configure the line discipline. The line discipline acts as the kernel's interface to the terminal. The line discipline intercepts data coming from the terminal, buffers it, and does some initial processing of special keys. Refer to Chapter 4, "Setting Up the Environment," for instructions on how to configure the line discipline.

3) Set the TERM variable for your particular terminal, as specified below.

Using DEC Emulation mode

Data General's new terminal product line—the D216, D412, and D462 terminals—have DEC emulation modes. The emulation modes are covered by vt100 and vt220 terminfo/termcap entries (e.g. DEC vt100 emulation mode would use TERM=vt100). D216 terminal emulates the VT100; however, D412 and D462 terminals emulate the VT220. VT220 emulation contains VT100 functions, plus additional functions that improve the terminals' speed and appearance. Your terminal is configured by invoking the Port Menu with SHIFT-N/C.

Using Tektronix Emulation Mode

Data General's D462 terminal emulates the Tektronix 4010 graphics terminal. Tektronix emulation mode is covered by the tek4010 terminfo/termcap entry (i.e. Tektronix emulation mode would use TERM=tek4010). Your terminal is configured by invoking the Port Menu with SHIFT-N/C.

Using ANSI Mode

Data General's older terminals are supported in ANSI mode only. Users porting software using Data General's older terminals should note that DG mode is unsupported. Thus, terminal definitions for Data General terminals in DG mode are no longer actively supported, although a few still remain in our terminfo and termcap databases. Entries for the following Data General terminals in ANSI mode exist:

D210
D211
D214
D215
D220
D410
D411
D460
D461
D470
D577
D578

Your terminal is configured by invoking the Terminal Configuration Menu with N/C, or by setting DIP switches at the back of the terminal.

Using xterm Emulation Modes

Because X windows is supported, the xterm terminal type is supported as well. The standard I/O goes through an emulation of DEC vt102 or Tektronix 4014 terminals. The TERM variable is automatically set to xterm when a window is opened (if an xterm entry is found in terminfo/termcap databases) If the xterm entry is not found, terminfo/termcap database are searched for vt102, then vt100, and then tek4014 entries. If you use terminfo/curses and termcap, you should have no problems, however, the application must be able to handle the window being resized. See the graphics chapter for window resizing implications and the X manuals for general xterm issues.

Using a Graphics Console on a Workstation

The workstation can be booted from the graphics console. The TERM variable is automatically set to xterm when a window is opened (if an xterm entry is found in terminfo/termcap databases) If the xterm entry is not found, terminfo/termcap database are searched for vt102, then vt100, and then tek4014 definitions.

Table 5-1 lists features of the workstation monochrome monitor.

Table 5-1 Workstation Monochrome Monitor Features

Feature	Description
Graphics Processors	Monochrome -- NEC uPD72120
Pixel Aspect Ratio	1:1
Refresh Rate	70 Hz, flicker-free
Controls	Brightness, contrast
Monitor Format	20″ landscape
Displayable Resolution	1280x1024
Addressable Resolution	1280x1638

Using Data General Terminal Features

Table 5-2 provides a summary of major features in the D216, D412, and D462 terminals:

Table 5-2 Features of D216, D412, and D462 Terminals

Feature	D216	D412	D462
Windowing		x	x
Dual Port	x	x	x
80 Columns	x	x	x
132 Columns		x	x
Graphics			x

Working with Character Sets

The following character sets are available on the D216, D412, and D462 terminals:

Keyboard Language
U.S. ASCII
U.K. National
French National
German National
Swedish/Finish National
Spanish National
Danish/Norwegian National
Swiss National
Kana G0 Set
D.G. International
Kana G1 Set
Word Processing, Greek, Math
Forms-ruling (Line drawing) set
D.G Special Graphics (PC characters)
VT220 Multinational
VT220 Line Drawing

Curses and **terminfo** support the Forms-ruling set to draw line graphics, and the international character set to support European languages in eight-bit communication modes. Refer to your terminal programmer's manual to determine what character sets are available on your particular terminal and how to select them. A complete list of terminal programmer's manuals is provided in the Related Manuals section after the Index.

Using International Keyboards

The D216, D412, and D462 terminals support 15 keyboard nationalities:

Arabic
Canadian/French
Canadian/English
Danish/Norwegian
French
German
Hebrew
Italian
Katakana
Spanish
Swedish/Finnish
Swiss/French
Swiss/German
United Kingdom
U.S.

Refer to your terminal programmer's manual to determine what nationalities your particular terminal supports. A complete list of terminal programmer's manuals is provided in the Related Manuals section located after the Index..

End of Chapter

Chapter 6
Using Software Development Tools

This chapter discusses software development tools for performing the following tasks:

- Checking C syntax (lint)
- Building complex programs (make)
- Controlling source code (RCS and SCCS)
- Debugging programs (mxdb, sdb, and dbx)
- Using preprocessor functions (IFDEF)
- Editing source files (vi, ed, and sed)

Checking C Syntax (lint)

This section describes what **lint** is, discusses how it is useful in the porting process, and gives examples of messages **lint** generates.

What is lint?

Once you have loaded your program, you are ready to check the code and then compile and link it. The DG/UX system provides a compile time tool, **lint**, that can help catch and correct many porting problems before you run the program. **lint** is a major tool for measuring your program's portability. **lint** is a C program checker for C language programs. It attempts to detect features of C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compiler. **lint** detects unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. It also checks for functions that return values in some places and not in others, functions called with varying number of arguments, and functions for which the values are not used or are used but not returned.

Using lint

The **lint** command has the following form:

lint [*options*] *file*.**c** ...

where *options* are optional flags to control **lint** checking and messages, and *file*.**c** is the name of the file to be checked. *Using DG/UX Programming Tools* contains a complete list of options and gives details on their usage. **lint** provides options that are common with those of the compiler command cc, e.g., -I, -D, and -U. -I specifies the include directory. -D and -U respectively predefine and undefine preprocessor symbols. If you compile using these options, you must also use these options when you use **lint**.

By default, **lint** checks programs against the standard C library (**lint** version of the standard C library). However, when the -p option is used, the portable C library is checked. It contains the standard library routines which are expected to be portable across various machines. You can also include a **lint** version of the math library and the **curses** library by inserting -lm on the command line to include the math library and inserting -lcurses to include the **curses** library.

Understanding Differences Between cc and lint

Although **cc** and **lint** both check C syntax, the programs are different.

- **cc** is a command that invokes the compiler, which generates error messages and object code.

- **lint** is a syntax and program checker that generates error messages and no object code.

- **cc** compiles each source file as a separate entity. **lint** can handle all constituent files at once. It can therefore cross-reference function and variable usage throughout the whole program.

Interpreting lint Messages

While catching many portability errors, **lint** cannot catch every error. The following section describes the types of messages **lint** will output. Please note that this is not an exhaustive list. See "Using Libraries and System Calls" and "Understanding Hardware Architectural Issues" chapters for additional information on areas **lint** can isolate.

- `value type declared inconsistently`

  When you pass an argument, you must make sure that the type of expression you pass matches the type that the function expects. For example, the **qsort** function expects the pointer to the array start to be a pointer to a character. You must explicitly cast the array you pass as a character pointer. In addition, you must call the function compare with two character pointers.

  When a function's return type is not defined, the default return type is integer. All library functions that are used in the program must also have their return type explicitly declared, or the returned value will be treated as an integer.

- `name used before set`

  lint attempts to detect where a variable is used before it is initialized. **lint** detects local variables (automatic and register storage classes) whose first use appears earlier than the first assignment.

- `nonportable character comparison`

  The following code produces a "nonportable character comparison" message on the line comparing getc's return against EOF. EOF has the value -1, which will fit in a char on some systems, but not others.

```
#include <stdio.h>

copyfile (fin, fout)
FILE *fin, *fout;
{
     char c;

     while ((c = getc(fin)) != EOF)
           putc(c,fout);
```

```
}

        if ( (c = getchar ()) < 0)....
```

- `illegal combination of pointer and integer`

  This means you tried to assign a pointer to an integer variable or vice versa. The following code sample produces the "illegal combination" message:

```
void f()
{
   int a; char *b; int *c;
   a = c; b = a; *b = a;
```

  Be sure to assign pointers only to pointer variables.

The **lint** program also generates messages that are not fatal, i.e. your program can execute successfully. For example, when a variable is defined but not used, **lint** generates the message "variable_name defined but not used." Although this does not produce a fatal error, it is not a good programming practice.

Building Complex Programs (make)

This section discusses what make is, why it is useful in the porting process, and the differences in System V and Berkeley functionality.

What is make?

The **make** program is a useful tool throughout the entire porting process because recompiling and relinking will occur repeatedly during the process. **make** provides a simple mechanism to maintain up-to-date versions of programs that result from many operations on a number of files. It is possible to tell **make** the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, **make** will create the proper files simply, correctly, and with minimum effort.

Using make

The **make** command has the following form:

make [*options*] [*macro-definitions*] [*targets*]

where *target* can be a makefile or a particular target name. **make** executes commands in the makefile to update one or more target names. **make** provides a macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name of the macro by a dollar sign. *Using DG/UX Programming Tools* contains a complete list of available options and also gives more detailed information on using **make**.

A makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a ":", then a (possibly null) list of prerequisite files or dependencies. Text following a ";" and all following lines that begin with a tab are shell commands to be executed to update the target. Note that shell commands must be preceded by a tab character at the beginning of the line. A # in column 1 denotes a comment line. The first line that does not begin with a tab or # begins with a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><new-line> sequence.

Example

The following example illustrates the use of macro definitions, e.g. $(SRC), $(OBJ), $(LIBS), and the inclusion of **lint :**

```
#
# Makefile for example
#
SRC =      zip.c zap.c
OBJS =     zip.o zap.o
LIBS =     -lcurses
LINT=      lint
DEFS=      -DHELP
BIN =     /bin

pgm:       $(OBJS)
           $(CC) $(DEFS) -o pgm $(OBJS) $(LIBS)

lint:
           $(LINT) $(DEFS) $(SRC) $(LIBS) > pgm.lint

clean:
           -rm -f $(OBJS) pgm core

install: pgm
           cp pgm $(BIN)
           chmod 755 $(BIN)/pgm
           echo pgm install in $(BIN)
```

Using System V or BSD make: the Differences

The **make** command is an example of a command for which the functionality differs between BSD and System V. DG/UX **make** has System V functionality. For users porting software from BSD systems, please note the following differences:

- circular dependencies

  The BSD implementation allows circular dependencies, and the DG/UX implementation does not. Circular dependency means that during a build, a makefile may specify that program "A" be made before program "B" and also specify that "B" be made before "A". BSD makes an arbitrary decision about which program to make first. Such circular dependencies must be removed to run a makefile under the DG/UX system. If you have a BSD program, you also should check **make**(1) for differences in default values.

- include name

  DG/UX **make** uses the string "include" to indicate include files. If you have a filename that begins with the string "include", such as include.o:include.c, the DG/UX **make** will try to open a file specified by the remaining characters on the line. If you have files beginning with the string "include," you must rename your file. Refer to the **make**(1) man page for details on the DG/UX system implementation and default inference rules.

Controlling Source Code

This section discusses controlling source code revisions.

When moving software from one operating system to another, there are certain times when code changes must be made in order for the software to run on the new operating system. In order to maintain your original source and also track all changes that you make to the code, you will need to use a source code control system. The DG/UX system provides two source code control systems, Revision Control System (RCS) and Source Code Control System (SCCS).

RCS is a system that uses a BSD-defined, free form ASCII file. RCS retains only the changes for previous revisions and stores the entire file for the latest revision. Thus it can quickly rebuild the latest revision.

SCCS is an AT&T System V defined ASCII file control system. SCCS stores the file changes with the original file, and therefore, must rebuild the specified version by applying deltas, in order, to the initial version until the desired version is obtained.

Both source control systems let you rebuild any revision stored within the source code control file. If you are currently using SCCS and would like to use RCS, use the **sccstorcs** command to build an RCS file from an SCCS file.

Using the Revision Control System (RCS)

The Revision Control System (RCS) contains multiple revisions of text, an access list, a change log, descriptive text, and control attributes. Rcs manages the storing, retrieval, logging, identification, and merging of revisions. Rcs is useful for keeping a logged account of text that is frequently changed, such as programs and documentation.

The user interface for **rcs** is simple for the novice user. The only commands that are essential to know are **ci** and **co**. **ci** stands for "checkin" and saves the contents of the file into an archived RCS file. **co** stands for "checkout" and retrieves the specified revision from the RCS file.

How to Get Started

Here are the steps to take:

1) Create and initialize a new RCS file. (RCS files end in ',v'. All other files are considered working files.) The following command line creates a new RCS file named foo.c,v and locks it for editing:

```
rcs -i -l foo.c
```

The **rcs** program will ask for you to enter a description of this file. When you have entered the description, press return and then enter 'Ctrl-D'. **rcs** tries to place the file first in it's default directory ./RCS and then into the current directory. You can also specify the directory by providing a path prefix for foo.c.

2) Make changes to your file using your favorite editor. For a list of provided editors, refer to "Editing Source Files" later in this chapter.

3) Store the working file contents into the corresponding RCS file. The following examples check in the file foo.c and assign a new revision, provided the file was checked out with the -l option:

```
ci foo.c,v
ci foo.c
ci RCS/foo.c,v
```

If the file resides in a directory other than ./RCS or the current directory, you must specify the path prefix. **ci** can be used instead of the **rcs** command to create the initial **rcs** revision.

4) Check out the file for further changes. The following example checks out foo.c for reading and writing, locking the file:

```
co -l foo.c,v
```

Useful Tips for Using RCS

- The RCS caller must have read and write permission for the directory containing the RCS file and read permission for the RCS file.

- For **rcs** to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file, or the caller is superuser.

- Often there are numerous source files that compose a software package. Many times is may be easier to automate the check in and check out of files through shell scripts and make files. When using the automated method, two command line options will help the process run smoothly, -m*msg* and -q. The command line option for **ci**, -m*msg*, uses the string *msg* as the log message for all revisions checked in. The command line option for **ci** and **co**, -q, suppresses diagnostic messages.

- Using the -k option with ci allows you to check in the file, reserving the current revision number, creation date, author, and status. This option is extremely useful with software distribution.

Refer to the following related manual pages in the *User's Reference for the DG/UX Systems* and *Programmer's Reference for the DG/UX System (Volume 1)* for additional information on **rcs**: **berk_diff**(1) and **berk_diff3**(1), **ci**(1), **co**(1), **rcs**(1), **rcsdiff**(1), **rcsfile**(1), **rcsintro**(1), **rcsmerge**(1), and **rlog**(1).

Using the Source Code Control System (SCCS)

The Source Code Control System (SCCS) is a maintenance and tracking tool for changes made to a specified file. SCCS is given control of the file and as changes are made to the file, they are stored with the original file. SCCS stores history data about each file it assumes control over. The history data includes when and why changes were made and who made the changes.

How to Get Started

Here are the steps:

1) Create your SCCS file with the SCCS **admin** command. The following example creates a SCCS file and initializes it with the contents of **foo.c**:

```
admin -ifoo.c s.foo.c
```

All SCCS files must have names that begin with .s, hence s.foo.c. You may receive the message 'No id keywords (cm7). In this case, disregard this message.

2) Remove the **foo.c** file. It is no longer needed, since SCCS had control of the file.

3) Retrieve your file foo.c in read-only mode by using the following command:

```
get s.foo.c
```

This command will ouput the revision you have requested, along with total number of lines in the file and will create foo.c.

4) Retrieve your file foo.c in edit mode:

```
get -e s.foo.c
```

When you retrieve a SCCS file in edit mode, you set up a new **SCCS IDentification** (SID) number for the new delta you will create. This command creates **p.foo.c**, which is used by the **delta** command.

5) Make changes to your new file **foo.c** and save the changes in the original s.foo.c file using the command:

```
delta s.foo.c
```

Delta prompts you for an explanation on why the changes were made. If you are changing a number of files for the same reason, then create a text file that contains the explanation of your changes, say *change*, and issue the following command:

```
delta s.foo.c < change
```

This automatically inserts your text into the SCCS file and will not prompt you for the explanation.

Useful Tips for Using SCCS

- If an error occurs with a SCCS command, a code is listed at the end of the error message in the format **(cox)**, where x is the error code number. Use the help command to print a more complete description of this error.

```
help cox
```

- If processing terminates abnormally, you can retrieve your unsaved changes by copying x.filename to s.filename.

- Use the unget command to cancel a delta reservation set up for your file. This is useful when you have retrieved your file for editing in error.

- Do not create links to SCCS files. When processing, SCCS removes old files and creates new files, thus destroying your links.

- Use the -i option on the get command to force the changes of one or more deltas to be applied to the specified SCCS file.

- Use the -l option on the get command to create a l.filename that lists the deltas used in creating a particular version of the SCCS file.

Refer to the following SCCS commands in the *Programmer's Reference for the DG/UX System (Volume 1)*: **get**(1), **unget**(1), **delta**(1), **admin**(1), **prs**(1), **sact**(1), **help**(1), **rmdel**(1), **cdc**(1), **what**(1), **sccsdiff**(1), **comb**(1), **val**(1), **and vc**(1). Also refer to the "Support Tools" chapter in *Using DG/UX Programming Tools*.

Debugging Application Programs

The DG/UX system currently supports the Symbolic Debugger (sdb) for debugging C and FORTRAN 77 programs. In addition to **sdb**, The DG/UX system will provide the Multi-eXtensible DeBugger (**mxdb**) and the **dbx** debugger in a future release. **sdb** is useful for monitoring and controlling the execution environment as well as examining core images of aborted programs. Running your program under the control of **sdb** allows you to view the progress of your program up until the time of the abort. It also allows you the option of stepping around the failure point and continuing with execution. Using **sdb** to examine the program core image allows you to view the status at the point of failure and to determine the cause of the error.

Preparing for a Debugging Session

The **sdb** debugger handles programs written in C and FORTRAN 77. Compile your program from the DG/UX shell, using the appropriate C and FORTRAN commands. In order to use **sdb ,** you should compile with the -g and -ga option. The -g option puts line numbers and other symbolic information into the executable file. The -ga option generates a frame pointer for stack traces. Without the -ga option, the compiler optimizes the program to the point that stack traces become impossible. All modules (functions, procedures, libraries) that are a part of your program should be compiled with the -g and -ga options in order to use the debugger to it's fullest potential. If all modules are not compiled with these options, **sdb** will output an error message, but will continue to debug the other modules compiled with the options. It is also important not to strip the symbol table from the object file.

Getting Started with sdb

The command line for **sdb** is as follows:

sdb [*objfil* [*corfil* [*directory-list*]]]

objfil is an executable program file; the default is **a.out**. *corfil* is the core image file produced by the executing *objfil*; the default is **core**. *directory-list* is a list of directories separated by colons containing the source of the program; the default is the current directory.

After executing the **sdb** command, you will be prompted with an asterisk (*), which shows that **sdb** is waiting for a command. Table 6-1 gives a subset of **sdb** commands that you an enter at this prompt:

Table 6-1 Commonly Used sdb Commands

Command	Description
r *args*	Run program with given arguments
t	Print a stack trace
variable/	Display the value of the variable
p	Print the current line
<Ctrl-D>	Scroll 10 lines
proc:12b	Set breakpoint in on line 12 of funtion proc
q	Exit from sdb

NOTE: If your application uses a private shared library, the possibility arises that a program bug may be located in a file that resides in the shared library. Shared library data is not dumped to core files, and sdb does not read shared libraries' symbol tables.

For further information as well as an example of an sdb session, refer to "Using sdb (Symbolic Debugger)", in the *Using DG/UX Programming Tools*. Also refer to the **sdb** manual page in the *Programmer's Reference for the DG/UX System (Volume 1)*.

Using Preprocessor Functions

This section discusses what a preprocessor is and how preprocessor directives are used in the porting process.

What Is a Preprocessor?

A preprocessor is a generic program that prepares an input file for another program. The C preprocessor is part of the C compilation process. Statements in C source code which start with a # are instructions to the preprocessor. The preprocessor's main function is to include files and to perform macro substitutions.

The C preprocessor is called only during the first stage of compilation. It normally only handles text manipulation such as file merging.

The DG/UX system provides **cpp**, which is the standard preprocessor supplied with the AT&T System V C compiler.

In addition, the DG/UX system provides C preprocessors which are built into the DG/UX C compilers. Each compiler may have different default definitions. The **cpp** command may produce different results than if you use the preprocessor built into the compiler you are using.

The -E option can be used with the cc command to help debug preprocessor macros. This option tells the compiler not to compile the program, but instead, put the output of the preprocessor in the standard output file.

Using Preprocessor Directives

The main use for the preprocessor during the porting process is through the use of the #ifdef and #define directives. These directives allow parts of the C source code to be compiled only in certain cases. For example, given the following code segment,

```
#ifdef DGUX
    {execute this section of code only if DGUX is defined as true,
     i.e., only if running on a DGUX system}
    |
    |
    |
#endif
#ifdef BERKELEY
    {execute this section of code only if BERKELEY is defined as true,
     i.e., only if running on a BERKELEY system}
    |
    |
    |
```

```
#endif
```

you could compile your program with this line:

cc -DDGUX *progname*

which defines DGUX and will cause the "ifdef DGUX" code segment to be executed.

In the above example, code that is machine-specific will only be executed if that particular machine is defined using the -D switch at compile time.

The "#define" preprocessor directive is helpful in debugging code that you are porting. The following section of code will print the value of x if DEBUG is defined.

```
#define DEBUG
      |
      |
      |
#ifdef DEBUG
   printf("value of x" %d,x);
#endif
```

The above code lets the programmer check the value of x during the debugging process. When not debugging, he can simply make the #define statement into a comment. Note that the #define directive can be used in place of the -D option in the cc command. In the above example, the #define DEBUG could have been left out. In that case, you would compile the program with:

cc -DDEBUG

to define DEBUG.

Another preprocessor directive is "#include", which is used for file inclusion. The #include command can have one of two formats:

```
#include "filename"
     or
#include <filename>
```

Either of these lines will cause that line to be replaced by the contents of the file named filename. The quotes around the file name indicate that the search for the file should begin in the same directory as the source file. The angle brackets around the filename indicate that the search for the filename should follow the default path.

For more information about preprocessing, see Chapter 7, "Compiling and Linking Programs."

Editing Source Files

This section discusses the **vi**, **ed**, and **sed** text editors. When porting your application, you will need to use an editor if you have to make changes to your code. The DG/UX System provides several editors, including vi, ed, and sed. In addition, other editors such as gnu emacs are available. This section lists some of the common commands for the available editors. Use the editor with which you are most familiar.

Using vi

Vi is an interactive full-screen editor.

Table 6-2 summarizes the basic **vi** commands.

Table 6-2 Summary of Basic vi Commands

Command	Description
vi filename	Invoke vi
:wq or ZZ	Save file and quit vi session
:q	Quit vi session without saving

For more information about **vi**, type ":!man 1 vi" from vi, "man 1 vi" from the shell, or refer to *Using the DG/UX System: The Editors*.

Using ed

The **ed** program is a line editor. Table 6-3 summarizes the basic **ed** commands.

Table 6-3 Summary of Basic ed Commands

Command	Description
ed filename	Invoke ed
H	Turn on Help Message mode so you receive help messages when ed detects an error
h	Prints help message text for the most recent error
w	Save to filename you named when invoking ed
q	Quit ed and return to shell (must save first)
Q	Quit ed and return to shell (does not check for save)

For more information about **ed**, type "man 1 ed" or refer to *Using the DG/UX System: The Editors*.

Using sed

Sed is a non-interactive editor used for editing a copy of the text from a file or standard input. The edited version goes to the screen by default or you may redirect it to a file. Sed provides a convenient way to edit text located in an input file using a sed command you type on the command line.

For more information about **Sed**, type "man 1 sed" or refer to *Using the DG/UX System: The Editors*.

End of Chapter

Chapter 7
Compiling and Linking Programs

This chapter explains how to compile and link programs written in C, FORTRAN, Pascal, and assembler.

Compiling C Programs

Data General offers both GNU C and Green Hills C for use with the DG/UX system on AViiON workstations and servers. The GNU C compiler is included in the standard release of DG/UX 4.10. Green Hills C is an optional compiler that may be purchased from Data General. For detailed information on these compilers, please refer to the user's guides.

While both GNU C and Green Hills C conform to the ANSI C language definition, each provides language features that are not found in ANSI standard C. In addition, the compilers differ in their approach to implementation-defined behavior. Differences between these compilers are outlined in Appendix B, "Comparison of C Compilers."

The following list identifies issues that you should be aware of in porting to either compiler:

- The standard **cc** command invokes the GNU C compiler. GNU C may also be invoked using the **gcc** command. Facilities unique to **gcc** may not be accessible from the **cc** command; instead you must use **gcc** directly. The **ghcc** command, included with the Green Hills release media, accesses Green Hills C. GNU C, when invoked by the **cc** command, supports PCC features. When invoked by **gcc**, GNU C defaults to support most ANSI features and GNU C extensions. Green Hills C defaults to support PCC features. Both **gcc** and **ghcc** support options to enable varying levels of ANSI support.

  Appendix B, "Comparison of C Compilers," lists options you can use when invoking the compilers.

- Both GNU C and Green Hills C include C preprocessors that provide a superset of the features of ANSI Standard C. The compilers preprocess your program automatically before compiling them. Programmers that use cpp to debug preprocessor macros should be aware that **cpp** may provide inaccurate results when used with Green Hills C or GNU C because of differences in predefined preprocessor macros. If you want to preprocess

your program before the actual compilation phase, use the -E option when compiling.

- The asm statement (for inline assembly code) is supported by both compilers. If you are using the Green Hills compiler, you must compile with the -Z309 option to enable support for this feature. Green Hills C allows asm to appear in your code anywhere a statement can be used. GNU C supports asm by default; however, asm is not supported by GNU C if you are compiling your code with -ansi.

- Programs that expect unsigned chars and/or integers may exhibit problems since both GNU C and Green Hills C default variables of type char and int to be signed. This problem may be resolved by recompiling with options to make variables unsigned.

- Both GNU C and Green Hills allow automatic structures, unions and arrays to be initialized. Green Hills requires the use of the -ansi option in order to compile programs that initialize automatic structures, unions or arrays but does not require the -ansi option if these are globally defined. GNU C does not require any special compiler options.

- Problems may be seen in code that expects structures to be aligned at specific offsets. Both GNU C and Green Hills C align each struct, union, or array to the maximum alignment requirement of any of its components. For example, shorts are aligned on boundaries divisible by 2, longs are are aligned on boundaries divisible by 4. Members of a given structure may have different offsets on AViiON systems than on other systems, and the structures may have different sizes. Differences in structure alignment may be seen in the following example:

```
                GNU C and Green Hills C   MV/Family DG C
struct {
  short  a;     aligned at offset 0        aligned at offset 0
  short  b;     aligned at offset 2        aligned at offset 2
  short  c;     aligned at offset 4        aligned at offset 4
  long   d};    aligned at offset 8        aligned at offset 6
```

- sdb level debugging support is available with both compilers; however, GNU C provides debugging capabilities not seen in Green Hills. The Green Hills C compiler does not allow you to debug optimized code whereas GNU C does.

While many compilers have implemented some portion of the ANSI standard, there exists a large body of code that has been written for the Portable C Compiler (PCC) standard. The following list provides an overview of some of the major differences between PCC compilers and ANSI standard compilers. You may find that code that does not compile using the default PCC features compiles using ANSI options.

1) The ANSI standard more clearly defines preprocessing than the PCC standard. New control lines include #elif and #pragma. New operators have been defined for concatenation of tokens (##) and creation of strings (#).

2) Minimum significance of identifiers has been lengthened to 31 characters.

3) The keywords 'void,' 'const,' 'volatile,' 'signed,' and 'enum' have been introduced in the ANSI standard.

4) Characters may carry the sign bit or not, depending on declaration; i.e.: signed char, and unsigned char.

5) Declaration of 'void *' as a generic pointer type has been included in the ANSI standard.

6) Explicit ranges on mathematical types are defined. Headers <limits.h> and <float.h> contain these ranges.

7) Assignments of the form '=+' are no longer supported.

8) Structures may now be assigned, passed to functions, and returned from functions.

9) The sizeof operator now returns a type of size_t, as defined in <stddef.h>.

10) A pointer may be created to point beyond a given array, and arithmetic operations may be done on it.

11) Function prototypes (as in C++) are available. These declarations include the types of the parameters in function declaration.

Compiling FORTRAN Programs

The Green Hills FORTRAN 77 compiler is available for Data General AViiON systems running DG/UX. Green Hills implements the ANSI FORTRAN Standard, ANSI X3.9-1978, MIL-STD-1753. It is also compatible with the Berkeley 4.3BSD **f77** compiler and the VAX/VMS FORTRAN V4.6 compiler. Green Hills allows FORTRAN subroutines to be called from C programs. You can also call C functions from FORTRAN programs. Refer to the *Green Hills FORTRAN-88000 User's Manual* for more information.

Compiling Pascal Programs

The Green Hills Pascal compiler is also offered for Data General AViiON systems. Green Hills Pascal implements the ANSI/IEEE standard and the BSI/ISO Level 0. Many of the extensions present in the Berkeley 4.2BSD **pc** compiler have also been implemented. Additional features include support for argc and argv, set implementation, and separate compilation. Green Hills allows Pascal external variables, procedures and functions to be accessed from C functions linked with the Pascal program and vice-versa. Refer to the *Green Hills Software User's Manual: Pascal-88000* for more information.

Assembling and Linking Programs

Assembly language files may be assembled using the as command. By default, results are placed in a file ending with a .o suffix. Programs that use **m4** macros should be assembled with -m on the **as** command line in order to run the **m4** macro processor on the input to the assembler. Refer to the online manual pages for more information on **as** and **m4**.

The **ld** command combines object files into one file, performs relocation, resolves external symbols, and places the output into a file named **a.out**. If no errors occurred, the file a.out is executable. Refer to the online manual page for more information on **ld**.

End of Chapter

Chapter 8
Using Libraries And System Calls

This chapter discusses the standard subroutine libraries and system calls supported by the DG/UX system. It describes porting issues that result from differences in library routine and system call function and implementation.

Comparing BSD and AT&T Libraries

The DG/UX system supports both System V and BSD library routines and system calls. For a list of system calls, refer to Appendix A, "DG/UX System Calls and Commands."

In general, when there is a conflict between a system call found in both BSD and System V, the DG/UX system resolves the conflict in favor of System V. Usually these conflicts are minor. For example, when a file is created under System V and the DG/UX system, the group ID is taken from the creating process. Under BSD, the group ID is taken from the containing directory.

The areas of greatest difference between System V and BSD are the handling of signals and the input/output (I/O) control system (**ioctl**(2)) system call.

Handling Signals

The DG/UX system supports a hybrid of System V and BSD signal functionality. Details on DG/UX signaling are given in relevant manual pages, such as **signal**(2), **sigsys**(2), and **sigset**(3) in the *Programmer's Reference for the DG/UX System (Volume 1 and Volume 2)*. This section will cover a few of the major differences in signaling.

The DG/UX system supports all System V and most BSD signals. You may need to modify some System V programs to anticipate and perhaps handle the extra BSD signals.

By default, the DG/UX system follows the System V signal-handler protocol. In particular, a signal handler is always disabled while it is processing a signal. It remains disabled until the program specifically re-enables it. If your handler has not been re-enabled by the time a new signal arrives, the default action for that particular signal will be taken. BSD programs may presume that their signal handlers are

automatically re-enabled when the signal handler returns. If the program you are porting makes this assumption, the system default handler might be invoked at times when the program's signal handler should be invoked. This problem may manifest itself as an early program termination. Fix this signal-disabling problem by using the **sigvec**(2) system call to reset your program's signal handler.

The DG/UX system differs from BSD signal handling in that the program is aborted when a signal indicating a severe error occurs. This is true even if a user-defined signal-handler function is provided. When a user-defined signal handler is provided, the program is aborted when the signal handler finishes processing the error. Because the abort happens on the normal return from your signal handler, the only way to prevent such an abort is to have your signal handler exit with a non-local jump. The **setjmp(3C)** and **longjmp(3C)** library routines are examples of non-local jump routines. These routines will cause your signal handler to exit and program execution to resume at the point the **jmp** routine was called.

With BSD signal handling, if a system call is interrupted by a signal, the system call is automatically restarted after the handler returns. Under the DG/UX system, the interrupted system call returns an error (errno=EINTR). If your program presumes the BSD method, you may get some unexpected failures.

You can easily remedy these differences between standard DG/UX system signal handling and BSD signal handling by using the **berk_signal.h** include file instead of **signal.h**. **berk_signal.h** provides you with all the expected BSD signal-handling features. Alternatively, you can replace every call to **signal**(2) to **berk_signal**(2).

On BSD, signals are blocked while the handler is active and are unblocked when the handler returns. On BSD, because calls to **longjmp**(3C) do not return, **longjmp**(3C) automatically unblocks signals when called from a signal handler. On the DG/UX system, **longjmp**(3C) does not automatically unblock signals. If you use **longjmp**(3C) in a signal handler under **berk_signal**(2), you will need to unblock signals with a call to **sigsetmask**(2).

Another aspect of BSD signal support on DG/UX concerns signal numbering. In compliance with the 88open Binary Compatibility Standard, DG/UX no longer limits the number, which can be assigned to a signal, to values less than 32. In particular, the following BSD based signals have numbers greater than 32 on the DG/UX system:

- SIGURG
- SIGIO
- SIGXCPU
- SIGXFSZ
- SIGVTALARM
- SIGPROF

- SIGLOST

This change to higher signal numbers can lead to problems with code which uses these signals. The particular problem you may encounter if you use these signals is that the signal masks passed between user programs and the kernel via the **sigblock(2)**, **sigpause(2)**, **sigsetmask(2)**, and **sigvec(2)** system calls are only 32 bits in size. Thus, the mask bits for those signals cannot be directly represented via those system calls. If you must represent those signals explicitly in signal masks, you will need to change your source code to use the POSIX functions which can manipulate 64 bit signal masks. You will need to change any occurrence of local variables used to manipulate signal masks so that they will handle a 64-bit mask.

Using the I/O Control System

The DG/UX system supports both the AT&T and BSD terminal drivers. In some cases, the AT&T modes and ioctls can not be accurately reflected in the BSD mode set. For strategies in using the appropriate system calls to select the line discipline for your program, refer to "Manipulating the Line Discipline" in Chapter 4, "Setting Up the Environment."

Using DG/UX-specific Routines

DG/UX provides support for several library routines and system calls that are not derived from either AT&T or BSD. Some of these routines provide enhanced functionality in areas where AT&T and BSD based systems are incomplete. In other cases, they provide the functionality of a routine you may use on another system while taking advantage of special resources in the DG/UX operating system or AViiON hardware. The following is a list of some DG/UX specific routines you may wish to use in your application. Refer to Chapter 2 in the *Programmer's Reference for the DG/UX System (Volume 1)* for additional information on these routines.

- dg_ext_errno(2)

  Returns extended errno for the current process. The low order word contains the specific error.

- dg_file_info(2)

  Returns structures containing data that specifies the format of the information, the object referred to, and the file usage information for a specified process.

- dg_ipc_info(2)

  Returns static and configured state information on IPC components; shared memory, message queues, and semaphores.

- dg_mount(2)

  Mounts all "nfs" and "dgux" file system types. Attaches a file system to a directory, after which all references to that directory will refer to the root directory on the newly mounted file system.

- dg_mstat(2)

  Returns current attributes of a specified file. Dg_mstat also returns file status on the target of a symbolic link and returns status information for a mounted directory.

- dg_process_info(2)

  Searches for valid active processes and based on the select condition supplied by a parameter, returns information about that process. Only one search through the process table is made.

- dg_sys_info(2)

  Returns system information gathered from kernel databases.

- dg_xtrace(2)

  Lets a process control the execution of another process. It's primary use is to implement breakpoint debugging.

Handling Common Porting Problems

This section gives examples of some possible porting problems associated with library routines and system calls.

Missing Library Function

```
Symptom:  Link errors such as Undefined symbol.
Cause:    Missing library functions.

Solution: Replace calls with System V or user-supplied routines
          that operate similarly.
          Lint will catch this problem.
```

If you get an *Undefined symbol* error during linking, your program may contain a call to a library function that is not found on the DG/UX system. If this is the case, you will have to replace the call to the missing routine with a call to a DG/UX system routine that functions similarly or with a call to your own routine. This error can be the result of leaving an object or library off of the **ld** (link) command line or putting libraries in the wrong order.

To ensure that all the necessary libraries are being linked, use **cc -v** [*options*] to see the libraries that are being linked. If the library function is not available on the DG/UX system but a similar one exists, a macro definition can be used in the source file to replace the original function call with a call to the provided function as well as any re-ordering of the arguments.

Misuse of Library Implementation Internals

```
Symptom:  Compile-time error such as Undefined fields of structure;
          Link-time error such as Undefined symbol;
          runtime error such as unexpected value.
Cause:    Program makes assumptions about implementation of
          a library routine.

Solution: Change code as needed for each case.
          Lint will catch the compile- and link-time problems.
```

The DG/UX system follows System V compatibility at the user interface level. However, compatibility standards do not specify the internal implementation of command and library routines. Consequently, internal implementations may differ among vendors. You may have to revise programs based on implementation specifics.

The following list outlines three examples of the misuse of implementation specifics:

- **Different internal routines** — System V uses an internal routine called **_doprnt** to implement the **printf** family of commands (**printf**, **fprintf** and **sprintf**). Many programs directly invoke **_doprnt** because it allows easy use of a variable number of arguments. The DG/UX system does not have a **_doprnt** routine. Programs that use **_doprnt** must be revised to use a user-level routine. For example, though they do not behave identically, you can usually replace **_doprnt** with **vfprintf** or **vsprintf**, because they also handle a variable number of arguments.

- **Different data layout** — The standard input/output (I/O) library as given in the file **stdio.h** defines the FILE structure. The DG/UX system uses information in the FILE structure to access I/O streams (reading and writing to disks, terminals, pipes, etc.). Some programs may attempt to access information in the FILE structure directly and, in doing so, presume a particular layout of this information.

  Programs should not access the fields of the FILE structure except through the standard library routines.

- **Different resource allocation** — Your program may make assumptions about how a library routine operates internally. For example, it might presume that **malloc** allocates memory from a specific location. Many programs presume that a pointer allocated by **malloc** must be a pointer to the **sbrk** area (see "Memory Layout Issues" in Chapter 3). They may use this assumption to identify a pointer to the heap. This assumption is not valid on the DG/UX system.

  You may be able to avoid rewriting the code by using the alternative version of **malloc** found in **libmalloc.a**. As with System V, the DG/UX system has two sets of memory allocation routines. The default routines are tuned for the DG/UX system. The internals of the alternate set of routines are more like the alternate set available under System V.

  However, the best solution to this type of problem is to rewrite the code. Invocations of **malloc** should be made only on the basis of how **malloc** is defined at the user interface level, not on special knowledge of internal workings that may change.

The flock Library Function

```
Symptom:  Runtime error involving flock(3C).
Cause:    Difference in functionality of BSD flock(3C)
          and DG/UX system flock(3C).

Solution: Check functionality for flock(3C).
          Lint does not catch this problem.
```

The DG/UX system supports a version of **flock(3C)** based on System V's **fcntl(2)**

locking options. The DG/UX **flock(3C)** has slightly different functionality than BSD **flock**. This system call is likely to be used in database applications or other programs involved in multi-user file access. If your program follows BSD standards, check the DG/UX **flock(3C)** manual page in the *Programmer's Reference for the DG/UX System (Volume 2)* for details on the differences in functionality.

Reserved Subroutine Name Error

```
Symptom:    Segmentation violation error at runtime;
            Program accesses wrong subroutine.
Cause:      Using reserved subroutine names (as found
            in the C Language Reference).

Solution:   Don't use standard names for user-supplied
            subroutines.
            Lint will catch this problem.
```

Programmers sometimes give their own local subroutines the same name as a library subroutine. For example, a storage management routine may have an internal subroutine called **malloc** or **free**. As a result of this name duplication, either the standard library routine or the local routine may be invoked at runtime. This can cause unpredictable results in your program's operation, because it expects one routine and may get another.

The problems that arise from this error are very difficult to diagnose. If your program uses reserved names throughout its code, we recommend that you use the preprocessor to fix this problem as shown below:

```
#define malloc my_malloc
#define free my_free
```

By placing statements like the ones above at the top of each module that uses the user or local routine, the references to the local subroutine are automatically renamed throughout the code.

Variable to Subroutine Naming Error

```
Symptom:    Overwrite error at link time;
Cause:      Variable using system subroutine name.

Solution:   Rename the variable.
            Lint will catch this problem.
```

A variation on the naming-error problem occurs when the program has a global variable with the same name as a system routine. In this case, instead of accessing a different routine, the naming error will look like a memory allocation problem. You get an *overwrite error*. The solution to this problem is to rename your variable.

The sigpause System Call

```
Symptom:  Runtime error involving sigpause(2).
Cause:    Difference in functionality of BSD sigpause(2)
          and DG/UX system sigpause(2).

Solution: Check functionality for sigpause(2).
          Lint does not catch this problem.
```

The DG/UX system provides both BSD's and System V's versions of the system call named **sigpause**. In order to eliminate a name conflict, the name **sigpause** is used to refer to the System V call, and the name **berk_sigpause** is used to refer to the BSD call.

The implementation-specific signal **SIGEMT** (specific to PDP® systems) is not present in the DG/UX system; this may be significant for user programs that have used this signal as a user-defined signal, since it would be otherwise guaranteed to be unused on a non-PDP implementation such as the DG/UX system.

There is an issue with the statuses **EWOULDBLOCK** (BSD) and **EAGAIN** (present in both BSD and System V). The DG/UX system returns **EAGAIN** for SVID conformance. **EAGAIN** and **EWOULDBLOCK** will return an identical value.

End of Chapter

Chapter 9
Understanding Hardware Architectural Issues

This chapter discusses porting issues that result from differences in hardware architecture. Such issues result from assumptions commonly made by programmers about the hardware. We also discuss system memory alignment and layout issues.

This chapter also highlights some C language issues resulting from architectural differences. The types of language issues covered here result in runtime errors rather than compile-time errors. You may wish to check Chapter 7 for additional C language issues.

Handling Alignment and Storage Layout Issues

Between machines, the required alignment, storage layout, sizes, and bit patterns often differ. Problems arise when programs are designed to presume such architectural specifics. For example, on some machines the byte order of integers is the opposite of what one might expect. The AViiON system memory architecture is byte addressable with 32-bit addresses. Memory is allocated in blocks of 4096 bytes. Memory is laid out in the Big-Endian byte order format with the most significant byte of a multiple byte word in the lowest address. This is equivalent to the MV, IBM/370, SPARC, and Motorola 68000. This is opposite from the VAX and Intel 80x86.

Similarly, a program might make size or allocation calculations based on the storage layout (packing) rules of the original system. In both of the cases described, the program works correctly on the original system but cannot be moved to another system without major changes.

Given below are examples of some of the more common issues resulting from storage and addressing assumptions.

Using the NULL Pointer

```
Symptom:    Memory fault error at runtime.
Cause:      Indirecting NULL pointer.

Solution:   Remove all use of the NULL pointer to access
            a zero word.
            Lint will not catch this problem.
```

The NULL pointer is a special C language feature that allows you to test for an uninitialized pointer or to pass an error condition through a pointer. The NULL pointer is particularly prone to misuse. For example, on some machines, the NULL pointer points to a zero integer or a zero character stored in memory. Consequently some programs rely on de-referencing the NULL pointer to obtain a zero character or a value of zero. This practice is called indirecting zero. On AViiON systems, the NULL pointer does not point to zero.

Programs that expect the NULL pointer to point to a specific value may misuse it in several different ways. For example, your program is making assumptions about the NULL pointer if it uses:

```
if( *p ) ...
```

as a shorthand for:

```
if( p && *p ) ...
```

In this example, the program de-references a pointer without knowing whether it is the NULL pointer. De-referencing the NULL pointer would work only on machines for which the NULL pointer points to a zero character or integer. As with most UNIX systems, the AViiON system generates an addressing exception for this use of the NULL pointer.

Similarly, it is incorrect to pass the NULL pointer as a pointer to zero, as follows:

```
f(NULL)

  or

f(0)
```

The proper way to pass a null character string is:

```
f("")
```

The proper way to pass a pointer to zero is:

```
{ int i=0, f(&i); }
```

Meeting Structure and Union Alignment Requirements

```
Symptom:    Your structure or union data is not laid out
            as expected.
Cause:      Unexpected padding or alignment of structure/union.

Solution:   Rewrite programs to avoid padding and
            alignment assumptions.
            Lint will not catch this problem.
```

AViiON systems running DG/UX conform to the structure and union alignment requirements of the 88Open Object Compatibility Standard.

Meeting Integer Alignment Requirements

```
Symptom:    Integer fetch is off by one byte; for example,
            read fetches a wrong value from memory or
            store writes over wrong byte in memory.
Cause:      Misalignment of integer.

Solution:   Rewrite code to use pointers that meet all
            alignment requirements.
            Lint will not catch this problem.
```

AViiON systems running DG/UX conform to the integer alignment requirements of the 88Open Object Compatibility Standard.

Using Signed/Unsigned Characters or Integers

```
Symptom:    Program loops continuously; literal comparisons fail.
Cause:      Program presumes unsigned integers and/or characters.

Solution:   Use the unsigned character compiler option or rewrite
            code to avoid these assumptions.
            lint will catch some instances of this problem.
```

A common data format problem concerns signed versus unsigned characters or integers. The AViiON family architecture presumes signed characters. In particular, when a character is compared to an integer, the character's sign bit is extended. This means that a negative one (-1) stored in a character variable will match a negative one (-1) integer.

You can specifically request that all characters in a program be unsigned with a command line compiler option. See **cc**(1). Use the keyword **signed** in the character declaration to identify specific characters that are to be signed. You can also rewrite the offending code to use integer variables instead of character variables.

Passing Structures and Unions

Symptom: *Memory fault* error at runtime; unrealistic or unexpected values found in arguments.

Cause: Passing a structure or union variable to a routine that expects a true integer or pointer variable, or vice versa.

Solution: Pass a temporary variable of the appropriate type as the argument to the routine. Lint will catch this problem.

If a routine is expecting a structure or union as an argument and is passed an integer or pointer, it will get a *memory fault*. Conversely, if a routine is expecting an integer or pointer argument and receives a structure or union argument, a random integer value will be received.

The solution to these mismatches is to pass a temporary variable as the argument. Thus, if the routine is expecting an integer, store the desired structure field into a temporary integer variable and pass the temporary variable as the argument.

Remember, structures and unions may hold pointer or integer *values*, but this does not mean that they themselves are integer- or pointer-type variables.

Using Floating-point Format

The AViiON family floating-point format follows the IEEE-754 standard. Other systems, including the Data General ECLIPSE® system, use the IBM standard. If you find unexpected floating-point values when transferring data from another system, floating-point format may be the problem. The target system may handle floating-point numbers differently from the source system.

One way to handle the problem is to convert your floating-point values into some form that will not be altered in transportation, to transport them, and then to convert them to floating-point on the target machine; this will invoke the floating-point format on the destination machine. For example, convert the number into a character string, move the character string to the destination machine, and translate back from a character string to floating-point format. Converting IBM floating-point format to IEEE format results in a loss of precision but an increase in range.

NOTE: The range and precision of different floating-point formats varies; this might cause unavoidable loss of precision. Floating point is implemented using IEEE format. It is supported by an integrated on chip floating-point unit which greatly enhances floating-point performance.

Handling Memory Layout Issues

When you run a program, the kernel allocates an address space for the program and any data storage it may require. Within this address space, areas are reserved for various uses, as follows:

- Data Area — This area holds statically allocated variables.
- Text Area — This area holds the actual program code.
- Unallocated Space — This space is reserved for data storage that is allocated dynamically during runtime. There are three areas needed within the unallocated space:
    - Stack — The system uses the stack to hold automatic variables, the state of each function call, and arguments being passed to routines. On the DG/UX system, the stack is managed by system runtime routines and not your program. On AViiON systems the default stack starts at address 0xF0000000 and grows downward.
    - Heap — The heap is used for dynamic storage requests that your program makes during runtime. Calls to the **malloc(3C)** runtime routine return space taken from the heap. On the DG/UX system, the heap is managed by system runtime routines and not your program. On AViiON systems the default heap starts at the first four megabyte offset from the end of the text and data sections and grows upward. For most programs, this location is address

0x400000.

- General Storage Area — This area is reserved for any shared or unshared dynamic allocation requests that your program may need to make. The default location for this area is in the middle between the stack and heap. Unlike the other unallocated space areas, your program directly controls this storage area through system calls. You can create two different types of areas within this general storage area. These are:

    - Unshared Memory (sbrk/brk) — Your program uses the **sbrk**(2) and **brk**(2) system calls to allocate/deallocate space in this area.

    - Shared Memory — Your program uses the **shmat**(2) and **shmdt**(2) system calls to attach and detach space in this area.

The size of the stack and the heap change during runtime as information is added or space is allocated or deallocated.

Different machines may arrange these areas differently in memory. Figure 9-1 shows the *default* memory configuration used on the DG/UX system.

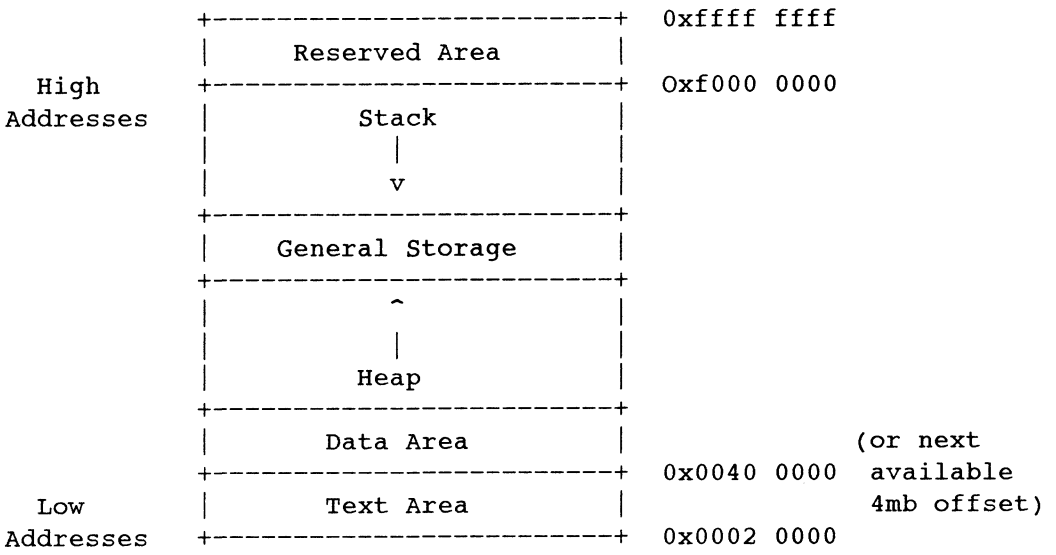


Figure 9-1 Program Layout in Memory

There are two main porting issues that may arise in regard to memory use:

- **Differences in the way the unallocated area is managed and allocated** — If your program does not have the required amount of heap, stack, or general

storage area, it may generate a fatal memory fault during execution. Database management programs that perform sophisticated memory operations may be particularly prone to this problem. On different systems, the way you reserve address space for each area will differ. The "Reserving and Using Dynamic Storage" section below discusses this problem in more detail.

- **Making assumptions about how the stack is implemented** — On AViiON family systems the stack grows downward (from higher addresses to lower addresses). On MV/Family machines, the stack grows upward (lower to higher). Many programs use assumptions about which way the stack grows for various purposes. For example, a program may try to determine whether a function call has been made by trying to check for arguments pushed on the stack.

  Assumptions about the order in which arguments are pushed onto the stack or about the direction of stack growth may not be valid on DG/UX systems.

Reserving and Using Dynamic Storage

Symptom: A *runtime initialization error* returned from a runtime library routine, a negative one (-1) returned from **sbrk** call, or a NULL returned from a **malloc** or **calloc** call.

Cause: Insufficient memory reserved for either stack, heap, or unshared memory area.

Solution: Increase reserved memory area through compiler and linker options. Alternatively, in the case of **sbrk**, you may need to change the location of your shared memory segment.
lint will not catch this problem.

If your program does not have enough space reserved for the heap or stack, the runtime library routines that handle initialization may return a *runtime initialization error* before ever giving control to your program. To correct the problem, you must increase the amount of space reserved for these structures at link time. Use a combination of runtime library definitions and linker options to control the allocation of the stack, heap and general storage area. See the *Programmer's Reference for the DG/UX System* for details on how to use these definitions.

Another problem with dynamic storage may occur if you make an unshared memory request (**sbrk**) that overlaps into a shared memory area. The kernel allocates shared memory at higher addresses than unshared memory in the general storage area. The lowest attached shared memory segment forms an upper boundary for the growth of unshared memory requests. Figure 9-2 illustrates this situation.

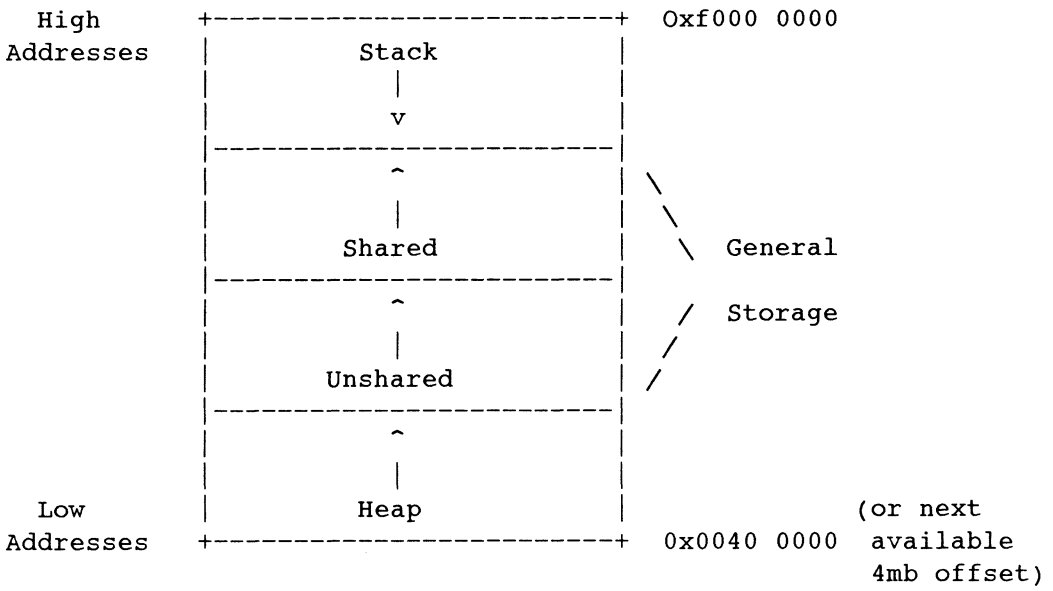


Figure 9-2 Shared and Unshared Memory Segments In The General Storage Area

If one of your **sbrk** calls (unshared memory request) returns a negative one (-1) (which indicates insufficient memory), it is quite possible that your unshared memory has grown into the space reserved for shared memory. The best way to handle this situation is to make sure that your shared memory area is as close as possible to the text area. This will reduce any waste space and provide more room for unshared memory within your general storage area. The easiest way to reserve space for unshared memory in general storage is to call **sbrk** before attaching (**shmat**) your first shared memory segment. The call to **sbrk** should cover an area large enough for all your program's unshared memory needs. Then attach the shared memory segment and call **sbrk** again to release the space that was reserved by the first **sbrk** call. By initially reserving unshared memory directly above the heap, you force the shared memory to be attached at an address closer to the text area. Because you have anticipated all your future unshared memory needs, you should not have to worry about overlapping the shared storage area.

Retrieving Arguments from the Stack

```
Symptom:    Invalid argument values found.
Cause:      Program that presumes a particular layout
            of arguments on the stack or one that presumes
            that the stack grows toward increasing addresses.

Solution:   Rewrite code using standard C functions.
            Lint may not catch this problem.
```

Programs that attempt to retrieve arguments pushed on the stack during a subroutine call should be rewritten to use standard language facilities without stack traversal. An example of improper argument retrieval code is as follows:

```
void f(a) int a;{
    int *ap = &a, i;
    while( i=*ap++ ){
        ...
    }
}
```

In this example, **f** is a function that can be invoked with a variable number of integer arguments. The code shown above tries to locate the address of the first argument and then increase that address by one in order to find the address of the next argument. Thus, the code is written to expect a certain layout of arguments on the stack. This logic will not work on the DG/UX system.

You can use the *varargs* C facility for a variable length argument list. The **vprintf**(3S) and **vscanf**(3S) families of functions are also useful for dealing with variable numbers of arguments during input and output operations. See the *Programmer's Reference for the DG/UX System (Volume 2)* for details on these facilities.

Programming In A Multi-Processor Environment

The AViiON computer family includes several platforms which support multiple 88000 processors. The DG/UX operating system includes several extensions to effectively support these multi-processor system architectures in a fully symmetric fashion. The user's view is of a single system, independent of the number of processors actually running. Because of this support, most user applications can run unmodified across the product line and benefit from the added performance of the multi-processor systems.

Some applications which may encounter problems when running on true multi-processor architectures are those which run as several processes which share data via either shared memory or shared data files. Database applications are likely to use these techniques. For these applications the ability to effectively run on a multi-processor system is greatly dependent on the program's design for handling resource contention and avoiding deadlock. Designs which include fixed sleep times to wait for an unavailable resource or retain locks for unnecessarily long periods of time may have problems. In some cases, this can cause applications to actually take longer to execute on a multi-processor system than a uni-processor system. This situation arises when resource contention causes the processes of an application to unnecessarily wait for resources to free.

If you have an application that experiences delay problems on a multi-processor system then you will need to examine your resource sharing algorithms and alleviate unnecessary waits.

End of Chapter

Chapter 10
Porting Graphical Applications

This chapter discusses the DG/UX 4.10 graphics hardware and software. After reading this chapter the reader should understand the graphics hardware and software supported, and associated porting issues. Familiarity with the X Window System, hereafter called X, is assumed. Please consult the X manuals available from Data General for a complete discussion of the architecture, usage, and programming of the X system.

This chapter covers six topics, and includes: a set of questions, reference section, and a glossary. The topics are: a description of the graphics environment, hardware issues, build and run environments, X issues and environment, xterminal issues, and issues that all applications must deal with that pertain to graphics.

Data General's graphical environment consists of the X Window System X11R3 from MIT, along with supporting hardware. Data General has committed to support the OSF Motif user interface. Due to the support of major standards, most applications (that use the standards) will have few problems in the port to the AViiON platforms.

The porting question remains: Given an application working on another host system, what needs to be done to get it running on AViiON hardware? The questions at the end of this chapter pertain to the graphical environment your application expects, they do not touch on general porting issues. A non-graphical application need only answer the last question.

ALL GENERAL OPERATING SYSTEM AND LANGUAGE PORTING ISSUES STILL APPLY TO GRAPHICAL APPLICATIONS!

The style guide that Motif specifies has been committed to by Data General. This specification is still in the draft stage. The Athena widget set may be used in the interim; however there will be changes required of applications when Motif becomes available.

Another standard that is still in draft form is the "Proposed Interclient Communication Conventions" appendix of the *Xlib Programming Manual*. This is aimed at enabling X clients to be ported between different window managers.

The color version of the AViiON workstation is not available at the present. The issues involved with this are a different depth of the bitplane, a different X server implementation, and a different graphics controller chip.

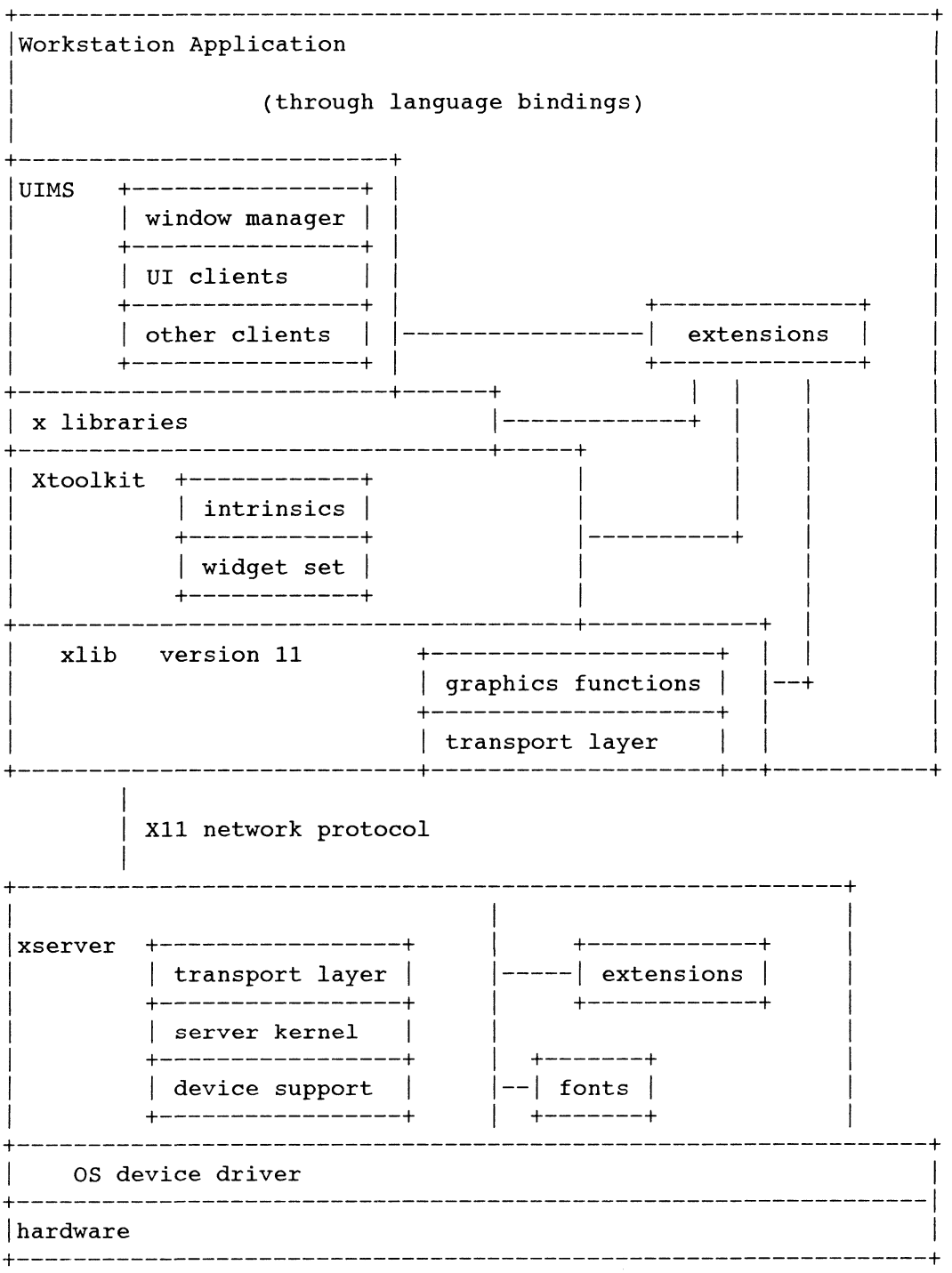


Figure 10-1 Conceptualized Environment Architecture

Controlling the Graphical Software Environment

Data General's graphical software environment consists of language and operating system bindings, the X Window System from MIT, and a commitment to the Motif style guide.

Currently "C" and DG/UX are the only language and operating system bindings available for X applications on AViiON systems. See the other chapters of this manual for assistance in these areas of porting.

A complete X Window System is provided, including the server, library, toolkit intrinsics, Athena widget set, and nine of the most useful clients (uwm, xbiff, xclock, xload, xwd, xwud, xpr, xstart, and xterm). The contributed software sources that are supplied on the X release tapes from MIT will be passed on by Data General, with no implied support or warranty. See the X Environment and Issues section for more information on these areas.

Controlling the Graphical Hardware Environment

Applications written to the controlling chip level are highly unportable. It is strongly suggested that applications not use this level. The preferred method is to convert the application to make requests of the X server. This will greatly enhance the application's portability. See the AViiON hardware documentation for a description of the graphical controller chip for both the monochrome and color configurations. See Data General and vendor documentation for specifics on the controller chip in the x-terminal.

The porting issues concerning cpu, standard devices, and networking are covered in other sections of this document. Issues involving graphical hardware, either workstation or x-terminal, consist of the characteristics of the controlling chip, the supported devices (screen, keyboard, mouse), and, supporting new devices. There are two distinct X server implementations, one for mono and one for color. The X-terminal also implements the X server internally.

An application that uses only the supported devices through X should have few, if any, hardware porting issues; on either the workstation or the x-terminal. See Table 5.1 in Chapter 5 for characteristics of the supported graphics devices and x-terminal characteristics. See the appropriate sections of the X manuals for directions on adding support for new devices. See *Writing a Device Driver for the DG/UX System* for information on how to add device support to the operating system. The AViiON workstation also provides an async port, SCSI port, and printer port. These do not impact graphical applications.

Controlling the Build and Run Environment

There are few issues specific to graphics in the build and run environment. If dealing with an X application, the placement of the graphical libraries and includes should be the only major issues. Data General provides the X libraries and includes from MIT, in the standard places, refer to the X manuals for contents and placement. See also the X release notes from Data General. See the X documentation for build procedures for the X system applications. See chapters 6 through 8 and chapter 4 for general build and run issues.

You should be aware that the majority of software sources that are provided in the /contrib portion of the X release from MIT consist of programs that Data General HAS NOT VERIFIED. There is no guarantee that the sources or makefile will build or run correctly. Many of these applications were developed on platforms which differ greatly from the AViiON architecture and environment.

If you are porting from another graphical package, consult the X documentation for porting hints.

Porting to the X Window System

Data General's graphical environment consists of the X Window System from MIT and a separate library of primitive functions, along with supporting hardware. No change in functionality from that described in the X documentation should exist. See the X manuals for porting and internationalization issues. Note that if you have used a toolkit other than the MIT release, your port will require rewriting some of your code. If you use a User Interface Management System that is X based, you should be able to port your UIMS with a simple recompile.

Using Software Development Tools

The standard DG/UX development tools are available, see chapter 6 for more information on them.

Handling X Terminal Issues

Since an X terminal provides display hardware and the X server software, the issues identified above in the X environment section pertain. Xlib and the X client application will still reside and run on the host AViiON system. A working X application should run with no changes on an X terminal. The size of the screen is the only allowable difference between the X terminal and the AViiON workstation that an application may discern. See the *X-terminal Installation and Operation Manual* for more details on functionality and characteristics. Typically x-terminals have a smaller amount of memory in them than would be present on a workstation. This may lead to applications hanging and/or crashing if the memory resource is

expended. X terminals (and diskless workstations) will also result in greatly increased network traffic.

Graphical Issues for All Applications

The only difference in environment that a non-graphical application must accommodate is being spawned by an xterm. The only issues stemming from this difference are: standard I/O goes through an emulation of Digital vt102 or Tektronix 4014 terminals, and the application must be able to handle the window being resized. A window resize event is handed to the application by the SIGWINCH signal. See the X manuals for window resizing implications. Font sizes also interact with the window size, and must be considered. See chapters 4 and 5 for more details on setup and terminal issues. This assumes that your application does not by-pass the termcap/terminfo interface.

Determining the Current Environment

The following checklist may be used to determine the extent of the porting effort needed to convert your application to the Data General environment. The effort is proportional to the amount of differences between the environments. Note that this set of questions address graphical issues only. The questions below apply to all applications that may be executed in an **xterm** window, whether or not they are explicitly graphical. The item in square brackets is the Data General offering.

a) What environment(s) do you expect?

- Windowing environment [X Window System]

b) What hardware do you expect?

- Graphics monitor or workstation characteristics [display, keyboard, mouse]
- Any non-monitor devices [none]
- Do you have any cpu or network dependencies [TCP/IP, NFS]

c) What are your build and run environments?

- Libraries & include files [standard UNIX, C, and X]

d) Do you use the X environment, and if so;

- Hardware dependencies assumed [display, keyboard, mouse]
- What environment is expected .
  Library calls [C and X]

Client / Server functions [standard X]
Window Manager expected [uwm]
Toolkit calls [standard Xt intrinsics, and Athena widget set]
Style guide [Motif - committed to]

- What development tools are expected
- Do you use any extensions to X [none are currently approved by MIT]

e) Does the non-graphical portion of your application?

- Use termcap or terminfo/curses [DGC provides both]
- Handle the SIGWINCH signal

End of Chapter

Chapter 11
Building and Running BCS Applications

This chapter describes how to build and run BCS-compliant programs. The material in this chapter is not yet final.

End of Chapter

Appendix A
DG/UX System Calls and Commands

This appendix lists the system calls and commands supported by the DG/UX system.

System Calls

Table A-1 summarizes the system calls available in the DG/UX system and other UNIX systems. The table uses the following notation conventions:

4.1	New in DG/UX Release 4.10
x	Included in the system named by the column head
-	Not included in the system named by the column head
L	Implemented as a library routine
BA_OS	Part of SVID Base OS function
KE_OS	Part of SVID Kernel Extension OS function
88C	Performed completely by calling a BCS gate
88L	Performed by a library routine that calls one or more BCS gates but provides additional logic to complete the function
88X	Performed by invoking **exec**(2) on a user-level program
88?	Only partially supportable via BCS

Table A-1 Summary of System Calls

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
accept	x	-	-	x	-	-	-
access	x	x	x	x	88C	x	BA_OS
acct	x	x	x	x	88X	-	KE_OS
adjtime	-	-	-	x	-	-	-
alarm	x	x	x	-	88L	x	BA_OS
async_daemon	x	-	-	S	-	-	-
audit	-	-	-	S	-	-	-
auditon	-	-	-	S	-	-	-
auditsvc	-	-	-	S	-	-	-
berk_sigpause	x	-	-	x	88L	-	-
bind	x	-	-	x	-	-	-

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
brk	x	x	x	x	88C	-	-
cfgetispeed	-	-	-	-	-	x	-
cfgetospeed	-	-	-	-	-	x	-
cfsetispeed	-	-	-	-	-	x	-
cfsetospeed	-	-	-	-	-	x	-
chdir	x	x	x	x	88C	x	BA_OS
chmod	x	x	x	x	88C	x	BA_OS
chown	x	x	x	x	88C	x	BA_OS
chroot	x	x	x	x	88C	-	KE_OS
close	x	x	x	x	88C	x	BA_OS
closedir	-	-	-	-	-	x	-
connect	x	-	-	x	-	-	-
creat	x	x	x	x	88L	x	BA_OS
ctermid	-	-	-	-	-	x	-
cuserid	-	-	-	-	-	x	-
dg_ext_errno	x	-	-	-	-	-	-
dg_file_info	x	-	-	-	-	-	-
dg_fstat	4.1	-	-	-	-	-	-
dg_ipc_info	x	-	-	-	-	-	-
dg_mknod	4.1	-	-	-	-	-	-
dg_mount	4.1	-	-	S	-	-	-
dg_mstat	x	-	-	-	-	-	-
dg_process_info	x	-	-	-	-	-	-
dg_stat	4.1	-	-	-	-	-	-
dg_sys_info	x	-	-	-	-	-	-
dg_xtrace(2)*	x	-	-	-	-	-	-
dup	x	x	x	x	88L	x	BA_OS
dup2	x	x	x	x	88L	x	BA_OS
execl	L	x	x	-	88L	x	BA_OS
execle	L	x	x	-	88L	x	BA_OS
execlp	L	x	x	-	88L	x	BA_OS
execv	L	x	x	-	88L	x	BA_OS
execve	x	x	x	x	-	x	BA_OS
execvp	L	x	x	-	88L	x	BA_OS
_exit	x	x	x	x	88C	x	BA_OS
exit	L	x	x	-	88L	-	BA_OS
exportfs	4.1	-	-	S	-	-	-
fchmod	x	-	-	x	-	-	-
fchown	x	-	-	x	-	-	-
fcntl	x	x	x	x	88C	x	BA_OS
fdopen	-	-	-	-	-	x	-
fileno	-	-	-	-	-	x	-
flock	L	-	-	x	-	-	-
fork	x	x	x	x	88C	x	BA_OS
fpathconf	4.1	-	-	-	88C	x	-
fstat	x	x	x	x	88C	x	BA_OS
fstatfs	x	-	x	-	-	-	-

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
fsync	x	-	-	x	-	-	-
ftruncate	x	-	-	x	88L	-	-
getauid	-	-	-	S	-	-	-
getcwd	-	-	-	-	-	x	-
getdents	x	x	x	S	88C	-	-
getdirentries	x	-	-	S	-	-	-
getdomainname	x	-	-	S	-	-	-
getdtablesize	x	-	-	x	88L	-	-
getegid	x	x	x	x	88L	x	BA_OS
getenv	-	-	-	-	-	x	-
geteuid	x	x	x	x	88L	x	BA_OS
getgrgid	-	-	-	-	-	x	-
getgrnam	-	-	-	-	-	x	-
getfh	4.1	-	-	-	-	-	-
getgid	x	x	x	x	88C	x	BA_OS
getgroups	x	-	-	x	88C	x	-
gethostid	x	-	-	x	-	-	-
gethostname	x	-	-	x	-	-	-
getitimer	x	-	-	x	88C	-	-
getlogin	-	-	-	-	-	x	-
getmsg	4.1	x	x	S	88C	-	-
getpagesize	x	-	-	x	88?	-	-
getpeername	x	-	-	x	-	-	-
getpgrp	x	x	x	-	88C	x	BA_OS
getpgrp2	4.1	-	-	x	-	-	-
getpid	x	x	x	x	88C	x	BA_OS
getppid	x	x	x	x	88L	x	BA_OS
getpriority	x	-	-	x	-	-	-
getpsr	4.1	-	-	-	88C	-	-
getpwnam	-	-	-	-	-	x	-
getpwuid	-	-	-	-	-	x	-
getrlimit	x	-	-	x	-	-	-
getrusage	x	-	-	x	-	-	-
getsockname	x	-	-	x	-	-	-
getsockopt	x	-	-	x	-	-	-
gettimeofday	x	-	-	x	-	-	-
getuid	x	x	x	x	88C	x	BA_OS
ioctl	x	x	x	x	88C	-	BA_OS
isatty	-	-	-	-	-	x	-
kill	x	x	x	x	88C	x	BA_OS
killpg	x	-	-	x	88L	-	-
link	x	x	x	x	88C	x	BA_OS
listen	x	-	-	x	-	-	-
lockf	L	L	x	-	88L	-	BA_OS
lseek	x	x	x	x	88C	x	BA_OS
lstat	x	-	-	x	-	-	-
memctl	4.1	-	-	-	88C	-	-

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
mincore	-	-	-	S	-	-	-
mkdir	x	x	x	x	88C	x	BA_OS
mkfifo	-	-	-	-	-	x	-
mknod	x	x	x	x	88C	-	BA_OS
mmap	4.1	-	-	S	-	-	-
mount	x	x	x	x	88X	-	BA_OS
msgctl	x	x	x	S	88L	-	KE_OS
msgget	x	x	x	S	88L	-	KE_OS
msgrcv	x	x	x	S	88L	-	KE_OS
msgsnd	x	x	x	S	88L	-	KE_OS
msync	-	-	-	S	-	-	-
munmap	4.1	-	-	S	-	-	-
nfsmount	x	-	-	-	-	-	-
nfssvc	x	-	-	S	-	-	-
nice	x	x	x	-	88C	-	KE_OS
open	x	x	x	x	88C	x	BA_OS
opendir	-	-	-	-	-	x	-
pathconf	4.1	-	-	-	88C	x	-
pause	x	x	x	-	88L	x	BA_OS
pipe	x	x	x	x	88C	x	BA_OS
plock	x	x	x	-	88C	-	KE_OS
poll	4.1	x	x	S	88C	-	-
profil	x	x	x	x	88C	-	KE_OS
ptrace	x	x	x	x	88C	-	KE_OS
putmsg	4.1	x	x	S	88C	-	-
quota	-	-	-	B	-	-	-
quotactl	-	-	-	S	-	-	-
read	x	x	x	x	88C	x	BA_OS
readdir	-	-	-	-	-	x	-
readlink	x	-	-	x	-	-	-
readv	x	-	-	x	-	-	-
reboot	x	-	-	x	-	-	-
recv	x	-	-	x	-	-	-
recvfrom	x	-	-	x	-	-	-
recvmsg	x	-	-	x	-	-	-
rename	x	-	-	x	88C	x	-
rewinddir	-	-	-	-	-	x	-
rmdir	x	x	x	x	88C	x	BA_OS
sbrk	x	-	x	x	88?	-	-
select	x	-	-	x	-	-	-
semctl	x	x	x	S	88L	-	KE_OS
semget	x	x	x	S	88L	-	KE_OS
semop	x	x	x	S	88L	-	KE_OS
send	x	-	-	x	-	-	-
sendmsg	x	-	-	x	-	-	-
sendto	x	-	-	x	-	-	-
setaudit	-	-	-	S	-	-	-

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
setauid	-	-	-	S	-	-	-
setdomainname	x	-	-	S	-	-	-
setgid	x	x	x	B	88C	x	BA_OS
setgroups	x	-	-	x	-	-	-
sethostid	x	-	-	B	-	-	-
sethostname	x	-	-	x	-	-	-
setitimer	x	-	-	x	88C	-	-
setlocale	-	-	-	-	-	x	-
setpgid	-	-	-	-	-	x	-
setpgrp	x	x	x	S	88?	-	BA_OS
setpgrp2	4.1	-	-	x	-	-	-
setpriority	x	-	-	x	-	-	-
setpsr	4.1	-	-	-	88C	-	-
setquota	-	-	-	B	-	-	-
setregid	x	-	-	x	-	-	-
setreuid	x	-	-	x	-	-	-
setrlimit	x	-	-	x	-	-	-
setsid	4.1	-	-	-	88C	x	-
setsockopt	x	-	-	x	-	-	-
settimeofday	x	-	-	x	-	-	-
setuid	x	x	x	B	88C	x	BA_OS
setuseraudit	-	-	-	S	-	-	-
shmat	x	x	x	S	88L	-	KE_OS
shmctl	x	x	x	S	88L	-	KE_OS
shmdt	x	x	x	S	88L	-	KE_OS
shmget	x	x	x	S	88L	-	KE_OS
shutdown	x	-	-	x	-	-	-
sigaction	4.1	-	-	-	88C	x	-
sigaddset	-	-	-	-	-	x	-
sigblock	x	-	-	x	88L	-	-
sigdelset	-	-	-	-	-	x	-
sigemptyset	-	-	-	-	-	x	-
sigfillset	4.1	-	-	-	88C	x	-
sighold	x	x	x	-	88L	-	BA_OS
sigignore	x	x	x	-	88L	-	BA_OS
sigismember	-	-	-	-	-	x	-
signal	x	x	x	-	88L	-	BA_OS
siglongjmp	-	-	-	-	-	x	-
sigpause	x	-	x	-	88L	-	-
sigpending	4.1	-	-	-	88C	x	-
sigprocmask	4.1	-	-	-	88C	x	-
sigrelse	x	x	x	-	88L	-	BA_OS
sigret	4.1	-	-	-	88C	-	-
sigreturn	-	-	-	B	-	-	-
sigset	x	x	x	-	88L	-	BA_OS
sigsetjmp	-	-	-	-	-	x	-
sigsetmask	x	-	-	x	88L	-	-

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
sigsetops	-	-	-	-	-	x	-
sigstack	x	-	-	x	-	-	-
sigsuspend	4.1	-	-	-	88C	x	-
sigvec	x	-	-	x	88?	-	-
sleep	-	-	-	-	-	x	-
socket	x	-	-	x	-	-	-
socketpair	x	-	-	x	-	-	-
stat	x	x	x	x	88C	x	BA_OS
statfs	x	x	x	S	-	-	-
stime	x	x	x	-	-	-	BA_OS
swapon	x	-	-	x	-	-	-
symlink	x	-	-	x	-	-	-
sync	x	x	x	x	88C	-	BA_OS
sys_local	4.1	-	-	-	88C	-	-
syscall	-	-	-	x	-	-	-
sysconf	4.1	-	-	-	88C	x	-
sysfs	-	x	x	-	-	-	-
tcdrain	-	-	-	-	-	x	-
tcflow	-	-	-	-	-	x	-
tcflush	-	-	-	-	-	x	-
tcgetattr	-	-	-	-	-	x	-
tcgetpgrp	-	-	-	-	-	x	-
tcsendbreak	-	-	-	-	-	x	-
tcsetattr	-	-	-	-	-	x	-
tcsetpgrp	-	-	-	-	-	x	-
tell	-	-	-	S	-	-	-
time	x	x	x	-	88C	x	BA_OS
times	x	x	x	-	88C	x	BA_OS
truncate	x	-	-	x	88L	-	-
ttyname	-	-	-	-	-	x	-
tzset	-	-	-	-	-	x	-
uadmin	-	x	x	-	-	-	-
ulimit	x	x	x	-	88C	-	BA_OS
umask	x	x	x	x	88C	x	BA_OS
umount	x	x	x	B	88X	-	BA_OS
uname	x	x	x	S	88C	x	BA_OS
unlink	x	x	x	x	88C	x	BA_OS
unmount	-	-	-	S	-	-	-
ustat	x	x	x	-	88C	-	BA_OS
utime	x	x	x	-	88C	x	BA_OS
utimes	x	-	-	x	88L	-	-
vadvise	-	-	-	S	-	-	-
vfork	x	-	-	x	-	-	-
vhangup	x	-	-	x	-	-	-
wait	x	x	x	x	88L	-	BA_OS
wait3	x	-	-	x	88?	-	-
wait4	-	-	-	S	-	-	-

System Call	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD/Sun 4.3/4.0	BCS 10	POSIX	SVID 2
waitpid	4.1	-	-	-	88C	x	-
write	x	x	x	x	88C	x	BA_OS
writev	x	-	-	x	-	-	-

* Formerly known as **xtrace**(2).

User and Programmer Commands

Table A-2 summarizes the user and programmer commands available in the DG/UX, 386/ix, AT&T V.3, and BSD 4.2 operating systems and whether they comply with SVID. There is no column for BCS, since it does not specify conformance for commands. Since the POSIX standard for commands has not yet been approved, no column is given for POSIX.

The table uses the following notation conventions:

x	Included in the system named by the column head
-	Not included in the system named by the column head
4.0	Included in DG/UX 4.0 but not in 4.10
4.1	Included in DG/UX 4.10 but not in 4.0
BU_CMD	Part of SVID Basic Utilities Extension
AU_CMD	Part of SVID Advanced Utilities Extension
AS_CMD	Part of SVID Administered Systems Extension
SD_CMD	Part of SVID Software Development Extension
TI_CMD	Part of SVID Terminal Interface Extension
NS_CMD	Part of SVID Network Services Extension

Table A-2 Summary of User and Programmer Commands

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
300	-	x	x	-	-
4014	-	x	x	-	-
450	-	x	x	-	-
acctcom	x	x	x	-	-
admin	x	x	x	x	SD_CMD
addrem	-	x	-	-	-
ar	x	x	x	x	BU_CMD
as	x	x	x	x	SD_CMD
as386.sed	-	x	-	-	-
asa	x	-	x	-	-
assist	x	-	x	-	-
astgen	x	-	-	-	-
at	x	x	x	x	AU_CMD
att_dump	x	-	-	-	-
att_stty	x	-	-	-	-
awk	x	x	x	x	BU_CMD
banner	x	x	x	-	BU_CMD
basename	x	x	x	x	BU_CMD
batch	x	x	x	-	AU_CMD
bc	x	x	x	x	-
bdiff	x	x	x	-	-
berk_diff	x	-	-	-	-
berk_diff3	x	-	-	-	-

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
berk_stty	x	-	-	-	-
bfs	x	x	x	-	-
cal	x	x	x	x	BU_CMD
calendar	4.1	x	x	x	BU_CMD
cancel	x	x	x	-	AU_CMD
cat	x	x	x	x	BU_CMD
cb	x	x	x	x	-
cc	x	x	x	x	SD_CMD
ccoff	-	x	-	-	-
cd	x	x	x	x	BU_CMD
cdc	x	x	x	-	-
cflow	x	x	x	-	SD_CMD
chgrp	x	x	x	x	AU_CMD
chmod	x	x	x	x	BU_CMD
chown	x	x	x	-	AU_CMD
ci	4.1	-	-	-	-
clear	x	-	-	x	-
cmp	x	x	x	x	BU_CMD
co	4.1	-	-	-	-
col	x	x	x	x	BU_CMD
comb	x	x	x	-	-
comm	x	x	x	x	BU_CMD
conv	-	x	x	-	-
convert	-	x	x	-	-
cp	x	x	x	x	BU_CMD
cpio	x	x	x	-	BU_CMD
cpp	x	x	x	-	SD_CMD
cprs	-	x	x	-	-
crontab	x	x	x	-	AU_CMD
crypt	x	x	x	x	-
csh	x	x	-	x	-
csplit	x	x	x	-	AU_CMD
ct	x	x	x	-	-
ctags	x	-	-	x	-
ctrace	x	x	x	-	-
cu	x	x	x	x	AU_CMD
cut	x	x	x	-	BU_CMD
cxref	x	x	x	-	SD_CMD
date	x	x	x	x	BU_CMD
dbx	x	-	-	x	-
dc	x	x	x	x	-
dd	x	x	x	x	AU_CMD
deblock	x	-	-	-	-
delta	x	x	x	-	SD_CMD
deroff	x	x	x	-	-
dghost	x	-	-	-	-
diff	x	x	x	x	BU_CMD

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
diff3	x	x	x	x	-
dircmp	x	x	x	-	AU_CMD
dirname	x	x	x	-	BU_CMD
dis	x	x	x	-	SD_CMD
disable	x	x	x	-	-
dossette	-	x	-	-	-
du	x	x	x	x	BU_CMD
e	-	x	-	-	-
echo	x	x	x	x	BU_CMD
ed	x	x	x	x	BU_CMD
edit	x	x	x	x	-
egrep	x	x	x	x	AU_CMD
enable	x	x	x	-	-
env	x	x	x	-	SD_CMD
erase	-	x	-	-	-
ex	x	x	x	x	AU_CMD
expr	x	x	x	x	BU_CMD
factor	x	x	x	-	-
false	x	x	x	x	BU_CMD
ffil	-	x	-	-	-
fgrep	x	x	x	x	AU_CMD
file	x	x	x	x	BU_CMD
find	x	x	x	x	BU_CMD
fold	x	-	-	x	-
fsplit	x	-	x	x	-
gdev	-	x	-	-	-
ged	-	x	x	-	-
get	x	x	x	-	SD_CMD
getopt	x	x	x	-	-
getoptcvt	x	x	x	-	-
getopts	x	x	x	-	-
ghost	-	x	-	-	-
glossary	x	x	x	-	-
graphics	-	x	x	-	-
greek	-	x	x	-	-
grep	x	x	x	x	BU_CMD
gutil	-	x	-	-	-
hardcopy	-	x	-	-	-
hashcheck	x	x	x	-	-
hashmake	x	x	x	-	-
head	x	-	-	x	-
help	x	x	x	-	-
history	-	x	-	-	-
hp	-	x	x	-	-
hpd	-	x	-	-	-
hpio	-	x	x	-	-
i286emul	-	x	-	-	-

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
id	x	x	x	-	AU_CMD
ident	x	-	-	-	-
infocmp	x	x	-	-	-
insdriver	-	x	-	-	-
inskern	-	x	-	-	-
ipcrm	x	x	x	-	AS_CMD
ipcs	x	x	x	-	AS_CMD
ismpx	-	x	-	-	-
join	x	x	x	x	AU_CMD
jterm	-	x	-	-	-
jwin	-	x	-	-	-
kconfig	-	x	-	-	-
kill	x	x	x	x	BU_CMD
last	x	-	-	x	-
layers	-	x	-	-	-
ld	x	x	x	x	SD_CMD
lex	x	x	x	x	SD_CMD
lfd	x	-	-	-	-
line	x	x	x	-	BU_CMD
list	x	x	x	-	-
lint	x	x	x	x	SD_CMD
ln	x	x	x	x	BU_CMD
locate	x	x	x	-	-
login	x	x	x	x	-
logname	x	x	x	-	AU_CMD
lorder	x	x	x	x	SD_CMD
lp	x	x	x	-	AU_CMD
lpstat	x	x	x	-	AU_CMD
ls	x	x	x	x	BU_CMD
m4	x	x	x	x	SD_CMD
m68k	4.1	-	x	-	-
machid	x	x	x	-	-
mail	x	x	x	x	BU_CMD
mailx	x	x	x	-	AU_CMD
make	x	x	x	x	SD_CMD
man	x	x	x	x	-
mcs	-	x	-	-	-
merge	x	-	-	-	-
mesg	x	x	x	x	AU_CMD
mkdir	x	x	x	x	BU_CMD
mkshlib	-	x	-	-	-
mkstr	x	-	-	x	-
more	x	-	-	x	-
mv	x	x	x	x	BU_CMD
mxdb	x	-	-	-	-
newfile	-	x	-	-	-
newform	x	x	x	-	-

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
newgrp	x	x	x	x	AU_CMD
news	x	x	x	-	AU_CMD
nice	x	x	x	x	AS_CMD
nl	x	x	x	-	BU_CMD
nm	x	x	x	x	SD_CMD
nohup	x	x	x	x	BU_CMD
od	x	x	x	x	AU_CMD
pack	x	x	x	-	BU_CMD
page	x	-	-	x	-
passwd	x	x	x	x	AU_CMD
paste	x	x	x	-	-
pcat	x	x	x	-	-
pdp11	4.1	-	x	-	-
pc	x	-	-	x	-
pg	x	x	x	-	BU_CMD
pr	x	x	x	x	BU_CMD
printenv	x	-	-	x	-
prof	x	x	x	x	SD_CMD
prs	x	x	x	-	SD_CMD
prtty	-	x	-	-	-
ps	x	x	x	x	BU_CMD
pwd	x	x	x	x	BU_CMD
ratfor	x	-	x	x	-
rcs	4.1	-	-	-	-
rcsdiff	4.1	-	-	-	-
rcsintro	4.1	-	-	-	-
rcsmerge	4.1	-	-	-	-
readfile	-	x	-	-	-
red	x	x	x	-	BU_CMD
regcmp	x	x	x	-	-
reset	x	-	-	x	-
rev	x	-	-	x	-
revision	x	-	-	-	-
rlog	x	-	-	-	-
rlogin	x	-	-	x	-
rm	x	x	x	x	BU_CMD
rmail	4.1	x	x	x	BU_CMD
rmdel	x	x	x	-	SD_CMD
rmdir	x	x	x	x	BU_CMD
rmhist	-	x	-	-	-
rpl	-	x	-	-	-
rsh	x	x	x	-	BU_CMD
sact	x	x	x	-	SD_CMD
sadp	4.0	-	-	-	-
sag	-	x	-	-	-
sccsdiff	x	x	x	-	-
sccstorcs	4.1	-	-	-	-

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
sdb	x	x	x	x	SD_CMD
sdiff	x	x	x	-	-
sed	x	x	x	x	BU_CMD
setup	-	x	-	-	-
sh	x	x	x	x	BU_CMD
shl	x	x	x	-	AU_CMD
showrem	-	x	-	-	-
size	x	x	x	x	SD_CMD
sleep	x	x	x	x	BU_CMD
sno	x	-	x	-	-
sort	x	x	x	x	BU_CMD
spell	x	x	x	x	BU_CMD
spellin	x	-	x	x	-
spline	-	x	x	x	-
split	x	x	x	x	BU_CMD
stat	-	x	x	-	-
starter	x	x	x	-	-
strings	x	-	-	x	-
strip	x	x	x	x	SD_CMD
stty	x	x	x	x	AU_CMD
su	x	x	x	x	AU_CMD
sum	x	x	x	x	BU_CMD
tabs	x	x	x	x	AU_CMD
tail	x	x	x	x	BU_CMD
tar	x	x	x	x	AU_CMD
tconvert	-	x	-	-	-
td	-	x	-	-	-
tdigest	-	x	-	-	-
tee	x	x	x	x	BU_CMD
tekset	-	x	-	-	-
test	x	x	x	x	BU_CMD
time	x	x	x	x	SD_CMD
timex	x	x	x	-	AS_CMD
toc	-	x	x	-	-
touch	x	x	x	x	BU_CMD
tplot	-	x	x	-	-
tput	x	x	x	-	TI_CMD
tr	x	x	x	x	BU_CMD
true	x	x	x	x	BU_CMD
tsort	x	x	x	x	SD_CMD
tty	x	x	x	x	AU_CMD
u3b	4.1	-	x	-	-
u3b5	4.1	-	x	-	-
ul	x	-	-	x	-
umask	x	x	x	-	BU_CMD
uname	x	x	x	-	BU_CMD
unget	x	x	x	-	SD_CMD

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
uniq	x	x	x	x	BU_CMD
units	x	x	x	x	-
unpack	x	x	x	-	BU_CMD
usage	x	x	x	-	-
uucp	x	x	x	x	AU_CMD
uudecode	x	-	-	x	-
uuencode	x	-	-	x	-
uulog	x	x	x	x	AU_CMD
uuname	x	x	x	-	AU_CMD
uupick	x	x	x	-	AU_CMD
uustat	x	x	x	-	AU_CMD
uuto	x	x	x	-	AU_CMD
uux	x	x	x	x	AU_CMD
val	x	x	x	-	SD_CMD
vax	4.1	-	x	-	-
vc	x	x	x	-	-
vedit	x	x	x	-	-
versions	-	x	-	-	-
vi	x	x	x	x	AU_CMD
view	x	x	x	-	-
vpix	-	x	-	-	-
wait	x	x	x	x	BU_CMD
wc	x	x	x	x	BU_CMD
what	x	x	x	x	SD_CMD
whereis	x	-	-	x	-
who	x	x	x	x	AU_CMD
write	x	x	x	x	AU_CMD
xargs	x	x	x	-	SD_CMD
xstr	x	-	-	x	-
yacc	x	x	x	x	SD_CMD

Administrative Commands

Table A-3 summarizes the administrative commands available in the DG/UX, 386/ix, AT&T V.3, and BSD 4.2 operating systems and whether they comply with SVID. There is no column for BCS, since it does not specify conformance for commands. Since the POSIX standard for commands has not yet been approved, no column is given for POSIX.

The table uses the following notation conventions:

x	Included in the system named by the column head
-	Not included in the system named by the column head
4.0	Included in DG/UX 4.0 but not in 4.10
4.1	Included in DG/UX 4.10 but not in 4.0
BU_CMD	Part of SVID Basic Utilities Extension
AU_CMD	Part of SVID Advanced Utilities Extension
AS_CMD	Part of SVID Administered Systems Extension
SD_CMD	Part of SVID Software Development Extension
TI_CMD	Part of SVID Terminal Interface Extension
NS_CMD	Part of SVID Network Services Extension

Table A-3 Summary of Administrative Commands

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
accept	x	x	x	-	-
acct	x	x	x	-	-
acctcms	x	x	x	-	AS_CMD
acctcon	x	x	x	-	AS_CMD
acctcon1	x	x	x	-	AS_CMD
acctcon2	x	x	x	-	AS_CMD
acctdisk	x	x	x	-	AS_CMD
acctdusg	x	-	x	-	-
acctmerg	x	x	x	-	AS_CMD
accton	x	-	-	x	-
acctprc	x	x	x	-	AS_CMD
acctprc1	x	x	x	-	AS_CMD
acctprc2	x	x	x	-	AS_CMD
acctsh	x	x	x	-	-
acctwtmp	x	x	x	-	AS_CMD
bcheckrc	4.1	-	-	-	-
bdblk	x	-	x	-	-
brc	x	x	x	-	-
captoinfo	x	x	-	-	-
chargefee	x	x	x	-	AS_CMD
chroot	x	x	x	-	SD_CMD
ckbupscd	4.1	x	-	-	-
ckpacct	x	x	x	-	AS_CMD

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
clri	x	x	x	x	AS_CMD
config	x	x	x	x	-
crash	x	x	x	x	-
cron	x	x	x	x	AU_CMD
dcopy	-	x	x	-	-
devnm	x	x	x	-	AS_CMD
df	x	x	x	x	BU_CMD
diskadd	-	x	-	-	-
diskman	x	-	-	-	-
diskusg	x	x	x	-	AS_CMD
dmesg	x	-	-	x	-
dname	-	x	-	-	-
dodisk	x	x	x	-	AS_CMD
dump	x	-	-	x	-
dumpdir	x	-	-	-	-
dumpfs	x	-	-	x	-
ermes_editor	x	-	-	-	-
errdemon	x	-	x	-	-
errpt	x	-	x	-	-
errstop	x	-	x	-	-
fdisk	-	x	-	-	-
ff	x	-	x	-	-
filesave	x	-	x	-	-
finc	x	-	x	-	-
format	-	x	x	-	-
frec	x	x	x	-	-
fsck	x	x	x	x	AS_CMD
fsdb	x	x	x	-	AS_CMD
fsirand	x	-	-	-	-
fsstat	-	x	-	-	-
fstyp	-	x	-	-	-
ftpd	x	-	-	x	-
fumount	-	x	-	-	-
fusage	-	x	-	-	-
fuser	x	x	x	-	AS_CMD
fwtmp	x	x	x	-	AS_CMD
gencc	-	x	-	-	-
getty	x	x	x	x	-
grpck	x	x	x	-	AS_CMD
halt	x	-	-	x	-
helpadm	x	x	x	-	-
i552pump	-	x	-	-	-
iacload	x	-	-	-	-
ib	-	x	-	-	-
idload	-	x	-	-	-
ifconfig	x	-	-	x	-
init	x	x	x	x	AS_CMD

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
install	x	x	x	x	-
intro	x	-	x	x	-
killall	x	x	x	-	AS_CMD
labelit	x	x	x	-	AS_CMD
lastlogin	x	x	x	-	AS_CMD
lib_merge	x	-	-	-	-
link	x	x	x	-	AS_CMD
lpadmin	x	x	x	-	-
lpmove	x	-	x	-	-
lpsched	x	x	x	-	-
lpshut	x	-	x	-	-
makekey	x	-	-	x	-
mkfs	x	x	x	x	AS_CMD
mklost+found	x	-	-	x	-
mknod	x	x	x	x	AS_CMD
mkpart	-	x	-	-	-
mkunix	-	x	-	-	-
monacct	x	x	x	-	AS_CMD
mount	x	x	x	x	AS_CMD
mountall	-	x	-	-	-
mtjcpio	x	-	-	-	-
mvdir	x	x	x	-	AS_CMD
ncheck	x	x	x	x	AS_CMD
nlsadmin	-	x	-	-	-
nsquery	-	x	-	-	-
nulladm	x	-	x	-	-
ping	x	x	-	-	-
portmap	x	-	-	-	-
prctmp	x	x	x	-	AS_CMD
prdaily	x	x	x	-	AS_CMD
prfdc	x	-	x	-	-
prfld	x	-	x	-	-
prfpr	x	-	x	-	-
prfsnap	x	-	x	-	-
prfstat	x	-	x	-	-
profiler	x	x	x	-	-
prtacct	x	x	x	-	AS_CMD
pwck	x	x	x	-	AS_CMD
rc	x	-	x	x	-
rc0	-	x	-	-	-
rc2	-	x	-	-	-
rdump	x	-	-	x	-
reject	x	-	x	-	-
renice	x	-	-	x	-
restore	x	-	-	x	-
rexecd	x	-	-	x	-
rfadmin	-	x	-	-	-

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
rfpasswd	-	x	-	-	-
rfstart	-	x	-	-	-
rfstop	-	x	-	-	-
rfuadmin	-	x	-	-	-
rfudaemon	-	x	-	-	-
rlogind x	x	-	x	-	
rmntstat	-	x	-	-	-
rmount	-	x	-	-	-
rmountall	-	x	-	-	-
rmt	x	x	-	x	-
route	x	-	-	x	-
routed	x	-	-	x	-
rpcinfo	x	-	-	-	-
rrestore	x	-	-	x	-
rshd	x	x	-	x	-
runacct	x	x	x	-	AS_CMD
rwhod	x	-	-	x	-
sa1	x	x	x	-	AS_CMD
sa2	x	x	x	-	AS_CMD
sadc	x	x	x	-	AS_CMD
savecore	x	-	-	x	-
sendmail	x	x	-	x	-
setmnt	x	x	x	-	AS_CMD
shutacct	x	x	x	-	AS_CMD
shutdown	x	x	x	x	-
startup	x	x	x	-	AS_CMD
strace	-	x	-	-	-
strclean	-	x	-	-	-
strerr	-	x	-	-	-
swap	-	x	-	-	-
swapon	x	-	-	x	-
sync	x	x	x	x	AS_CMD
sysadm	x	x	x	-	-
sysdef	x	x	x	-	AS_CMD
tapesave	x	-	x	-	-
telinit	x	-	x	-	-
telnetd	x	x	-	x	-
tftpd	x	x	-	x	-
tic	x	x	x	-	TI_CMD
tpfix	-	x	-	-	-
tunefs	x	-	-	x	-
turnacct	x	-	x	-	AS_CMD
uadmin	-	x	-	-	-
umount	x	x	x	x	AS_CMD
unadv	-	x	-	-	-
unlink	x	x	x	-	AS_CMD
update	x	-	-	x	-

Command	DG/UX 4.0,4.1	386/ix 1.06	AT&T V.3	BSD 4.2	SVID 2
uucheck	x	x	-	-	-
uucico	x	x	-	-	-
uuclean	-	x	x	x	-
uucleanup	x	x	-	-	-
uugetty	x	x	-	-	-
uusched	x	x	-	-	-
uutry	x	x	-	-	-
uuxqt	x	x	-	-	-
vipw	x	-	-	x	-
vlx	-	x	-	-	-
volcopy	x	x	x	-	AS_CMD
wall	x	x	x	x	AU_CMD
whodo	x	x	x	-	AS_CMD
wtinit	-	x	-	-	-
wtmpfix	x	x	x	-	AS_CMD
xtd	-	x	-	-	-
xts	-	x	-	-	-
xtt	-	x	-	-	-

End of Appendix

Appendix B
Comparison of C Compilers

Table B-1 compares the GNU and Green Hills C compilers.

Table B-1 Comparison of C Compiler Options

Description of Command Line Option	GNU C	Green Hills C
Produce only object files	-c	-c
Do not discard comments in preprocessor output	-C	-C
Define 'name' to preprocessor with value 1		-Dname
Define 'name' to preprocessor with value 'string'		-Dname=string
Run preprocessor and output to standard output	-E	-E
Generate debug information	-g	-g
Generate a frame pointer for stack traces		-ga
Add 'name' to searchlist for #include	-Iname	-Iname
Generate minimum profiling information	-p	-p
Generate maximum profiling information	-pg	-pg
Place executable in file 'name'	-o name	-o name
Optimize	-O	-O
Optimize assuming memory locations don't change		-OM
Optimize for speed even if more memory is used		-OL
Put all data in the text section		-R
Produce only assembly files as output	-S	-S
Undefine a preprocessor symbol	-Uname	-Uname
Print out program name and arguments as they run	-v	-v
Suppress warning diagnostics	-w	-w
Allocate enum types as smallest type to represent values		-X6
Disable local optimizer		-X9
Do not produce output		-X13
Allocate only local register variables to register		-X18
Map all identifiers to uppercase		-X21
Display filenames as they are opened		-X32
Emit a warning when dead code is eliminated		-X37
Don't move frequent use procedures & addresses to registers		-X39
Set default to signed for int, short, and char		-X55
Don't underscore before global variables and procedures		-X58
Target system is UNIX System V		-X74
Turn off branch tail merging optimization		-X80
Allow extrn variables to be initialized		-X81
Generate error messages for C anachronisms		-X84
Generate .bss for zero initialized statics		-X85
Pack structures with no space between members		-X89
Allow redefinition of #define synbols to preprocessor		-X105
Target is BSD 4.2		-X114
Target is BSD 4.1		-X115
Enable ANSI extensions		-X153
Don't stop for code generator abort or internal compiler error		-X164
Only evaluate float operands as float, else use double		-X167
Don't move invariant floating point expressions out of loops		-X168
Don't create a static base register		-X171
Suppress output from #ident		-X187
Use Fortran mixed mode expression evaluation rules		-X188

Description of Command Line Option	GNU C	Green Hills C
Don't put "." before assembler directives		-X202
Don't generate inline code for external calls with optimization		-X211
Suppress elimination of Jumps to jumps		-X219
Suppress common subexpression elimin. and value propagation		-X230
Functions that return float return single precision, not double		-X233
Apply associative rules in common subexpression elimination		-X237
Print description of enabled X switches		-X255
Don't remove useless sign and zero extend functions		-X264
Suppress register database phase of peephole		-X265
Repeat peephole phase iteratively til code is unchanged		-X266
Don't delete redundant register alignments		-X268
Don't merge and remove excess move instructions		-X271
Suppress relvar code in database phase		-X272
Don't merge index calculation into load instruction		-X278
Suppress block merge phase		-X285
Truncate names to eight characters on input		-X304
Don't recognize the C "asm" inline directive		-X306
Don't reorder instructions		-X307
Perform tail recursion optimization		-X308
Don't make multiple copies of blocks during block merge		-X311
Don't recognize ?: operators as absolute value and min/max		-X312
Enable all ANSI extensions appropriate to UNIX		-X316
Generate "stabd" psuedo-ops for line numbers		-X329
Allocate unused variables if symbolic debugging enabled		-X331
Substitute multiply for floating point divides where possible		-X332
Don't pass front end information to peephole optimizer		-X333
Use ANSI rules for operator assignment		-X334
Suppress adrconst optimizations		-X344
In ANSI, allow /**/ to be macro concatenation operator		-X350
Don't extend float arguments to double to pass to functions		-X352
Perform common subexpression analysis twice		-X352
Put line numbers in the assembly file		-X370
Don't associate over parentheses		-X380
Support ANSI trigraphs	-T	
Create dependency information for make files	-M	
Create user header dependency information for make files	-MM	
Make debugging dumps at specified phases	-dphase	
Specify target machine	-mspecification	
Don't use standard libraries and files	-nostdlib	
Don's store float variables in registers	-ffloat-store	
Don't recognize asm, inline, or typeof	-fno-asm	
Pop arguments to function call on return	-fno-defer-pop	
Optimize with loop strength reduction	-fstrength-reduce	
Force memory operands into registers before arithmetic	-fforce-mem	
Force memory address constants into registers before arithmetic	-fforce-addr	
Drop frame pointer for register unless required	-fomit-frame-pointer	
Integrate all simple functions into their callers	-finline-functions	
Always create separate runtime callable function	-fkeep-inline-functions	
Store string constants in the writable data segment	-fwritable-strings	
Don't put function addresses in registers	-fno-function-cse	
Treat all memory references through pointers as volatile	-fvolatile	
Let type char be unsigned	-funsigned-char	
Let type char be signed	-fsigned-char	
Treat register reg as fixed	-ffixed-reg	
Treat reg as allocatable and clobbered by function calls	-fcall-used-reg	
Treat reg as allocatable and saved by function calls	-fcall-saved-reg	
Warn on implicit instructions	-Wimplicit	
Warn on when function return defaults to int	-Wreturn-type	
Warn when a local variable is unused	-Wunused	
Warn when a comment start is inside a comment	-Wcomment	
Warn on copying a nonconstant to a constant char	-Wwrite-strings	
Issue extra warnings	-W	

End of Appendix

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

O

P

Q

R

S

T

U

V

X

Related Manuals

Following is a list of related software and hardware manuals:

Software Manuals

This section lists manuals by category: DG/UX system, DG/UX communications, SNA DG/UX, languages, and graphics.

DG/UX System

General

Finding Your Way Around the DG/UX Documentation (069-701013).

Learning the UNIX Operating System (069-701042).

Using

Using the DG/UX System: The Shells and Commands (069-701035).

Using the DG/UX System: The Editors (069-701036).

User's Reference for the DG/UX System (093-701054).

Administering

Installing and Managing the DG/UX System (093-701052).

System Manager's Reference for the DG/UX System (093-701050).

Programming

Using DG/UX Programming Tools (093-701048). This manual describes tools such as interprocess communication, **curses**, **terminfo**, **SCCS**, **awk**, **make**, **lint**, **lex**, **yacc**, **ld**, **sdb**, and **as**.

Programmer's Reference for the DG/UX System (Volumes 1 and 2) (093-701055 and 093-701056). Volume 1 describes commands and system calls. Volume 2 describes subroutines and libraries, file formats, miscellaneous features, and communications protocols.

Porting Applications to the DG/UX System (093-701047).

Writing a Device Driver for the DG/UX System (093-701053).

STREAMS Primer for the DG/UX System (069-701033).

STREAMS Programmer's Guide for the DG/UX System (069-701034).

DG/UX Communications (NFS and TCP/IP)

Managing NFS and Its Facilities on the DG/UX System (093-701049).

DG TCP/IP (DG/UX) User's Manual (093-701023).

Installing and Managing DG TCP/IP (DG/UX) (093-701051).

Programming with DG TCP/IP (DG/UX)

SNA DG/UX Communications

Using DG/UX SNA/3270 (069-701030).

Managing DG/UX SNA/3270 (069-701044).

Using DG/UX SNA/RJE (069-701031).

Managing DG/UX SNA/RJE (093-701046).

DG/UX SNA/3270 API Programmer's Reference (093-701045).

Languages

C: A Reference Manual by Samuel Harbison and Guy Steele (069-100226).

Green Hills Software User's Manual: C-88000 (069-100230).

Green Hills Software User's Manual: FORTRAN-88000 (069-100231).

Green Hills Software User's Manual: Pascal-88000 (069-100232).

The C Preprocessor (Stallman, Free Software Foundation).

Using and Porting GNU CC (Stallman, Free Software Foundation).

Graphics

NCD16 Network Display Station Installation and Operation Manual (93-00001-A). Network Computing Devices Inc., 350 N. Bernardo Ave., Mountain View, CA 94043.

OSF Motif Window Manager External Specifications (Open Software Foundation).

OSF Motif Toolkit External Specifications (Open Software Foundation).

X Window System User's Guide (069-100229).

Xlib Reference Manual for Version 11 (O'Reilly & Associates, Inc.).

Xlib Programming Manual for Version 11 (O'Reilly & Associates, Inc.).

Hardware Manuals

This section lists manuals by category: processors, peripherals, and controllers.

Processors

Programming Your AV 400 Series Workstation (014-001800).

Setting Up and Starting AViiON 300 Series Stations (014-001801).

Using the AViiON Station Control Monitor (SCM) (014-001802).

Setting Up and Starting AViiON 5000 Series Systems (014-001806).

Starting AViiON 6000 Series Systems (014-001807).

MC88100 User's Manual by Motorola (MC88100UMAD/AD).

MC88200 User's Manual by Motorola (MC88200UMAD/AD).

uPD72120 Advanced Graphics Display Controller User's Manual (NEC).

Terminals

D216/D216E and D412/D462 Display Terminals User's Manual (014-001396).

The D216/D216E and D412/D462 Display Terminals Technical Reference Manual (014-001397).

DASHER D210/D211 Display Terminal Programmer's Reference Card (014-000763).

DASHER D210/D211 Display Terminal User's Manual (014-000746).

DASHER D214/D215 Display Terminal Programmer's Reference Manual (014-001164).

DASHER D214/D215 Display Terminal User's Manual (014-001163).

DASHER D220 Color Display Terminal Programmer's Reference Card (014-000965).

DASHER D220 Color Display Terminal User's Manual (014-000950).

DASHER D410/D460 Display Terminal Programmer's Reference Card (014-000760).

DASHER D410/D460 Display Terminal User's Manual (014-000761).

DASHER D410/D460 Programmer's Reference Card (014-001123).

DASHER D410/D460 Programmer's Reference Manual (014-001114).

DASHER D410/D460 User's Manual (014-001124).

DASHER D411/461 Owner's Manual (014-001161).

DASHER D411/461 Programmer's Reference (014-001162).

DASHER D470C Color Graphics Terminal Programmer's Reference Card (014-000787).

DASHER D470C Color Graphics Terminal User's Manual (014-000788).

DASHER D470C Programmer's Reference (014-001015).

Disk and Tape Hardware

Installing Your Half-Height Winchester Disk Drive (014-001722).

Installing Your Half-Height Streaming Cartridge Tape Drive (014-001699).

Installing Your Model 6491 Series Disk Drive (014-001460).

Model 6586/6587 Magnetic Tape Drive Installation Guide (014-001692).

Model 6536 Operation and Maintenance (014-001699).

Disk Drive Model 6491 Series General Specifications (014-001460).

Controllers

Installing and Operating the Model 6544 VMEbus SCSI Host Adapter with Floppy Port (014-001756).

Installing and Operating the Model 6544 ESDI Drive Controller (014-001757).

Installing and Operating the Model 6545 VMEbus SMD Disk Controller (014-001758).

End of Related Manuals

134-755-02

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name _____ Your Title _____

Company _____ Phone _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title _____ Manual No. _____

Who are you? ☐EDP/MIS Manager ☐Analyst/Programmer ☐Other _____
☐Senior Systems Analyst ☐Operator _____
☐Engineer ☐End User

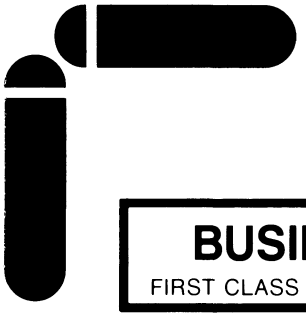
How do you use this manual? *(List in order: 1 = Primary Use)*

___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide _____

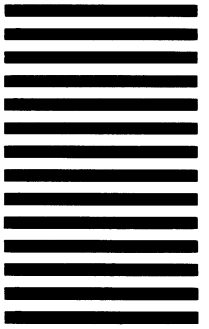
		Yes	No
About the manual:	Is it easy to read?	☐	☐
	Is it easy to understand?	☐	☐
	Are the topics logically organized?	☐	☐
	Is the technical information accurate?	☐	☐
	Can you easily find what you want?	☐	☐
	Does it tell you everything you need to know?	☐	☐
	Do the illustrations help you?	☐	☐

If you wish to order manuals, use the enclosed TIPS Order Form (USA only) or contact your sales representative or dealer.

Comments:



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 WESTBORO, MA 01581

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral

Customer Documentation
MS E-111
4400 Computer Drive
P.O. Box 4400
Westboro, MA 01581-9890

Porting Applications to the DG/UX™ System

093-701047-00

Cut here and insert in binder spine pocket

DataGeneral
Data General Corporation, Westboro, Massachusetts 01580

093-701047-00