Data General

Introduction to RDOS

Data General

Introduction to RDOS

Notice

DATA GENERAL CORPORATION (DGC) HAS PREPARED THIS DOCUMENT FOR USE BY DGC PERSONNEL, CUSTOMERS, AND PROSPECTIVE CUSTOMERS. THE INFORMATION CONTAINED HEREIN SHALL NOT BE REPRODUCED IN WHOLE OR IN PART WITHOUT DGC PRIOR WRITTEN APPROVAL.

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

CEO, DASHER, DATAPREP, ECLIPSE, ENTERPRISE, INFOS, microNOVA, NOVA, PROXI, SUPERNOVA, PRESENT, ECLIPSE MV/4000, ECLIPSE MV/6000, ECLIPSE MV/8000, TRENDVIEW, SWAT, GENAP, and **MANAP** are U.S. registered trademarks of Data General Corporation, and **AZ-TEXT, DG/L, ECLIPSE MV/10000, GW/4000, GDC/1000, REV-UP, XODIAC, DEFINE, SLATE, microECLIPSE, BusiPEN, BusiGEN** and **BusiTEXT** are U.S. trademarks of Data General Corporation.

Revision History:

093-000083
Original Release - September 1972
First Revision - October 1972
Second Revision - November 1972
Third Revision - October 1973
Fourth Revision - January 1975
069-000002 (replaces 093-000084-04)
Original Release - June 1978

Revision History:

069-000022
Original Release - September 1978
First Revision - August 1979

This manual, *Introduction to RDOS* (069-400011-00), supersedes the *Introduction to RDOS* (069-000002-00), and the *Learning to Use RDOS* (069-000022-01).

Ordering No. 069-400011
©Data General Corporation, 1983
All Rights Reserved
Printed in the United States of America
Rev. 00, September 1983

Preface

This manual addresses both new and experienced users of RDOS. For new users, basic information is presented on the features and use of the system, the Command Line Interpreter, the file system, text editors, and peripheral device I/O. For experienced users, RDOS features are summarized in discussions of I/O, memory management, multitasking, hardware and software options, program development, and languages.

The contents of Chapters and Appendices are listed below, followed by suggested reading paths for new and experienced RDOS users.

Chapter 1	Overview
Chapter 2	The Command Line Interpreter (CLI) CLI functions, command line format and extensions, terminal protocol, punctuation, error handling.
Chapter 3	Files, Disks, and Directories Device and file names, disk storage, disk preparation, directory management, links, templates, file attributes and characteristics.
Chapter 4	SPEED and EDIT Text Editors Tutorials for EDIT and SPEED, with sample editing sessions illustrating basic commands.
Chapter 5	Advanced Features of the CLI Indirect and macro command files, Batch processing, variables, command interpretation and parsing order, CLI-assembly language interface.
Chapter 6	Input/Output RDOS peripherals, spooling, I/O commands of the CLI, I/O through system calls.
Chapter 7	Memory Management and Multitasking RDOS organization, program development, RLDR, memory conservation, foreground-background processing, mapped systems, extended memory, task states and priorities, task calls, task timing.
Chapter 8	Special RDOS Features User device drivers and interrupts, power fail handling, Interprocessor Bus (IPB), Multiprocessor Communications Adapter (MCA), tuning.
Chapter 9	Assemblers and Program Utilities Extended Assembler and Macroassembler, RLDR, Library File Editor, Overlay Loader, debuggers.
Chapter 10	RDOS High Level Languages FORTRAN IV, FORTRAN 5, DG/L, Interactive COBOL, Extended BASIC, Business BASIC.
Chapter 11	RDOS Utilities Capsule summaries of the utilities.
Appendix A	CLI Command Summary
Appendix B	Control Characters
Appendix C	CLI Special Symbols
Appendix D	Batch Monitor Command Summary
Appendix E	System Call Summary
Appendix F	Task Call Summary
Appendix G	Programming Language Examples (Assembly language, FORTRAN IV, FORTRAN 5, DG/L, Interactive COBOL, Extended BASIC, Business BASIC.)

Reading Paths

New users of RDOS are directed to Chapter 1 for general information on RDOS features, and Chapters 2 through 6 for learning how to use RDOS. Users interested in system management should read Chapters 7, 8 and 11. Users interested in programming should read Chapters 7, 9 and 10.

Experienced users of RDOS are directed to Chapter 1 for general information on RDOS features, and Chapters 7 through 11 for summaries of RDOS functions, languages, and utilities. System programmers can benefit by reading the

Chapter 5 section on the CLI-assembly language interface, and the Chapter 6 section on I/O through system calls.

Related Manuals

The following manuals are part of a series of books published on RDOS.

Guide to RDOS Documentation (DGC No. 069-400012) describes all of the books that comprise the revised documentation set for RDOS, DOS, and RTOS and lists the previous books that each replaces.

How to Load and Generate RDOS (DGC No. 069-400013) explains how to load, generate, and maintain RDOS. Instructions cover preparing hardware, program loading, disk initialization, bootstrapping, tailoring, system backup, and system tuning.

RDOS/DOS Command Line Interpreter (DGC No. 069-400015) discusses the user interface with the operating system. It covers the Command Line Interpreter (CLI) features and command mechanisms, and instructs you on how to use CLI commands. Features and operation of the Batch monitor are also presented.

RDOS/DOS Text Editor (DGC No. 069-400016) documents how to load, use, and operate the Text Editor (Edit) or Multi-user Text Editor (Medit) to create and edit text files.

RDOS/DOS Superedit Text Editor (DGC No. 069-400017) introduces the commands and concepts of the Superedit Text Editor (Speed), which offers many powerful features for editing text.

RDOS System Reference (DGC No. 093-400027) describes RDOS system features, calls, and user device driver implementation for assembly language and high-level language programming.

RDOS/DOS User's Handbook (DGC No. 093-000105-04) summarizes the commands, calls, and error messages of the RDOS/DOS CLI, Batch monitor, and utility programs.

RODS/DOS Assembly Language and Program Utilities (DGC No. 069-400019) details the Extended Assembler (ASM), Macroassembler (MAC), Extended Relocatable Loader (RLDR and OVLDR), and Library File Editor (LFE) utilities that aid you in programming.

RDOS/DOS Debugging Utilities (DGC No. 069-400020) describes five utilities that assist you in editing, debugging, and patching programs–the Symbolic Editor (SEDIT), Symbolic Debugger (DEBUG), Disk Editor (DISKEDIT), and Patch (ENPAT and PATCH).

RDOS/DOS Sort/Merge and Vertical Format Unit Utilities (DGC No. 069-400021) offers functional information on Sort/Merge (RDOSSORT), which helps you manipulate records in data files, and the Vertical Format Unit (VFU), which enables you to define data channel line printer page formatting.

RDOS/DOS Backup Utilities (DGC No. 069-400022) presents the features and operation of the utilities that involve disk and tape backup. These are BURST/TBURST, DBURST/MBURST/RBURST, DDUMP/DLOAD, FDUMP/FLOAD, and OWNER.

Conventions

We use the following conventions for command formats in this manual:

COMMAND required [*optional*] . . .

Where	Means
COMMAND	Enter the command (or its accepted abbreviation) as shown. Upper-case letters indicate the command mnemonic.
required	Enter an argument (such as a filename).
	Required arguments appear as:
	$required_1$ | $required_2$
	In this case, you may choose between the arguments. Do not type the vertical bar; it merely separates the choices.
[*optional*]	Brackets mean that you have the option of entering the argument. Command switches also appear in this format. Do not include the brackets in your code; they only set off the choices.
. . .	Repeat the preceding entry or entries. The explanation indicates exactly what to repeat.
. . .	Where this symbol appears, the process has continued without incident and you may now take the next action described.

In examples of dialogue, we use:

THIS TYPEFACE TO SHOW YOUR ENTRY

and

THIS TYPEFACE TO SHOW SYSTEM RESPONSES

Additionally, we use certain symbols in special ways:

Symbol	Means
⟨CR⟩	Carriage Return. Press the CR key on your keyboard. **NOTE:** If you have a D100, D200, or G300 terminal, you should press the New Line key (NEW LINE) on your keyboard instead.
□	Include a space at this point. (We use this to clarify command lines in some cases. Normally, you can see where you should put spaces.)
CTRL-	Depress and hold the Control key (CTRL) while you press the character that follows CTRL-.
⟨NL⟩	New Line. Press the NEW LINE key on your keyboard.
R	RDOS/DOS Command Line Interpreter prompt.

All numbers are decimal unless we indicate otherwise; for example, to indicate octal 35, we use 35_8.

The keys defined as DEL and RUBOUT perform the same function. Depending on the console you are using, you will find one of these keys on your keyboard. In this manual, we use DEL to represent that function.

The up arrow symbol (↑) is also executed by different keys, depending on your console. You execute it by pressing either SHIFT-N or SHIFT-6. In this manual, we reference SHIFT-6 to execute the up arrow symbol.

We welcome your suggestions for the improvement of this and other Data General publications. To communicate with us, use the postpaid comment form at the end of this manual.

Table of Contents

Chapter 8
Special RDOS Features

Chapter 9
Assemblers and Program Utilities

Chapter 10
RDOS High Level Languages

Chapter 11
RDOS Utilities

Tables

Figures

Chapter 1

Overview

RDOS, Data General's Real-Time Disk Operating System, supports real-time control, program development, or batch operations in a flexible, user-oriented environment. RDOS runs on a wide range of NOVA® and ECLIPSE® configurations, including those that support Microproducts peripherals. Memory capacities range from 40 Kbytes on the NOVA to 2 megabytes on the ECLIPSE.

RDOS is compatible with the Disk Operating System (DOS) and the Real-Time Operating System (RTOS). Program development for RTOS takes place under RDOS, while DOS and RDOS can share some disks and some program applications.

Some major features of RDOS are:

- Disk and memory-residence,
- Mapped or unmapped foreground-background processing,
- Mapped extended memory,
- Multitasking,
- Broad peripheral support,
- User device driver support,
- Comprehensive, comprehensible Command Line Interpreter,
- Versatile disk file structure,
- Shared disk files,
- Buffered and unbuffered I/O,
- Interactive and batch job processing,
- Support for real-time FORTRAN IV, FORTRAN 5, DG/L, Interactive COBOL, Extended BASIC and Business BASIC applications,
- User program swaps, chains, and overlays,
- Automatic restart on power failure,
- Dual processor, shared disk support,
- Multiprocessor support,
- Tuning for improved performance.

Operating System Highlights

RDOS combines the capacity of a disk operating system with the speed of a memory resident system, forming a fast and efficient configuration. The RDOS multitasking facility makes it ideal for real-time process control because RDOS is able to schedule program control for many different subprogram tasks.This provides efficient use of system resources.

RDOS dual-ground processing enables two separate programs to run concurrently by partitioning main memory into foreground and background memory. The two programs share resources and can communicate with one another. The optional hardware Memory Allocation and Protection unit (MAP) makes dual-ground processing easier, and gives the user access to extended memory through virtual overlays and window mapping. With device driver interrupt support, users can add non-standard devices to an RDOS system, either loading the device driver software with the operating system, or introducing the device at runtime.

Using RDOS

To use RDOS, you need only a disk and a terminal connected to your NOVA or ECLIPSE processor. RDOS supports up to 2 megabytes of memory storage, over a billion bytes of disk storage, and sixteen magnetic tape transports (7 and 9 track) and cassette units. Additionally, RDOS can support data channel and non-data channel line printers; card readers; paper tape readers and punches; communications equipment; digital plotters; and multiprocessor communications apparatus.

The four paths of communication with RDOS are:

- Command Line Interpreter,
- Batch Monitor,
- System and task calls in an assembly language program,
- High-level language indirect access.

The Command Line Interpreter (CLI) is the primary interface with RDOS, providing commands to manage every aspect of the system. Through CLI commands, the user directs system generation and shutdown, file and directory management, input/output, system administration, and program development. The CLI also provides access to the RDOS system utilities, which include file backup software, text editors, compilers, assemblers, loader, and debuggers in addition to the Batch Monitor.

The Batch Monitor allows you to simulate a CLI session by grouping Batch commands (similar to CLI commands) in a file for later processing. Batch accepts job routines from a variety of devices and sends output to specified disk files or devices.

The RDOS disk file structure permits rationalizing of disk space in a hierarchy of partitions and subdirectories, with mechanisms for file protection, reference, and linking. Options for disk file block organization enable the user to balance disk resources with time requirements. RDOS utilities and CLI I/O commands facilitate data transfers between system devices.

Program Development

RDOS high-level language support includes FORTRAN IV, FORTRAN 5 and DG/L compilers, and the interpretive languages Extended BASIC and Business BASIC. The BASIC interpreters permit multiple-user programming. Assembly language programmers can use the Macroassembler for increased flexibility.

Program utilities include the Relocatable Loader for loading assembled or compiled code, the Library File Editor for maintaining libraries of binary files, and the Overlay Loader for easy overlay replacement. Debugging utilities allow breakpoint program execution and correction, symbolic disk editing, or patching of executable or overlay files. A variety of subroutine packages offer specialized aid for programming.

Memory management facilities permit user program swaps, chains and multiple overlays. Extended memory through the MAP feature allows greatly increased memory usage with virtual overlays and window mapping, including memory pages shared between foreground and background programs, and extended direct block I/O.

RDOS Options

Automatic restart after a power failure is a feature that can be enabled at system generation time. When a power loss occurs, RDOS transfers control to a routine that saves processor states; RDOS can also restart certain system devices. The user can supply device drivers to restart other devices. Systems with semiconductor memory need battery backup in order to implement the RDOS automatic restart.

Two hardware options enable the configuration of multiple processor RDOS systems. With the Interprocessor Bus (IPB), two processors can run up to four programs at once while sharing a disk and monitoring each other. With the Multiprocessor Communications Adapter (MCA), up to 15 processors can communicate with each other by means of their data channels.

The RDOS tuning feature allows the user to monitor use of system stacks, cells and overlays. With the reported statistics, the user can fine-tune the operating system or direct RDOS to self-tune in the system generation procedure.

Getting Started with RDOS

Initial loading procedures for RDOS include loading a starter system from the RDOS release medium, initializing the system disk with the DKINIT program, and installing on the disk a bootstrap program and the RDOS utilities.The starter system is a general-purpose operating system suited to any legal RDOS configuration of disks and diskettes. Generation of a new operating system with the SYSGEN program is the next step in initial loading.

SYSGEN allows you to tailor RDOS systems to your hardware configuration and application needs. SYSGEN accepts your specifications on I/O devices, memory management for mapped and unmapped systems, disk file structure, tuning, user-defined devices, and other system concerns, and then builds an appropriate operating system.

Initial loading procedures for RDOS are documented in the manual *How to Load and Generate RDOS*, (DGC No. 069-400013).

When RDOS is loaded (either the starter system or a tailored system), it runs the CLI. You direct all system operations from the CLI. The remaining chapters in this manual detail the use of the CLI and summarize RDOS capabilities.

Chapter 2

The Command Line Interpreter (CLI)

When RDOS starts, it first executes the Command Line Interpreter (CLI), your primary interface to RDOS. Through commands to the CLI, you direct RDOS to create files, transfer files between peripheral devices, develop and execute programs, and invoke system utilities. The CLI accepts commands from the terminal, translates them into RDOS commands, and returns results to the terminal. The CLI is thus a *dynamic terminal interface* to the operating system.

The CLI can be described as a system utility, that is, an auxiliary program to be used to perform tasks, such as editing text or sorting data. In fact, the scope of the CLI is so broad and its commands so versatile that practically every feature of RDOS can be exercised through it, including the invocation of other system utilities. Consequently, a familiarity with the CLI command repertoire serves as an introduction to the capabilities of RDOS.

Often-used sequences of CLI commands or frequently needed text can be grouped in *macro files* or *indirect files*, saving time and typing for the user. The CLI processes the commands or outputs the text in macro or indirect files when you call the files. In addition, CLI commands form the basis of commands to the Batch Monitor, itself a system utility invoked through the CLI. The Batch Monitor processes predetermined sequences of CLI-like commands, sending output to a specified device or file. We discuss macros, indirect files and the Batch Monitor, as well as other advanced features of the CLI, in Chapter 5 of this manual.

CLI commands can be used to perform five functions:

- *File Management* commands create, type, copy, list, rename, append, compare, and delete files; output files to devices; link files between directories.
- *Directory Management* commands create, initialize and release directories; set or change current directory; control magnetic tape I/O.
- *System Control* commands set, display, and collect system statistics; manage memory; boot system; control spooling; display messages; release devices.
- *System Utilities* commands invoke system utilities, including editors, backup software, compilers, assemblers, loader, debuggers.
- *Batch Monitor* commands perform many of the above functions in Batch mode.

This chapter introduces the CLI command structure, including the command line format, terminal protocol, CLI punctuation, control characters, command repetition, grouping and expansion, and error handling. Along the way, we use some basic CLI commands as examples. Throughout this manual, additional CLI commands are described where relevant, and Appendix A contains a complete list of CLI commands with brief descriptions of their functions.

Command Line Format

The CLI command line follows a precise but simple format that you can expand in a variety of ways to form more complex commands. Through the use of modifiers (*arguments*, and *switches*), and punctuation, CLI commands can be formulated to act on single elements or groups of elements, or to repeat actions. Separate commands can be given in the same line, and long commands can be extended over several lines.

The CLI displays a *prompt* (the letter R), and outputs a new line, indicating that it is *ready* to accept commands. You use the CLI by typing a command and terminating it with a carriage return ⟨CR⟩. When the CLI completes a command or finishes running a utility, it again displays the R prompt.

The format of a CLI command is

COMMAND/*global-switch arguments/local-switch*

where:

COMMAND is a mnemonic describing the action to take or the utility to invoke.

Argument signifies the object of the command action.

Global switch modifies the action of COMMAND.

Local switch modifies the argument. Switches are inserted in a command line with a slash (/). CLI commands must be entered in full, without abbreviation.

Some commands require no arguments or switches. For example, the command GTOD returns today's date and the time, set when RDOS was loaded, thus:

```
R

GTOD

12/14/82 12:58:04

R
```

Arguments

Most commands take one or more arguments, indicating the object(s) of the command. For example,

```
R

SPEED filename
```

invokes the ECLIPSE Superedit text editor SPEED to edit the text in argument filename. The next example shows a command with two arguments:

```
R

PRINT filea fileb
```

directs RDOS to print the two files, filea and *fileb*, on the line printer. In the case of the PRINT command, multiple arguments are optional. Some commands logically require a second argument. For example,

```
R

RENAME test newjob
```

causes the old file test to be named newjob.

Global Switches

One or more global switches, added immediately after COMMAND, affect the execution of the entire command. Global switches are always single alphabetic characters, and their meanings vary with the command used.

For example, the command LIST directs the CLI to list file directory information. Used without switches or arguments, the command returns a list including the file name, size, and type of all nonpermanent files in the current directory. (A permanent file has the attribute P and cannot be deleted or renamed. See Chapter 3 for a discussion of file attributes.) Global switches alter the contents of the list returned, thus:

```
R

LIST/C
```

directs the CLI to include the file *creation* time in the list of information on each file.

```
R

LIST/C/S
```

directs the CLI to *sort* the list alphabetically by file name, as well as to include file creation time.

Local Switches

Local switches modify command arguments. Inserted in a command line following an argument, a local switch affects that argument only, not other arguments in the command line. Local switches are either *letter* switches, which modify the action of COMMAND on the argument, or *numeric* switches, which specify the number of times COMMAND is to act upon the argument.

Local Letter Switches

Local letter switches, which modify the action of COMMAND on *argument*, are single alphabetic characters whose meanings differ, depending on the commands and the arguments to which they are appended. For example, in the command

```
R

LIST/S/B 08-01-82/B
```

the argument 08-01-82 is modified with the local letter switch /B, directing RDOS to list only those files created *before* that date. The global switches /S/B direct RDOS to return a *sorted*, *brief* listing, including only file names and excluding file sizes and types.

Local Numeric Switches

Local numeric switches, in the range of 0 to 9, cause the action of COMMAND to be repeated on the argument the number of times specified, in effect repeating the argument. For example,

```
R

PRINT afile/3 jfile/2
```

is equivalent to the command

```
R

PRINT afile afile afile jfile jfile
```

For an action to be repeated more than nine times, additional numeric switches must be included in the command line for a cumulative effect. The command

```
R

PRINT jfile/9/6
```

produces 15 copies of jfile. A /0 or /1 switch is ignored by the CLI when used without additional numeric switches: the argument is repeated once in the command as if no numeric switch were specified.

Terminal Protocol

Terminal protocol includes the use of special keys on your terminal, the use of *control characters* defined in RDOS, and CLI punctuation. Through terminal protocol you can stop and continue programs or terminal input/output (I/O), correct typing mistakes, and extend the meaning of CLI commands.

CLI Punctuation

CLI punctuation includes characters to set off parts of commands or to control execution sequence. Some punctuation has been shown earlier in this chapter in Command Line Format: ⟨CR⟩ serves to terminate a CLI command; a slash (/) sets off switches; commands and arguments are separated by spaces.

Additional CLI punctuation symbols are defined here, and more advanced CLI punctuation is described in Chapters 3 and 5. Appendix C contains a list of CLI Special Symbols.

CLI Command Terminators, Delimiters, and Extenders

A CTRL-L *terminates* a command line, just as a carriage return does. Note that on a display screen, CTRL-L clears the screen and that on a hardcopy terminal, it produces a formfeed. See the section ahead on Control Characters.

Command arguments can be *delimited* by either a space, as shown before, or a comma (,). For example, to delete two files:

```
R

DELETE afile,bfile
```

Two or more commands can be given on the same line (before ⟨CR⟩) if they are *delimited* with a semicolon (;). With a multiple-command line, the commands are executed sequentially (left to right) after the command line is terminated by a ⟨CR⟩. For example

```
R

RENAME testfile newdata;LIST;PRINT newdata ⟨CR⟩
```

Commands can be *extended* beyond one screen line (80 characters) by the use of ↑, (SHIFT-6 or SHIFT-N) *before* a ⟨CR⟩. The entire command will be executed when a terminator (⟨CR⟩) is given. Note that a delimiter (space, comma, or semicolon) must be included before or after a line break; the line break must not split a command or argument. For example,

```
R

PRINT afile,bfile,cfile,dfile;DELETE aold,bold,cold,
dold;LIST/S/C 09-15-82/B;GTOD
```

Note that a comma (,) separates the filenames cold and dold.

Special Terminal Keys

In addition to the regular character set, the terminal keyboard is equipped with special function characters. Due to the variety of terminals, these keys are described in general terms.

The ⟨CR⟩, as mentioned earlier, serves as a terminator to CLI commands. On some terminals this key is labelled RETURN. On the newer Data General terminals, the D100, D200 and G300, the New-line key, under RDOS, functions just as the ⟨CR⟩ or RETURN key does, as a terminator to commands.

The DEL or RUBOUT key allows typing mistakes to be corrected. Used a single time or sequentially, the key erases characters from right (the last character typed) to left. On cathode ray tube (CRT) screens, the deleted character simply disappears. On printing (hardcopy) terminals, DEL or RUBOUT echoes a backarrow (←) or underscore (_) for each erased character.

To cancel an entire line, type a backslash (\) or a SHIFT-L, which advances you to a new line without returning the R prompt.

CAUTION:*The NOVA 4, S/20, S/120, S/140 and S/280 processors are equipped with Virtual Console, that is, software that replaces processor front panel switches. If, while running the CLI, you inadvertently strike the BREAK key, the Virtual Console is invoked and displays its prompt, an exclamation point (!). To return to the CLI, type P (upper-case P).*

Control Characters

Control characters are specially typed characters that signal the CLI to perform a function immediately, even if a CLI command or program is executing. Control characters are also called *program controls* and *interrupts*, signifying their use in altering or stopping program flow.

A control character is implemented by pressing simultaneously the CTRL key and a particular character, for example A; no carriage return is necessary, since the control function is effected immediately. In this manual, a control character is shown as CTRL-character; for example, CTRL-A or CTRL-S.

Some basic RDOS control characters are defined below. A list of all RDOS control characters and their functions is contained in Appendix B.

CTRL-A Halts execution of CLI command, user program or system utility, *except* text editors; echoes -INT (interrupt) and returns to the CLI ready prompt.

CTRL-L Terminates CLI command line, erases cathode ray tube (CRT) display screen.

CTRL-S Suspends terminal output; useful on CRT display screens.

CTRL-Q Resumes terminal output suspended by CTRL-S.

For instance, when the CLI is executing a TYPE filename command, typing out the contents of filename

- CTRL-A stops the CLI from typing out the rest of the file. The CLI echoes -INT and displays the R prompt, ready to accept another command.
- CTRL-S typed at your convenience stops the CLI from typing out the file. The TYPE command and the CLI is suspended.
- CTRL-Q directs the CLI to continue typing filename at the point where it was stopped by CTRL-S.

Note that control characters are also implemented in the text editors, EDIT and SPEED. While some text editor control characters have functions similar to the program control and interrupt characters of RDOS, generally they are meaningful only within the editor.

Command Repetition, Grouping and Expansion

RDOS supplies additional CLI punctuation to allow lengthy or complex commands to be entered in abbreviated form. With parentheses () or angle brackets ⟨⟩, you can effectively group or repeat the elements of a command or an argument. The CLI expands the command lines according to its rules of logic and sequence, then executes the expanded commands.

Parentheses can be used to enclose commands or arguments. Parentheses cause the CLI to expand a command line into multiple lines, in effect creating a separate line for each element within parentheses. Angle brackets can be used to enclose arguments, most often filenames. Arguments in angle brackets are expanded into a single command line.

Note that the expansion of commands contained in parentheses or angle-brackets is not visible to you; the CLI executes compressed commands as if they were given in the standard way. For example,

```
R

(LIST,PRINT) file.⟨a,b⟩
```

in effect expands to

```
LIST file.a,file.b

PRINT file.a,file.b
```

Both parentheses and angle brackets can be used more than once in a command line. Each opening parenthesis must be matched with a closing parenthesis, and each opening angle bracket with a closing bracket. Parentheses can appear within angle brackets and vice versa, ⟨()⟩ and (⟨⟩), but sets must not overlap, for example, (⟨)⟩.

Rules for Parentheses

The following rules apply to parentheses:

- Parentheses cannot be nested, for example,(()).
- Each element within parentheses must be delimited with a comma, for example, (PRINT,DELETE) or (afile,1file).
- The CLI extracts the elements within parentheses from left to right.
- When multiple sets of parentheses are used in a command line, the CLI extracts elements in turn from each set, and uses an element only once. For example, the command

  ```
  (LIST,PRINT) (filea,fileb)
  ```

  expands to

  ```
  LIST filea
  PRINT fileb
  ```

- Elements not in parentheses are executed in sequence. For example, the command

  ```
  (LIST,PRINT) afile (bfile,cfile)
  ```

  expands to

  ```
  LIST afile bfile
  PRINT afile cfile
  ```

- The CLI ends processing of the command when it has executed all elements from the parenthesized set that contains the greatest number of arguments.

Rules for Angle Brackets

The following rules apply to angle brackets:

- Angle brackets can be nested to any depth; for example ⟨⟨⟩⟩.
- Arguments within angle brackets must be separated with commas or spaces. Multiple commas or spaces indicate null arguments, which you use to exclude some elements from expansion. For example, the command

  ```
  PRINT file⟨,1,2,3⟩
  ```

  expands to

  ```
  PRINT file,file1,file2,file3
  ```

  The null argument in the angle brackets causes the argument file to be repeated verbatim, without expansion.
- Arguments within angle brackets are expanded from left to right. Unbracketed arguments are executed in sequence.
- When angle brackets are nested, the CLI expands the innermost level first, then proceeds to the outermost level. An argument within a level is linked to each argument in the next outer level. If the argument delimiter is a comma or space, it is linked to the end of the new string; if it is an angle bracket, it is treated as a null. For example,

  DELETE DP1:⟨test.⟨01 02 03⟩⟩

  expands to

  DELETE DP1:⟨test.01 test.02 test.03⟩

  then to

  DELETE DP1:test.01 DP1:test.02 DP1:test.03

CLI Error Handling

The CLI, as an interpreter of command lines, outputs an *error message* when it encounters something it cannot interpret. An error message comprises an explanation of the error and, usually, a description of the argument that caused the problem. The CLI is packaged with a repertoire of error messages upon which it draws to interpret a particular error. For example, the TYPE command instructs the CLI to type out the contents of a file on the terminal. With a typographical error in the command:

```
R

TYPE glm#ph
```

the CLI returns the message

```
ILLEGAL FILENAME:GLM#PH
```

Because the pound sign (#) is an illegal RDOS filename character, the filename glm#ph is illegal, and the CLI aborts the command.

CLI error messages most often result from typographical mistakes or from simple forgetfulness—forgetting the required format of a command, or what files are available. Though errors are inconvenient, they are usually of no consequence. The fact that the CLI can interrupt and respond to an error indicates its harmlessness. Nevertheless, caution should be used with any irreversible command, such as DELETE: a perfectly-typed DELETE command would evoke no error message from the CLI but might delete wanted files.

In most cases, for a command that contains some legal arguments and some illegal arguments, the CLI executes the legal arguments in the command before it returns the error message. For example, with the sequence

```
R

RENAME afile bfile; PRINT afile bfile
```

the CLI renames afile to bfile, prints bfile, and outputs the error message

```
FILE DOES NOT EXIST:AFILE
```

Afile has been renamed bfile, and therefore does not exist.

Message	Meaning
FILE ALREADY EXISTS	You have attempted to create a file whose name exists in the current directory, or to move or load such a file into the current directory without appropriate switches.
FILE DOES NOT EXIST	File does not exist in current or specified directory.
NO SUCH DIRECTORY	The directory is not initialized, is not in the location specified, or does not exist.
NOT A COMMAND	The CLI does not recognize the command, possibly because of a typographical error or modification of CLI.SV or CLI.OL.
NOT ENOUGH ARGUMENTS	Command requires more arguments. For example, RENAME requires two arguments.
YOU CAN'T DO THAT	You have attempted to do something grossly invalid, like typing ENDLOG without giving the password specified in LOG, or running an ECLIPSE program on a NOVA.

Table 2.1 Examples of CLI error messages

Table 2.1 contains a list of some error messages. A complete list of CLI error messages can be found in an appendix to *RDOS/DOS Command Line Interpreter* (DGC No. 069-400015) and in the *RDOS/DOS User's Handbook* (DGC No. 095-000104).

Chapter 3

Files, Disks and Directories

RDOS manages data in the form of a *file*, that is, a discrete collection of information, whether the information be text or binary machine code. You store files on peripheral devices, most conveniently disk, but also magnetic tape, paper tape, or card decks. You accomplish the transfer of data between devices, called input/output or I/O, by accessing both data files and I/O devices with filenames. Under RDOS, a file is either a collection of data or a device for sending or receiving data. Devices and files are named according to RDOS rules and conventions.

Disk files, the most common form of storage, are easily accessed and manipulated by the user. RDOS provides several modes for disk *file block organization*, allowing you to balance access time and disk resources. After preparing a disk for use under RDOS by means of *hardware formatting* and *disk initialization*, you can rationalize file storage by dividing the disk into logical sections called *directories* (partitions and subdirectories).

RDOS allows you to reference directories and files through the use of *directory specifiers* and *initialized*, *current* and *master* directories. You can also *RELEASE* directories, *LINK* to files, and access groups of files with *templates*. Files are protected and identified with *attributes* and *characteristics*.

This chapter discusses the naming of devices and files, disk storage methods, disk preparation, directories, directory and file access, and file protection and identification. While the subject of peripheral device I/O is covered in Chapter 6, this chapter lays the groundwork for understanding I/O.

Names of Devices

I/O device filenames are pre-defined by RDOS, while disk filenames are user-defined, following RDOS rules and conventions. The names of I/O devices are *reserved* in the sense that your files cannot be given the names of existing devices. If you attempt to name a file with an existing device filename, you receive the error messsage

FILE ALREADY EXISTS

Table 3.1 shows RDOS reserved device names. Note that most of the names begin with a dollar sign ($), which is a legal RDOS filename character.

Device Name	Device
$CDR	Punched card reader; mark sense card reader.
CTn	Data General cassette unit n, first controller. n is in the range 0-7.
DP0	Data General model 6001-6008 fixed-head disk, first controller.
DPn	Data General moving-head disk pack, first controller. unit n is 0, 1, 2, or 3.
DPnF	Top loader dual-platter Disk Subsystem. For the first controller, unit n is number 0, 1, 2, or 3. This unit has two disks. The top (removable) disk is DPn, the fixed disk is DPnF. This controller also supports diskette drives.
DSn	Data General Model 6063/6064 fixed head disk. The 6063 is single-density, the 6064 double density. n is 0, 1, 2, or 3.
DZn	6060 series disk unit, first controller. n is 0,1, 2, or 3. The 6060 uses single-density disks, 6061 uses double-density disks.
$DPI	Input dual processor link.
$DPO	Output dual processor link.
$LPT	80- or 132-column line printer.
MCAR	Multiprocessor communications adapter receiver.
MCAT	Multiprocessor communications adapter transmitter.
MTn	First controller, 7- or 9-track magnetic tape transport. n is in range 0-7.
$PLT	Incremental plotter.
$PTP	High-speed paper tape punch.
$PTR	High-speed paper tape reader.
QTY	Asynchronous Line Multiplexor (ALM), Asynchronous Data Communications Multiplexor (QTY), or Universal Line Multiplexor (ULM).
$TTI	Teletypewriter or display terminal keyboard*.
$TTO	Teletypewriter printer or CRT display.
$TTP	Teletypewriter punch.
$TTR	Teletypewriter reader.

Table 3.1 Reserved device names

*For most devices, RDOS supplies an end of file mark. On $TTI and QTY input, however, you must indicate an end-of-file with CTRL-Z.

You use some devices such as disk or magnetic tape for both input and output, but use other devices for one or the other. For example, card readers, paper tape readers and terminal keyboards are input devices, from which the processor reads data. Line printers, paper tape punches and terminal screens are output devices, on which the processor produces data. Note that a terminal comprises two devices: $TTI for keyboard input and $TTO for screen or hardcopy output.

In CLI commands, you can use device names just as you use disk filenames, *within the limits of the device*. For example, the device $LPT (line printer) cannot perform the command TYPE, but $LPT can be addressed as an output file.

Names of Files

A filename can consist of one to ten of the following ASCII characters: uppercase and lowercase letters, numbers, and a dollar sign ($). You can optionally append to a filename an *extension*, which is delimited by a period (.) and consists of up to two letters or numbers or dollar signs. The following are legal filenames:

```
Data1
Data.$$
D$.1
Prog1.SR
Prog1.RB
Prog1.SV
$Monthend.MC
$Yearend.MC
Roster.
Roster.A
```

As the above list shows, the dollar sign can appear anywhere in the filename or extension, the period can delimit a filename without an extension, and an extension can be one or two characters. You can enter filenames in upper or lower case, or both; RDOS converts characters to upper case.

Note that, though more than ten characters can be entered for a filename and more than two for an extension, RDOS recognizes only the legal number of characters. For example, with the command

```
R

RENAME file.01 longfilename.one
```

the CLI renames File.01 to Longfilena.ON, having truncated the extra characters in both the filename and extension.

Filename Conventions

Filename extensions are very useful in describing file types and classifying sets of files. For example, files named Database.01 and Database.02 can be two versions of a database. In addition, RDOS recognizes filename conventions, that is, certain extensions that are descriptive of file contents. For example, the extension .SV indicates that a file is executable binary code—a program. Some filename extensions are required by RDOS, and are, in fact, assigned by RDOS when the files are created by the user; other extensions are suggested for your convenience. Required and suggested filename conventions are shown in Tables 3.2 and 3.3.

Extension	File type	Origin
.DR	User disk directory name	Appended to the directory name by RDOS when CLI command CPART or CDIR is used to create disk partition or subdirectory
.MC	CLI macro file	Appended to the macro filename by the user
.RB	Relocatable binary file	Appended to the source filename by the appropriate compiler or assembler
.LS	Program assembly listing	Appended to the source filename by the assembler, if listing is requested by user
.SV	Executable save file	Appended to the source filename by RLDR, the Relocatable Loader, when source.RB is loaded
.OL	Overlay file	Appended to the source filename by RLDR, the Relocatable Loader, when source.RB is loaded
.BU	Backup file	Appended by text editor SPEED or EDIT to original file after editing session
.CM	CLI command file	Appended by the CLI to its command files (F)CLI.CM and (F)COM.CM
.VF	VFU line printer format file	Appended to the format filename by the VFU utility
.TU	Tuning report file	Appended to the operating system filename by the tuning mechanism
.LB	Library file	Appended to library file of relocatable binary files (.RB) by LFE utility

Table 3.2 Required filename extensions

Extension	File type
.FR	FORTRAN source file.
.SR	Assembly language source file.
.JB	Batch job command file

Table 3.3 Suggested filename extensions

RDOS Disk Storage Methods

Disk files are the fastest and most versatile form of storage. A disk file can be a data or program file, a directory containing other files, or a link file pointing to a file in another directory.

The disk filename rules, discussed earlier, ensure your orderly access to the files. Further RDOS conventions that govern disk access are disk file block organization, by which the contents of a file are written and read in a certain way, and disk directories, by which you divide the disk into logical segments.

Disk File Block Organization

The process of hardware formatting prepares a disk's sectors, tracks, and surfaces for logical addressing. (See the section called Preparing Disks later in this chapter.) The primary unit of data on a disk is a *block*, consisting of 256 16-bit words. Each disk block is accessed by RDOS by means of a *logical block address* from which RDOS computes the physical sector, track, and surface numbers of the area. The RDOS computation of the logical block address is transparent to the user. A logical disk block corresponds to a physical disk sector.

All RDOS disk files are stored and accessed in blocks, and RDOS uses the logical block address to keep track of file blocks, which are not necessarily physically contiguous on disk. Because time-consuming disk accesses are needed to read or write disk blocks, RDOS offers three types of disk file block organization, allowing the user to strike a balance between time constraints and disk resources. The three types of file organization are *sequential*, *random* and *contiguous*.

Sequential Files

In a sequential file, RDOS writes and reads disk file blocks in logical sequence. RDOS reserves the last word in each block (the last two words of a block on a multiple-platter disk) for a pointer to the next sequential block; the pointer consists of a logical block address. When building a sequential file, RDOS appropriates the next available disk block, and constructs a pointer to the block. Figure 3.1 illustrates sequential block organization.

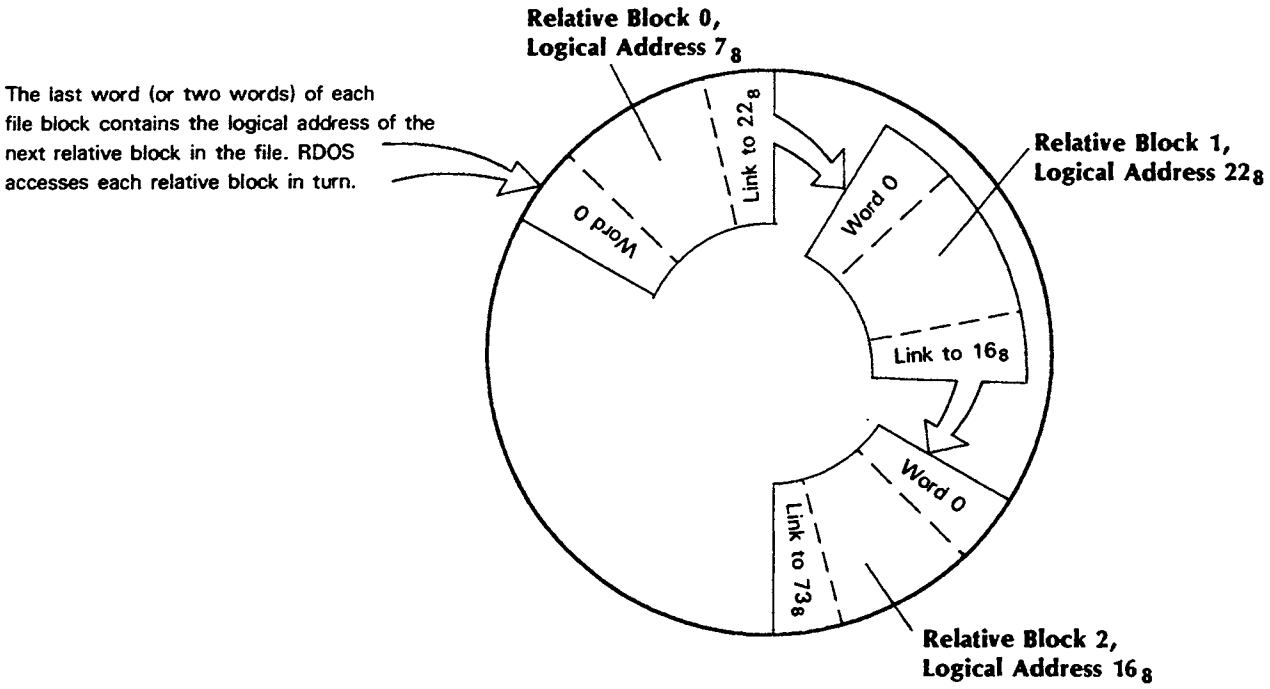


Figure 3.1 Sequential file block organization

ID-00328

Sequentially-ordered blocks are sequential in the sense that, after processing any given block, RDOS can step either to the previous block or to the next block in the series. To access, for example, the tenth block after the first block, RDOS reads the nine intervening blocks. Sequential files are flexible in that you can expand file length, but sequential access and system buffering are relatively time-consuming.

The CLI command to create a sequential file is CREATE. An example of the CLI command is

```
R

CREATE filea fileb
```

Both filea and fileb are created with zero length and no attributes (you can assign attributes to protect files; see the section of this chapter ahead entitled File Attributes). You can put data into zero length files with either a text editor or the CLI command XFER.

Random Files

For a random file, RDOS creates and maintains a separate *file index*, each word of which contains a logical block address for each block in the file. The first data block in the file is number 0, the second 1, and so on; the first entry in the index is entry 0, and contains the logical address of block 0, and so on. (On larger disks, generally, two words of the file index are used to contain one logical block address.) RDOS can thus step randomly to any block in the file by accessing the file index. For example, RDOS needs two disk accesses to write the first random file block of a file, one for the index and one for the file block; subsequent write operations need only one disk access for each file block. Random block organization is shown in Figure 3.2.

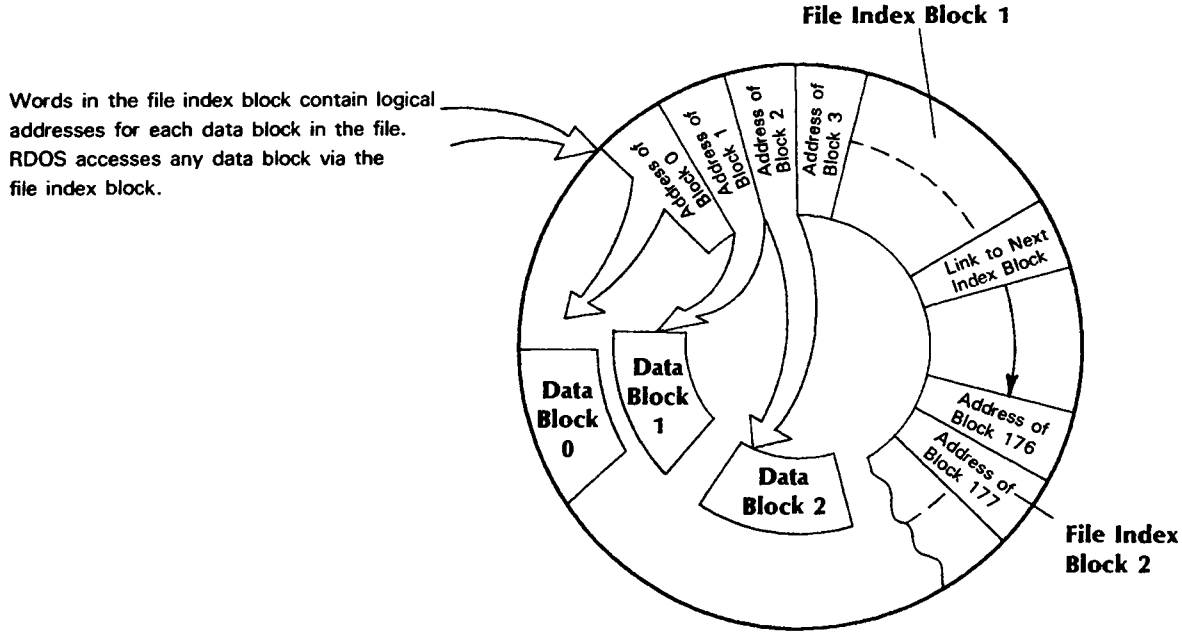


Figure 3.2 Random file block organization ID-00329

A file index is organized in blocks, just as a file is. For a large random file, RDOS maintains more file index blocks, as needed. File indexes are organized as sequential files, with the last word or two of an index block pointing to the next index block.

File index blocks reside in user disk space, though they are neither visible in the file nor apparent in a file LIST.

With random files, all 256 words of a file block are used for data storage. The organization of random files generally allows for faster I/O. When a file index or data block remains resident in memory, which can happen after an initial access, disk accesses are correspondingly reduced. As with sequential files, you can increase the length of random files. Random files offer you the best compromise between speed and efficient disk usage.

RDOS creates random files by default for any save file (executable program) produced by the Relocatable Loader (RLDR). When the text editors SPEED and EDIT are used to create a file, the result is random block organization.

The CLI command to create a random file is CRAND, as shown in the following example.

```
R

CRAND data.01
```

This command creates file data.01 with zero length, characteristic D, which indicates that it is a random file, and no attributes.

Contiguous Files

Blocks in a contiguous file are organized sequentially on disk; that is, each data block of the file is physically contiguous on the disk, with sequential logical block addresses. To access any block in a contiguous file, RDOS uses the address of the first block and the relative block number. RDOS can thus write or read a contiguous file block using fewer disk accesses than for a sequential or random file. Figure 3.3 shows contiguous block organization. Contiguous files are fixed in length; you specify the file size at creation, and cannot change it thereafter.

All 256 words of a contiguous file block are used for data storage, as in random files. Contiguous files offer the quickest access to data, but the least efficient use of disk space.

RDOS uses contiguous organization for disk partitions, which you create with the CLI command CPART (see the section ahead in this chapter entitled Creating Directories for a discussion of disk partitions). Contiguous organization is also used by RDOS for overlay files, which are auxiliary program routines and are created with the Relocatable Loader.

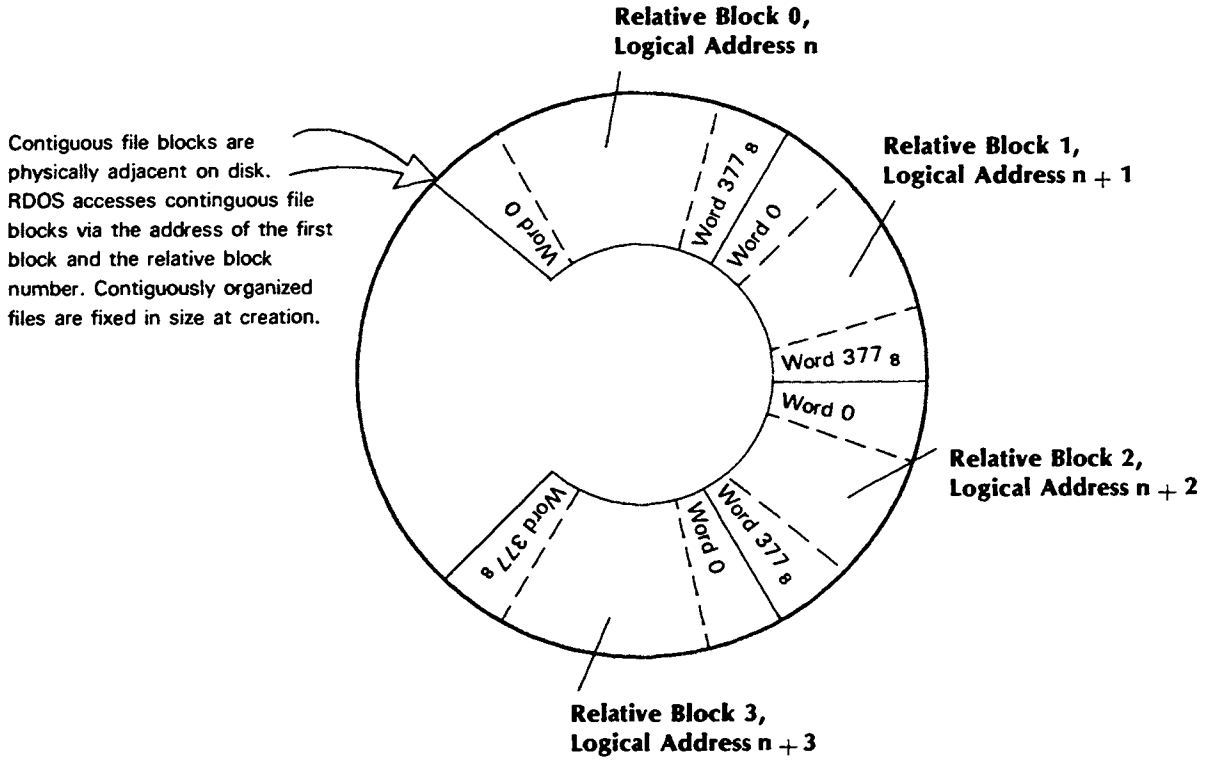


Figure 3.3 Contiguous file block organization ID-00330

To create a contiguous file, use the CLI command CCONT, as shown in this example:

```
R

CCONT names 20 cities 50
```

This command creates file names with 20 zeroed disk blocks and file cities with 50 zeroed disk blocks. The two files have the characteristic C, indicating they are contiguous files, and no attributes. (See the section later in this chapter called Protecting and Identifying Files for a discussion of characteristics and attributes.) You cannot expand or reduce the block counts of contiguous files.

Managing Sequential, Random and Contiguous Files

If you find that a particular block organization is unsuitable for your purposes or that your data has outgrown a contiguously organized file, you can transfer data to a new file. Create an empty file with the appropriate CLI command (CREATE, CRAND, or CCONT), then use the CLI command XFER to copy the contents of the old file into the new.

RDOS handles most file I/O through *system buffers*, areas of reserved operating system memory. See Chapter 6 for a discussion of file I/O. Direct block I/O, which does not use system buffers, is the fastest form of file I/O and can be used with contiguously organized files.

Preparing Disks

You divide disks into logical segments called directories in order to protect your files and facilitate access to them. For instance, groups of files with functions or users in common, such as application database files, can be placed in a single directory, giving you easy maintenance and controlled access.

Before you give a disk a directory structure, you must run several procedures on it to prepare it for use under RDOS. The procedures are known as hardware formatting and disk initialization, (sometimes called software formatting).

Hardware Formatting

Hardware formatting sets up parameters for address and data areas on the surfaces of a disk; the parameters are recognized by the hardware and are transparent to the user. Disks must be hardware formatted before they are initialized, or software formatted. Disks obtained from Data General are already hardware formatted. You can use Data General's DTOS, the Diagnostic Operating System, to hardware format a disk as well as to test computer systems. For information on DTOS, see *How To Use DTOS* (DGC No. 015-000103) or *DTOS Summary* (DGC No. 015-000082).

Disk Initialization

Disk initialization, also known as full initialization or software formatting, prepares a disk to run under a particular operating system. For RDOS, full initialization fits a disk with the framework for the RDOS directory and file structure.

You use the program DKINIT to initialize RDOS disks as well as to maintain already-initialized disks. DKINIT is a *stand-alone* program; that is, it executes alone, without the operating system. DKINIT, which is documented in *How to Load and Generate RDOS* (DGC No. 069-400013) enacts these functions in initialization:

- Analyzes the surfaces of the disk for block integrity;
- Sets aside bad blocks in the *bad block table* to remove them from use;
- Reserves a *remap area* of blocks to be used in place of bad blocks;
- Writes a *disk identification* block on the disk, accessible to the operating system.

The bad block accounting and disk identification functions of DKINIT support the later creation, with the CLI command INIT/F, of two administrative directories on the disk, MAP.DR and SYS.DR.

MAP.DR and SYS.DR

By separating the usable blocks from the bad blocks on the disk, DKINIT prepares the disk for MAP.DR, the *master allocation directory*. Through MAP.DR, a file of contiguous bits corresponding to each block on the disk, RDOS audits disk usage by setting to 1 those bits whose corresponding blocks are in use (part of a file), and setting to 0 all bits whose corresponding blocks are free for use.

The disk identification block written by DKINIT includes such information as the disk's model and controller type and the disk's *frame size*. The frame size is the number of blocks to be allocated for SYS.DR, the *system file directory*, which contains *user file definitions* (UFDs) that enable RDOS to locate files on the disk. In DKINIT, frame size is set either by default, according to the type of disk, or by the user, according to the number of directories needed on the disk.

The first word in each block of SYS.DR contains the number of files listed in the block; the remaining words contain the UFDs. The UFD words contain, among other information, a file's name, extension, size, attributes, characteristics, the address of the first block, and the disk device code. SYS.DR is organized as a random file, thus minimizing disk access time.

Both MAP.DR and SYS.DR are written on a disk with the CLI command INIT/F. The global switch /F, for full initialization, destroys any existing file structure on a disk as it writes a new MAP.DR and SYS.DR; for this reason, INIT/F is to be used with caution. The CLI issues a *CONFIRM?* inquiry when the global /F switch is used with INIT. If you type a Y, the CLI will output a *YES* and proceed with full initialization. If you type any other character, the CLI issues a *NO* and cancels the INIT command.

INIT/F is required after DKINIT's initialization functions (but not disk maintenance functions) have been performed on a disk.

Creating Directories

A disk that has been fully initialized and newly introduced to the system has essentially one directory, called the *primary partition*. The primary partition has the name of the disk itself, for example DP0, and comprises all area available on the disk for user and system storage. In order to allocate disk usage, you can then divide the disk into other directories called *secondary partitions* and *subdirectories*.

Primary partitions can contain files, secondary partitions, and subdirectories. Secondary partitions and subdirectories, which you name according to RDOS filename rules, are files containing other files. A secondary partition is a *fixed-length* directory that can contain files and subdirectories. A subdirectory is a *variable-length* directory that can contain only files.

A file, then, can reside in a primary partition, secondary partition or subdirectory. Files within any one directory (primary partition, secondary partition, or subdirectory) must have unique filenames within that directory. Figure 3.4 illustrates the relationship between partitions, subdirectories and files on a disk.

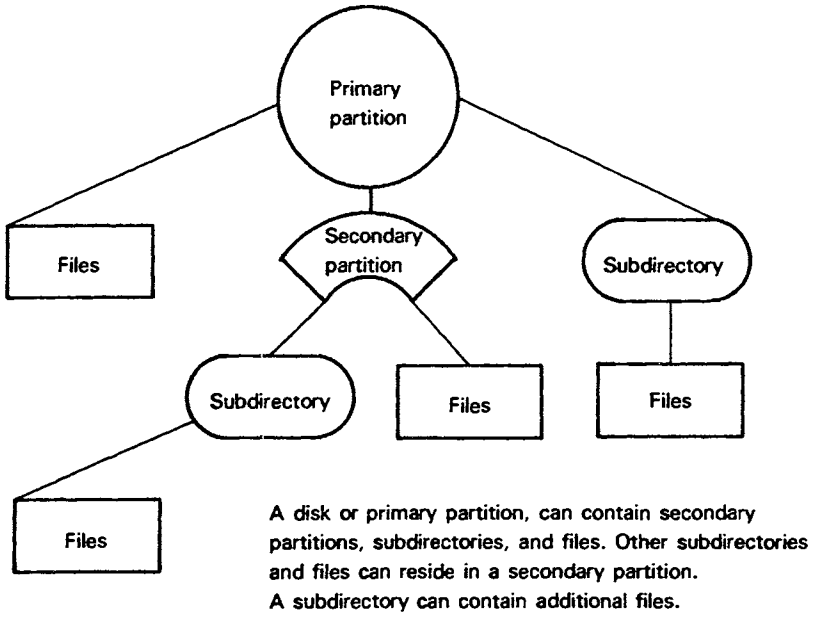


Figure 3.4 Directory hierarchy ID-00331

Directory files are unusual in that they contain other files, but they are no more privileged than a simple data file. All but the primary partition can be deleted.

You use CLI commands or system calls to create either type of directory; with either method, the extension .DR is affixed by the CLI to the name your have assigned. The number of directories you can place on a disk is limited only by available space on the disk.

Caution: Because of confusion that may arise in referencing or releasing directories in a disk directory structure, RDOS does not permit two directories of the same name to be initialized at any one time. (See the section ahead on Initialized Directories.) Therefore, we strongly recommend that each partition and subdirectory on a system be given a unique name.

We recommend that you use CLI commands, rather than system calls, for creating directories because of faster and simpler error interpretation. Once a disk structure is created, user programs can manipulate directories and files through system calls.

Secondary Partitions

Secondary partitions are fixed-length portions of the primary partition. Organized by RDOS as contiguous files, secondary partitions have the advantage of quick access time, since their blocks are contiguous on disk and of a fixed number. The CLI command CPART creates secondary partitions, thus:

```
R

CPART parta 256
```

This command creates a secondary partition called parta.dr with 256 contiguous blocks. Note that a secondary partition must have 48 or more disk blocks, in an integer multiple of 16. If the block count entered in the CPART command is not an integer multiple of 16, the system truncates the number to the next lower multiple.

Subdirectories

Subdirectories are variable length directories that can be created within primary or secondary partitions. They are random files, with a file index block pointing to each file in the subdirectory, thus offering efficient disk access. As random files, subdirectories grow when files are added and shrink when files are deleted. The CLI command to create subdirectories is CDIR, for example:

```
R

CDIR subdirx
```

This command creates a subdirectory called subdirx.DR. Note that the CDIR command creates a subdirectory in the *current directory*. Current directories are explained in the following section.

Referencing Directories and Files

The division of a disk into partitions and subdirectories forms a hierarchical structure. Figure 3.5 shows the formation of a directory structure on a disk. Because a file may exist in one of many subdirectories and a subdirectory may reside in one of many partitions, RDOS provides a variety of methods to reference directories and files. The RDOS protocol includes *directory specifiers*, *initialized directories*, *current directories*, the *master directory*, *links*, and *templates*.

(a)

The primary partition constitutes all area available on a disk for system and user storage, before and after disk division.

(b)

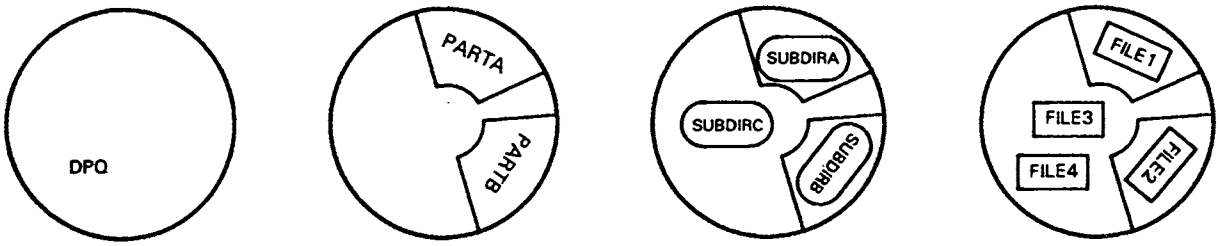
Secondary partitions of fixed sizes can be created within the primary partition.

(c)

Subdirectories of variable length can be created within primary or secondary partitions.

(d)

Files can be created within primary and secondary partitions and subdirectories.



(e)

Primary partition DPO contains two secondary partitions and a subdirectory. Each secondary partition contains a subdirectory. Files reside within each directory on this disk.

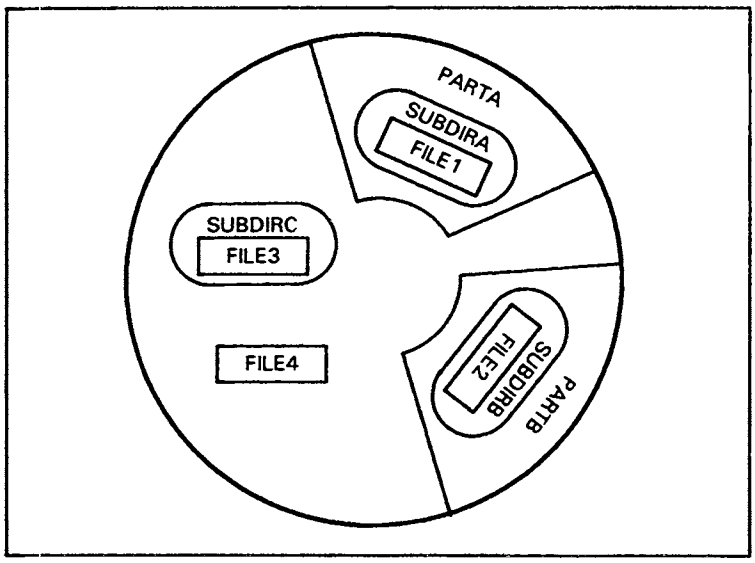


Figure 3.5 Disk directory structure

ID-00332

Directory Specifiers

Directory specifiers point to an object directory or file. To reach a file located in a subdirectory of a secondary partition for example, you list each level in the path to the file in order to create a directory specifier. Device, directory, and file names are separated in a directory specifier by a colon (:). In the command

R

PRINT dp0:parta:subdirx:myfile

the disk, DP0, is named first, then the secondary partition parta, and then the subdirectory subdirx, and then the file myfile.Note that the .DR extensions are omitted from partition and subdirectory names in directory specifiers. The effect of the above command is to PRINT file myfile residing in subdirx.

Because devices are named and referenced as files under RDOS, the inclusion of a disk name in a directory specifier is legitimate. For the same reason, you can include other devices, such as a magnetic tape transport, in a directory specifier. The term directory specifier, then, should be understood more generally as a file specifier.

When you include a disk name in a directory specifier, the specifier is sometimes called a *global specifier*. Global specifiers, as in the last command example, are usually necessary only when more than one disk is on a system.

Initialized Directories

As explained in the section entitled Preparing Disks, the CLI command INIT/F fully initializes a disk for the operating system, readying it for I/O. The INIT command, without the global/F switch, also acts to initialize a secondary partition or subdirectory that you have created, allowing access to the files in either. This process is known as *directory initialization*. Without directory initialization you cannot reach a directory's files for I/O.

During system generation, when the operating system is created with SYSGEN, you specify the maximum number of directories, up to 64, that can be initialized at any one time. See *How to Load and Generate RDOS* (DGC No. 069-400013) for directions.

An example of the command is

```
R

INIT parta
```

This opens partition parta to access by the user. All files in parta can be read or modified, and new files can be added. Another example shows the use of a directory specifier:

```
R

INIT parta:subdirx
```

This command initializes both partition parta and subdirectory subdirx, but does not initialize any other subdirectory that might reside in parta. Once a directory is initialized, the system remembers where it is, and you can reference it directly. For example you can now access myfile in subdirx without including parta in the directory specifier, thus:

```
R

TYPE subdirx:myfile
```

Only one directory of a given name can be initialized at any one time. For example, you cannot now INIT partb:subdirx until you RELEASE parta:subdirx. See the section ahead in this chapter entitled Releasing Directories.

Note that the INIT command can also open and control magnetic tapes. For information on magnetic tapes, refer to Chapter 6.

Current Directories

A current directory is both initialized (ready for I/O) and locatable without a directory specifier. That is, RDOS searches the current directory for any file named in a command, *unless* ordered to search elsewhere with a directory specifier. While many directories can be initialized at any one time, only one directory can be current.

Directories are made current with the command DIR (system call .DIR), which performs the INIT function as well. For example:

```
R

DIR dp1:parta:subdirx
```

With this command, subdirectory Subdirx becomes the current directory, which RDOS searches upon receiving any simple file command. You can determine the current directory with the command GDIR. When you DIR to another directory, the previously current directory remains initialized.

The Master Directory

The name *master directory*, sometimes called master device, refers to the directory on which the RDOS system software resides and from which the system is bootstrapped. The system software includes the operating system, the CLI, and the system utilities, library and bootstrap program. The master directory can also include administrative files of the operating system, such as spooling or tuning files.

The master directory files are placed on the primary partition of a disk when you first load your RDOS system. See the manual *How to Load and Generate RDOS* for information on the creation of the master directory. You can later move files from the primary master directory to a secondary partition.

The term master directory is relative in that it can refer to the directory in which the currently running operating system is stored. That is, the CLI command MDIR returns the name of the directory from which you booted your system, whether the primary or any secondary partition.

When you bootstrap your system, the master directory is the current directory and the only directory that the operating system recognizes, that is, the only initialized directory. Other directories must be initialized with INIT to be accessible for I/O, and, if desired, another directory can be made the current directory with the DIR command.

Releasing Directories

Directories are released from the state of being initialized or current with the CLI command RELEASE. Releasing a directory does not delete it or any of the files in it. When an initialized partition or subdirectory is released, it is closed to I/O. When a current directory is released, the master directory becomes the current directory until another directory is made current with DIR. For example,

```
R

RELEASE subdiry
```

removes subdiry from system recognition.

When the master directory is released, the system is shut down; this is the standard method for closing the system. An example of the command is

```
R

RELEASE DP0
```

On this command, RDOS displays the message

```
MASTER DEVICE RELEASED
```

and closes the system.

The RELEASE command is also used to control magnetic tapes, as discussed in Chapter 6.

Links

A link is a file that allows a user to access any disk file or device from any directory without using directory specifiers. Links save disk space by allowing users in different directories to access a single copy of a commonly used disk file; an application database in a user directory; or the editor SPEED.SV in the master directory. The object file of a link connection is called a *resolution file*. For example,

```
LINK SPEED.SV DP0:SPEED.SV
```

This creates a link file in the current directory called SPEED.SV; the link file is empty but, by its name, points to the program file SPEED.SV in the master directory. With the link set up, the only command needed to edit a file with SPEED is

```
R

SPEED filename
```

Figure 3.6 shows a directory structure with link entries and a resolution file.

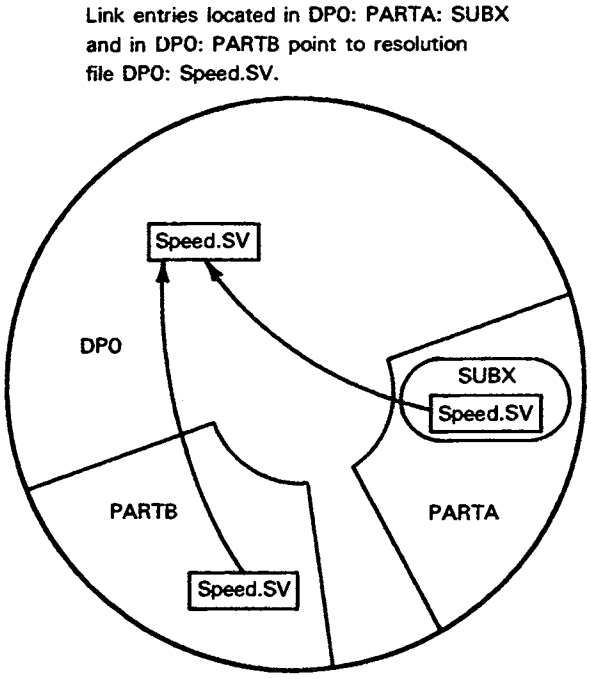


Figure 3.6 Directory structure with link entries ID-00333

It is recommended, but not required, that link entries and their resolution files share the same name. A link entry with a name differing from its resolution file is called an *alias*.

To create or use a link, the directory containing the resolution file and all intermediate directories in the link must be initialized.

Removing Links

Link files in the current directory are deleted with the CLI command UNLINK. For example,

```
R

DIR subdirb

R

UNLINK edit.sv
```

This deletes the link entry in subdirb but does not delete the resolution file.

> **CAUTION:** *If you use the DELETE command on a link file, the link file remains, but the resolution file is deleted, attributes permitting.*

Templates

The use of directories as well as the careful choosing of filenames and extensions can simplify classification and manipulation of groups of files. In addition, you can control groups of files by means of *templates*, special symbols representing parts of filenames. The RDOS template characters are

* (asterisk) Represents any single character, except a period (.), in a filename or extension. For example, LIST Dat* would list all four-character filenames beginning with Dat and without extensions. The command DELETE * would delete all single-character filenames.

- (hyphen, dash) Represents any string of characters, except period (.), in a filename or extension. For example, LIST D- would list all filenames beginning with D and without extensions. The command DELETE text.- would delete all files named text, with any extension.

You can mix the two templates in a command line:

```
LIST s-.$*
```

would list all filenames beginning with S and having two-character extensions beginning with a dollar sign.

There are several restrictions on the use of templates. First, templates cannot be used with a directory specifier (a pointer to a location outside the current directory). (Templates can, of course, be used on directory files within the current directory; for example LIST -.DR to see all the directories on this level.)

Second, templates can be used only with these CLI commands: LIST, DELETE, BUILD, DUMP, LOAD, MOVE, and UNLINK. LIST, DELETE and UNLINK have already been introduced; DUMP and LOAD pertain mainly to magnetic tape and are covered in Chapter 6. BUILD and MOVE are discussed below.

BUILD Command

The BUILD command copies input filenames, but not file contents, into a single file. An example of the command is

```
R

BUILD quarter april.- may.- june.-
```

This creates a file called quarter consisting of the filenames of all files named april, may and june, with any extension.

MOVE Command

The MOVE command copies files from the current directory to a specified directory, preserving name, length, attributes, creation time, and access time. For example

```
MOVE/V/K dp1:database text*.-
```

This directs RDOS to move all five-character filenames beginning with text into the database directory on disk DP1. The /V global switch causes RDOS to verify the move by listing on the terminal the names of the files moved. The /K switch excludes link files from the move.

Protecting and Identifying Files

RDOS provides for file protection and identification through *file attributes* and *file characteristics*. Attributes and characteristics are stored in a file's UFD.(UFDs are described earlier in this chapter in the section entitled MAP.DR and SYS.DR.) Attributes are assigned by you or by RDOS. Characteristics are assigned by RDOS and cannot be changed.

File Attributes

The attributes of a file protect it by permitting or restricting reading, writing, RENAMing, DELETing or LINKing. Users can assign and change attributes of a file with the CLI command CHATR. Link resolution files can be similarly protected with the command CHLAT. RDOS assigns certain attributes to nondisk files that cannot be changed. Table 3.4 lists file attributes.

Attribute	Meaning
P	*Permanent* file, cannot be DELETEd or RENAMEd.
S	*Save* (.SV) file, assigned by RLDR, without which program cannot be executed.
W	*Write-protected* file, cannot be edited or modified in any way.
R	*Read-protected* file, cannot be TYPEd or read in any other way.
A	*Attribute-protected* file, attributes cannot be changed with CHATR, including attribute A.
N	*No resolution* permitted, file resists links.
? (question mark)	First user-definable attribute.
& (ampersand)	Second user-definable attribute.

Table 3.4 File attributes

Clearly, attributes P and A are very powerful and should be assigned with caution. You cannot delete a file that has the P and A attributes without full initialization, which destroys the existing directory structure and data.

In the CHATR command, attributes are arguments and are preceded by a plus sign (+) to add the attribute, or a minus sign (−) to remove the attribute. For example

R

```
CHATR mysecret +R-W
```

This command allows anyone to read mysecret but not to alter it.

You can assign the question mark or ampersand attributes (? or &) to files, supplying your own definitions. The user-definable attributes are useful in classifying files.

File Characteristics

File characteristics identify the file block organization (sequential, random, or contiguous) of the file and indicate whether or not a file is a directory. The characteristics of a file are assigned by RDOS when the file is created, and you cannot alter them. You can inspect the characteristics of a file with the LIST command switches /A or /E. Characteristics are listed in Table 3.5.

Characteristic	Meaning
D	Randomly organized file (all program (.SV) files must have D characteristic).
C	Contiguously organized file.
L	Link file.
T	Partition file. (being contiguous, all partitions also have C characteristic).
Y	Directory file. (either partition or subdirectory).

Table 3.5 File characteristics

Chapter 4

SPEED and EDIT Text Editors

RDOS offers two text editors, Superedit (SPEED) and Text Editor (EDIT). Both allow you to open, close, store, and modify text files. SPEED and EDIT are character-oriented in that their commands work with character sequences. For example, you can use editor commands to search for or change character sequences without knowing precisely where they are in the text.

During an editing session you work in the edit buffer, a storage area in memory where editor commands modify your file. An editing session usually consists of:

- Using editor commands to open input and output files;
- Entering text into the edit buffer, or modifying text files called in from disk;
- Sending the edit buffer contents to the output file;
- Closing the input and output files;
- Storing the new file and the original, backup (.BU) file on disk.

Each editor has a prompt that appears when the editor is ready to receive your commands and a character pointer (CP) that marks the current editing position in the file. Each editor has commands that change the location of the CP within a file.

Commands to the editors consist, for the most part, of single characters and numbers combined with characters. Text you want to insert, change or delete is included in the command line, or indicated with appropriate commands. You can string several commands together on one command line. The editors execute commands sequentially, from left to right, in a multiple command line. An error message informs you when you enter an incorrect command anywhere in a command line.

You delimit commands to SPEED and EDIT with the Escape key, marked ESC on your terminal and echoing as a dollar sign ($) on your screen. You use a single Escape to separate two or more commands in one command line, and a double Escape (ESC ESC) to terminate a command line. In command examples, we show the command delimiters as $ for single Escape and $$ for double Escape.

Files created with either editor are randomly organized.

This chapter introduces each editor and guides you through a hands-on session for each, using basic commands that work in a similar fashion for both. In their full capabilities, however, the editors differ considerably. SPEED is the more advanced and powerful of the two. For a complete description of each editor refer to *RDOS/DOS Superedit Text Editor* (DGC No. 069-400017)and *RDOS/DOS Text Editor* (DGC No. 069-400016).

Typing Mistakes

Three terminal functions help you to remedy typing mistakes in your command lines (commands to the editor allow you to correct mistakes in your text).

- Delete (DEL) or Rubout key (hereafter referred to as DEL)
- CTRL-X
- CTRL-A

DEL and CTRL-X allow you to correct a mistake mode while typing a command, before you execute it with a double Escape.

Use the DEL key to delete the previous character in the command line. When you use DEL, your terminal displays the deleted character. Press DEL repeatedly to delete some or all of your command back to the beginning of the current line. Note that you can delete both commands and intended text; DEL does not delete the prompt.

Use CTRL-X to delete the entire current command line. This is equivalent to pressing DEL to delete each character back to the beginning of the line. CTRL-X deletes the current line only.

Use CTRL-A to cancel a command that the editor is executing after you have entered a double Escape. CTRL-A is most useful with complex commands; with the basic commands we describe in this chapter, you may not be quick enough to stop the whole command from being executed.

SPEED

This session introduces you to basic SPEED commands. The NOVA version of SPEED is called Nspeed, and is identical to SPEED; commands introduced here refer to both.

The SPEED prompt is an exclamation point (!), which appears on your screen when you first run SPEED and when SPEED completes a command. SPEED's Character Pointer (CP), which denotes your editing position, appears as an up-arrow in parentheses (↑) when you use a Type command. SPEED allows you to create a text file containing both upper-case and lower-case characters, though commands must be typed in uppercase. If you type an incorrect command in a command line, SPEED executes only those commands to the left of the invalid command, displays an error message, and repeats the command line.

Advanced features of SPEED allow you to work with multiple edit buffers and multiple input and output files. You can also create macroinstruction (macro) files to store one or more commands that you want to use repeatedly. When you call the macro filename in a SPEED session, the name becomes a command to execute the series of instructions.

Table 4.1 summarizes basic SPEED commands in the order in which they are introduced here, and Table 4.2 shows variations of the T command.

Command	Function
I	Insert text.
T	Type text*.
J	Move CP to start of buffer.
ZJ	Move CP to end of buffer.
*n*J	Jump CP to start of *n*th line of buffer.
[-]*n*L	Move CP to start of *n*th line from current position.
[-]*n*M	Move CP specified number of characters.
S	Search for string, relocate CP after string.
C	Search and change or delete string, relocate CP after string.
K	Delete current line from start to CP.
[-]*n*K	Delete specified number of lines.
[-]*n*D	Delete specified number of characters.
UE	Update file.
US	Update file, store .BU file.
H	Halt SPEED, return to CLI.

Table 4.1 Basic SPEED commands

*See Table 4.2 for T Commands

Starting SPEED

We assume that you have created in your directory a link named SPEED.SV that points to SPEED.SV in the master directory. When you work in your own directory, files you create with SPEED reside in your directory. See Chapter 3 for information on creating links.

To access SPEED from the CLI, you type SPEED filename (NSPEED filename on a NOVA computer), which either creates a new file or opens an existing file for editing. To illustrate the SPEED commands, we will create and edit part of a FORTRAN IV source program called Figure that calculates mortgage payments. We begin by entering the CLI command that executes SPEED, plus the name of the file we want to create:

```
R

(N)SPEED FIGURE ⟨CR⟩

CREATING NEW FILE

!
```

SPEED signals that it is creating the new file, and displays its prompt, the exclamation point. When you open a file that already exists, SPEED copies the first page of the file into the edit buffer. A page is a unit of text followed by a formfeed (new page) character; the formfeed is inserted by SPEED when you save text. Advanced SPEED commands (not covered in this chapter) allow you to define pages in a file.

Inserting Text (I)

The Insert (I) command inserts text, always *to the left of the Character Pointer*. When you open a file, the CP is always located at the first position in the buffer. In this session we insert text, including a mistake we later correct.

```
!IREAL INT,ITD,LB ⟨CR⟩

TYPE "ENTER AMOUNT, RATER, YEARS" ⟨CR⟩

TYPE "AND, 0 FOR SUMMARY OR 1 FOR"$$

!
```

The insertion starts with the I command and ends with the double Escape ($$); all the text as shown is placed by SPEED in the buffer and appears on the screen. After each insertion, SPEED locates the CP *after the last inserted character*. In this example, the insert stops at the end of the last line. To insert text on a new line, you can strike ⟨CR⟩ after the I command, then your text.

Generally, you should not insert more than ten lines of text with one I command. As a precaution against losing work, you can break up a long insertion with several I*text*$$ sequences.

Displaying Text (T)

As you work, you can display lines with the Type command (T). We recommend frequent use of the T command in order to verify your work when you are learning the editor.

The T command has many variations, as shown in Table 4.2. The T commands act relative to the beginning of the buffer or relative to the position of the CP. Note that *T commands do not change the position of the CP*. You can display many lines forward or backward in your text, but the CP remains where you left it. The T commands act on lines or on characters.

Command	Function
#T	Type entire buffer contents.
T	Type current line, showing CP.
0T	Type current line from beginning to CP.
*n*T	Type *n* lines following CP.
-*n*T	Type *n* lines preceding CP
m,*n*T	Type from (*m* + 1) character in buffer to *n*th character.

Table 4.2 SPEED T commands

Displaying the Buffer Contents (#T)

The #T command displays the entire buffer contents. For example,

```
!#T$$
```

displays all 3 lines of text so far inserted in this session.

Displaying Lines (T, *n*T)

The simple T command displays the current line, where the CP is positioned, showing the CP as an up-arrow (↑). For example,

```
!T$$
TYPE "AND, 0 FOR SUMMARY OR 1 FOR" ↑
```

The CP is located at the end of the last line you inserted.

The 0T$$ command (zero T) types the current line from the beginning up to the location of the CP. This is useful for locating your editing position in a line.

To display more than one line relative to the CP, you use a positive or negative numeric argument with T, in the form *n*T$$ or -*n*T$$. With a positive number, SPEED types the lines from the CP forward through the next *n* carriage returns. With a negative argument, SPEED types the contents of the *n* lines preceding the current line, plus the contents of the current line up to the CP. For example,

```
!-2T$$
```

displays

```
REAL INT,ITD,LB
TYPE "ENTER AMOUNT, RATER, YEARS"
TYPE "AND, 0 FOR SUMMARY OR 1 FOR"( )
```

Displaying Characters (*m*,*n*T)

The *m*,*n*T command acts on characters, not lines, and acts relative to the beginning of the buffer. *m*,*n*T displays the contents of the buffer from the (*m* + 1) character (measured from the first character in the buffer) up to and including the *n*th character. For example

```
!2,8T$$
AL INT,I
```

This command returns the third (2+ 1) to the eighth character.

You can use this command in the form 0,*n*T$$ to type from the first character in the buffer (0 + 1) to the *n*th character. For example,

```
!0,8T$$
REAL INT
```

Moving the CP (J, ZJ, L, M)

The current editing position is always to the left of the CP. You can change the editing position with commands that move the CP. It's a good idea to use a T command to verify your location after you have moved the CP.

As with T commands, the CP-moving commands function relative to the beginning of the buffer or relative to the current line (the line in which the CP is located).

The Jump Commands (J and ZJ)

The J command moves the CP to the start of the buffer, while the ZJ command moves it to the end of the buffer. For example,

```
!JT$$
( )REAL INTD,ITD,LB
```

With this command, the J moves the CP to the start of the buffer, the T types the current line (now the first line in the buffer), and the CP remains at the start of the buffer. You can issue the ZJ command, for example, to move the CP to the end of the buffer in preparation for inserting more text.

You can use the J command with a positive numeric argument to move the CP to the beginning of a line relative to the start of the buffer. The *n*J command moves the CP to the start of the buffer and then jumps it to the beginning of line number *n*. For example,

```
!3J$T$$
( )TYPE "AND, 0 FOR SUMMARY OR 1 FOR"
```

The editor starts counting lines from the beginning of the buffer, then moves the CP to the beginning of line three.

The Line Command (L)

To move the CP to the beginning of the current line, type

```
!L$$
```

To set the CP at the beginning of a line preceding or following its current line, use a numeric argument (*n*) with L, for example -3L$$. A positive *n* instructs SPEED to move the CP *n* lines forward from the current line; a negative *n* moves the CP *n* lines backward.

For example, suppose you insert the following text into the sample file Figure, then move the CP and verify the move with a T command:

```
!I⟨CR⟩

C THIS IS A PROGRAM TO COMPUTE RATES. ⟨CR⟩

C THIS PROGRAM OUTPUTS REAL NUMBERS. ⟨CR⟩

$$

!-1L$T$$

(↑)C THIS PROGRAM OUTPUTS REAL NUMBERS.

!
```

The Move Command (M)

The M command moves the CP by the specified number of characters relative to the CP. A positive numeric argument moves the CP forward *n* characters, while a negative argument moves the CP backward *n* characters. For example,

```
!8M$T$$

C TH(  )IS PROGRAM OUTPUTS REAL NUMBERS.
```

This command moves the CP forward eight characters, and shows the new position of the CP.

Searching Text (S)

One of the most useful editing commands is the Search (S) command, which enables you to find strings (character combinations) anywhere in the current buffer. You specify a string with the S command; SPEED searches forward from the point of the CP, and, if it finds the string, repositions the CP to the position *immediately after* the string specified in your command.

Suppose you want to examine line 2 of the sample file:

```
!J$STYPE $T$$

TYPE (  )"ENTER AMOUNT, RATER, YEARS"
```

SPEED moves to the start of the buffer, searches forward for TYPE, then positions the CP after the string and types the line.

In using the S command, you must type precisely the string you want. If, in this example, you mistakenly specify two spaces in the search string when the text contains only one space, SPEED does not find the string and displays an error message with the incorrect command. For example,

```
!J$STYPE $T$$

ERROR;UNSUCCESSFUL SEARCH

!J$STYPE $T$$

!
```

When a search is unsuccessful, SPEED positions the CP at the start of the buffer.

The S command searches forward from the position of the CP; if you specify a search for a text string located before the CP, your terminal displays the Unsuccessful Search error message.

Searching Text for Change or Deletion (C)

The Search and Change (C) command resembles the S command in that it searches for a string but it also allows you to change or delete the object string.

To search for and change characters with C, you specify the object string and the desired string in this form:

```
!Cstring1$string2$$
```

String1 specifies the string you want to search for and change and string2 specifies the string to take its place. You must separate these two arguments with an escape character ($).

As with the S command, the C command causes SPEED to search forward from the CP. When SPEED finds and changes the string, it places the CP *immediately after* the last character in the new string. If the search string precedes the CP, SPEED displays the *UNSUCCESSFUL SEARCH* error message and positions the CP at the start of the buffer.

For example, the second line of the inserted text contains the word RATER instead of RATE. If you instruct SPEED to move the CP to the beginning of the buffer, change TER to TE, and display the changed line, like this:

```
!J$CTER$TE$T$$

TYPE "ENTE(↑) AMOUNT, RATER, YEARS"
```

SPEED finds the first TER, in ENTER, and changes it to TE. This demonstrates one of the most common errors with the S or C command: searching for a string that is not unique. To correct the mistake and verify the correction, type:

```
!J$CENTE$ENTER$T$$

TYPE "ENTER(↑) AMOUNT, RATER, YEARS"

!
```

To fix RATER, type:

```
!CRATER$RATE$T$$

TYPE "ENTER AMOUNT, RATE ↑, YEARS"
```

You can also use the C command to search and delete a string, following this format:

```
!Cstring$$
```

The editor searches for the specified string, deletes it, and does not replace it with another string.

Deleting Lines and Characters (K, D)

The Kill (K) command and the Delete (D) command permit you to delete lines and characters inserted in your text.

The Kill Command (K)

The simple K command deletes the characters from the CP back to the beginning of the current line, preserving all characters to the right of the CP.

To delete more lines, use the K command with a numeric argument, for example 3K$$ or -3K$$. Though you can use a negative value with K, we recommend only positive values because you can easily lose track of what you are deleting.

With a positive argument, *n*K$$ deletes the lines from the location of the CP in the current line up to and including the next *n* carriage returns. With a negative argument, *-n*K deletes the *n* lines preceding the current line plus the characters up to the CP in the current line. For example,

```
!3K$$
```

deletes everything to the right of the CP on the current line, and the following three lines.

The Delete Command (D)

To instruct the editor to delete *n* characters, use the D command with a positive or negative argument, for example 5D$$ or -5D$$. These commands delete the 5 characters before the CP or the 5 characters after the CP.

You can use the M command with a positive or negative argument to move the CP to the position at which you want to delete characters, then use the D command for the deletion.

Closing Files and Leaving the Editor (UE, US, H)

After you finish editing, you can use one of two Update commands, UE or US, to store your file. The UE command incorporates all of the editing changes you have made to the file and creates a new, updated file. The US command does the same thing, additionally saving the original file as a backup file under filename.BU after deleting any older filename.BU. For this sample session, a UE would suffice because Figure is a new file.

The Home (H) command halts the editor and returns you to the CLI. Usually, you use the H command with UE or US, in the form:

```
/USH$$
```

If you use H without UE or US, you return to the CLI immediately, without writing out the contents of the edit buffer or copying the remainder of the input file. Your original file is intact on disk. The H command alone (without UE or US) would be useful if you realize that you have made some substantial editing errors, and that it would be easier to start again than to try to correct the errors by working on the updated file.

Any command string starting with an H terminates the editor and returns you to the CLI, and all work done in the edit buffer is lost. Therefore, be careful of starting a command string with H. For example, if you mistakenly begin a text string with HELP without preceding it with I, S, or C as in this example:

```
HELP RETRIEVE MY EDIT BUFFER$$
```

The editor terminates, returns you to the CLI, and you lose all the work you did in the edit buffer during that session.

Reopening a File for Editing

Until you learn your editor well, you should update your file often with the USH$$ sequence in order to avoid losing your work. You can re-enter SPEED with the command to start it, accompanied by the filename, in this format:

```
R

(N)SPEED filename ⟨CR⟩
```

To add text to the end of the buffer of a reopened file, type the ZJ command before the I command so that SPEED places your new inserts at the end of the previous text. If you do not move the CP to the end of the buffer, SPEED inserts the text at the beginning of the file.

If you lose text from the edit buffer after executing a command, type USH$$ to update the file and save the original version. You can examine both the current version and the original version (.BU) with either the appropriate Type (T) command or the CLI TYPE command.

EDIT

This session introduces you to basic EDIT commands. EDIT has a multiuser version called Medit whose commands are nearly identical to those of EDIT. Refer to the *RDOS/DOS Text Editor* manual for a description of Medit.

EDIT prompts you with an asterisk (*), which appears when you execute EDIT and when it completes commands. EDIT treats both upper- and lower-case letters as if they are upper-case. In EDIT you can change the location of the character pointer (CP), but there is no command to let you see its position. You can work with one edit buffer and one input or output file at a time.

Table 4.3 summarizes basic EDIT commands in the order in which they are introduced.

Command	Function
UN	Create and open file.
UY	Open existing file.
I	Insert text.
T	Type entire buffer contents.
1T	Type current line.
*n*T	Type *n* lines forward from CP.
B	Move CP to start of buffer.
Z	Move CP to end of buffer.
L	Move CP to start of current line.
[-]*n*L	Move CP to start of specified line.
*n*J	Jump CP to start of *n*th line of buffer.
[-]*n*M	Move CP specified number of characters.
S	Search for string, relocate CP after string.
C	Search and change or delete string, relocate CP after string.
K	Delete current line.
[-]*n*K	Delete specified number of lines.
[-]*n*D	Delete specified number of characters.
UE	Update file.
US	Update file, store .BU file.
H	Halt EDIT, return to CLI.

Table 4.3 Basic EDIT commands

Starting EDIT and Opening a File (UN, UY)

We assume that you have created in your directory a link named EDIT.SV that points to EDIT.SV in the master directory. When you work in your own directory, files you create with EDIT reside in your directory. See Chapter 3 for a discussion on creating links.

To illustrate EDIT commands, we will create and edit part of a FORTRAN IV source program called Figure that calculates mortgage payments.

To start EDIT from the CLI, type:

```
EDIT ⟨CR⟩
```

After a short delay your terminal displays the EDIT asterisk prompt (*). To open the sample file, Figure, use the Update New (UN) command, ending with double Escape ($$) to terminate the command. Note that you do not leave a space between the UN command and the filename, and that no carriage return (⟨CR⟩) is necessary after $$.

```
*UNFIGURE$$
*
```

EDIT creates an empty input file called FIGURE in the edit buffer and your terminal again displays the EDIT prompt.

If you want to reopen a file that already exists, use the UY command in the format:

```
*UYfilename$$
```

When you are in the CLI, you can open an existing file when you start EDIT, using this command format:

```
EDIT filename ⟨CR⟩
```

EDIT opens the file and reads (copies) the first page of the file into your edit buffer. A page is a unit of text followed by a formfeed (new page) character inserted by EDIT when you save text. Advanced EDIT commands (not covered in this chapter) allow you to define pages in a file.

Inserting Text (I)

When you open a file, the character pointer (CP) is always located at the first position in the buffer, the point at which you now insert text. The I command inserts the text at the CP. Insert the text as shown, mistakes and all. We will show you how to correct mistakes with other commands.

```
*IREAL INT,ITD,LB ⟨CR⟩

TYPE "ENTER AMOUNT, RATER, YEARS" ⟨CR⟩

TYPE "AND, 0 FOR SUMMARY OR 1 FOR" ⟨CR⟩

$$

*
```

After each insertion, the CP is located after the last inserted character.

Generally, you should not insert more than ten lines of text with one I command. As a precaution against losing your text, you can break up long insertions with several I*text*$$ sequences.

Displaying Text (T)

It is good practice to review each insertion, addition, or change after you make it. The Type (T) command permits you to display your work on your terminal. You can use T alone or in one of several variations. Note that *no T command moves the CP*.

With EDIT T commands introduced in this chapter you can display the entire buffer contents, the current line, and multiple lines in the buffer.

To display the entire buffer contents, type:

```
*T$$
```

EDIT then returns the three lines of text inserted so far.

To display the line on which you are located, type:

```
*1T$$
```

With this command, EDIT displays the current line, the line where the CP is located.

With the *n*T command you can examine multiple lines ahead of the CP. The T command does not take a negative argument in EDIT. In order to type lines preceding the CP, you can use a command that moves the CP, and then a T command. For example, move the CP to the beginning of the buffer with the B command and use the *n*T command to display lines:

```
*B$3T$$
```

Your terminal displays the first three lines of text in the buffer. To display the next *n* lines, use the Line (L) command to move the CP to the line at which you want the display to start, in this format:

```
*nL$$
```

Then type *n*T$$ for however many lines you want to see.

For a description of the B and L commands, refer to the next section, Moving the CP.

Moving the CP (B, Z, J, L, M)

The current editing position is at the point just after the location of the CP. You can change the editing position by moving the CP, as described in the next sections. Use your T command to display the line you are on after you move the CP. This helps you to check both the accuracy of your commands and the location of the CP after a command is executed, though the CP itself is not displayed.

The B and Z Commands

The Beginning command (B) moves the CP to the start of the buffer. For example,

```
*B$1T$

REAL INT,ITD,LB

*
```

The Z command, for example Z$$, moves the CP to the end of the buffer.

The L Command

To move the CP to the beginning of the current line, type:

```
*L$$
```

To set the CP several lines forward or backward from its current position, use a numeric argument with L, in the form *n*L. A positive *n* instructs the editor to move the CP *n* lines forward from the current line; a negative *n* instructs the editor to move the CP *n* lines backward.

For example, after you insert text, use the T command to check the position of the CP. Suppose you insert these lines into Figure, using I, followed by the text.

```
*IC THIS IS A PROGRAM TO COMPUTE RATES. (CR)

C THIS PROGRAM OUTPUTS REAL NUMBERS. (CR)

$$
```

The CP is located at the end of the text you insert. To move it back one line, type:

```
*-1L$T$$
```

The J Command

To move the CP to a position just before the first character in a given line, use the Jump (J) command with a positive numeric argument. Jump (J) moves the CP to the start of the buffer and then jumps it to line number *n* from the start of the buffer. Type the command, using this format:

```
*nJ$$
```

For example, to move the CP from the end of line 1 to the beginning of line 5, type:

```
*5J$$
```

The editor starts counting from the beginning of the buffer, then moves the CP to the beginning of line five. Use the T command to display the line.

> **NOTE:** *The L command moves the CP forward or backward from any point in the file. the J command moves the CP to the beginning of the file and moves forward* n *lines.*

The M Command

To move the CP to the left or right on a line, use the Move (M) command with a numeric argument, specifying the number of characters to move, in the form *n*M. To move the CP to the right, use a positive number for *n*, for example, 2M$$; to move the CP to the left, use a negative argument, for example -5M$$. If the CP is located at the beginning of the line:

```
REAL INT,ITD,LB
```

and you issue the command:

```
*2M$$
```

Then the CP would be located between RE and AL INT,ITD,LB.

To restore the CP to its orginal position and display the line, type:

```
*-2M$1T$$

REAL INT,ITD,LB
```

You can use M to move the CP past a carriage return character (⟨CR⟩) into another line by specifying a numeric argument that exceeds the number of characters in the line. For example, type 25M$$ and use your Type (T) command to see where the CP lands.

Searching Text (S)

One of the most useful editing commands is the Search (S) command. This command enables you to find strings (character combinations) anywhere in the current buffer. You specify a string with the S command; EDIT searches forward from the point of the CP and, if it finds the string, repositions the CP after the last character in the string.

Assume that you want to examine the second line of text. With the command

```
*B$$STYPE $1T$$

TYPE "ENTER AMOUNT, RATER, YEARS"
```

EDIT moves the CP to the start of the buffer and searches for the string TYPE, then displays the line. The CP is located just after the space following TYPE.

In using the S command, you must type precisely the string you want. If, in this example, you mistakenly insert two spaces in the search string while only one space appears in the text, EDIT does not find the string and the console displays an error message, followed by the incorrect command. For example, if you type:

```
*B$STYPE $1T$$

STR NOT FOUND

??B$STYPE $1T$$

*
```

When a search is unsuccessful in EDIT, the editor positions the CP at the start of the buffer and displays the following message:

```
STR NOT FOUND
```

The S command searches forward from the position of the CP; if you specify a search for a text string that precedes the CP, your terminal displays the *STR NOT FOUND* error message and locates the CP at the start of the buffer.

Searching Text for Change or Deletion (C)

The Search and Change command (C) resembles the S command in that it searches for a string to change, but it also allows you to change or delete the string. The general format for the C command is:

```
*CSTRING1$STRING2$$
```

String1 specifies the string you want to change, and string2 specifies the string to take its place. You must separate the two arguments with an Escape character ($).

For example, the second line of the inserted text reads RATER instead of RATE. Move the CP to the beginning of the buffer, change TER to TE, and display the changed line in EDIT with this command,

```
*B$TER$TE$1T$$

TYPE "ENTE AMOUNT, RATER, YEARS"
```

The editor finds the first TER (in ENTER) and changes it to TE. This demonstrates one of the most common errors with the S or C command: searching for a string that is not unique. To correct the mistake and verify the correction, type

```
*B$CENTE$ENTER$$
```

To fix RATER, type

```
*CRATER$RATE$L$1T$$

TYPE "ENTER AMOUNT, RATE, YEARS"
```

You can also use the C command to delete a string by typing it in this format:

```
*CSTRING$$
```

The editor searches for the specified string, deletes it, and does not replace it with another string.

Deleting Lines and Characters (K, D)

You can delete lines with the Kill command (K) and characters with the Delete command (D).

The K Command

To delete one or more lines, use the K command with a numeric argument, in the form *n*K$$. For example, to delete three lines from the current CP position (the first line consists of everything to the right of the CP on the current line), type:

```
*3K$
```

To delete the current line, including its ⟨CR⟩, use the command 1K$$.

Although you can use negative or positive values for *n*, we suggest that you use only positive values because it is easy to lose track of what you are deleting. For example, when the CP is in the middle of a line, a negative *n* deletes not only the *n* previous lines, but also deletes the left portion of the current line.

The D Command

The Delete (D) command deletes characters from the text. To instruct the editor to delete characters, use *n*D with a positive argument to delete characters to the right of the CP and a negative argument to delete characters to the left of the CP.

You can use the M command with a positive or negative argument to move the CP to the position at which you want to delete characters.

Closing Files and Leaving the Editor (UE, US, H)

After you finish editing, you can use one of two Update commands, UE or US, to store your file. The UE command incorporates all of the editing changes you have made to the file and creates a new, updated file. The US command does the same thing, additionally saving the original file as a backup file under filename.BU after deleting any older filename.BU.

The Home (H) command halts the editor and returns you to the CLI. Usually, you use the H command with UE or US, in the form

```
*USH$$
```

If you use H without UE or US, you return to the CLI immediately, without writing out the contents of the edit buffer or copying the remainder of the input file. Your original file is intact on disk. The H command alone (without UE or US) would be useful if you realize that you have made some substantial editing errors, and that it would be easier to start again than to try to correct the errors by working on the updated file.

Any command string starting with an H terminates the editor and returns to the CLI, and all work done in the edit buffer is lost. Therefore, be careful when starting a command string with H. For example, if you mistakenly begin a text string with HELP without preceding it with I, S, or C as in this example:

```
HELP RETRIEVE MY EDIT BUFFER$$
```

Then the editor terminates, returns you to the CLI, and you lose all the work you did in the edit buffer during that session.

Reopening a File for Editing

Until you learn EDIT well, you should update your file often with the USH$$ sequence in order to avoid losing your work. You can re-enter EDIT with the command to start it, accompanied by the filename, in this form

```
EDIT filename ⟨CR⟩
```

To add text to the end of the buffer of a reopened file, type the Z command before the I command so that EDIT places your new inserts at the end of the previous text. If you do not move the CP to the end of the buffer, EDIT inserts the text at the beginning of the file.

If you lose text from the edit buffer after executing a command, type USH$$ to update the file and save the original version. You can examine both the current version and the original version (.BU) with either the appropriate Type (T) command or the CLI Type command.

Chapter 5

Advanced Features of the CLI

The CLI offers commands and procedures that can streamline and simplify operations. This chapter explains some of the more sophisticated features of the CLI that can be very helpful in improving work flow.

You can group commonly performed sequences of CLI commands in either an *indirect file* or *macro file*, to be executed at your convenience. Indirect and macro command files serve to simulate a CLI terminal session. Indirect files can also contain text, which can be accessed indirectly. The CLI follows a *command interpretation order* to differentiate between indirect and macro command files, CLI commands, utility execution, and program execution when you issue a command.

Similarly to indirect and macro command files, routines can be submitted for *Batch processing* to the Batch Monitor, which accepts job routines from a variety of devices and sends output to specified disk files or devices.

CLI commands, indirect and macro files, and BATCH routines can elicit current information from the CLI through use of *variables*, CLI directives that return current values. In addition, the CLI MESSAGE command can be used to display text during execution of indirect files or macros. Variables can be used within a MESSAGE text string.

The CLI *command parsing order* is a set of rules by which the CLI evaluates all the components of a command in a certain order. Understanding of the command parsing order, in combination with the command interpretation order, is important in building complex commands and in explaining unexpected results.

Finally, your applications can access the CLI from within an assembly language program in order to execute CLI commands and to pass arguments to the program. Some methods of CLI command execution, including the use of files *CLI.CM* and *COM.CM*, facilitate the assembly language program-CLI interface. The system call *.ERTN*, to return a swapped program to the CLI with error status, can be used to take advantage of the CLI's error-handling properties.

All of the topics covered in this chapter are documented in the *RDOS/DOS Command Line Interpreter* (DGC No. 069-400015)

Indirect and Macro Command Files

Indirect files and macro files comprise sequences of CLI commands grouped in a file; alternatively, indirect files can contain only text. Indirect and macro files function identically when the contents of either are CLI commands; however, when an indirect file contains only text, the text is treated by the CLI as an object of a command. That is, an indirect or macro *command* file is invoked as a *command*, while an indirect *text* file is invoked as an *argument*.

For example, assume that you have two text files, text1 and text2, that you sometimes (but not always) want to output together. Create a file called alltext consisting of the filenames text1 and text2, in this manner:

```
R

BUILD alltext text1 text2
```

When you inspect alltext, you find that it contains only the filenames:

```
R

TYPE alltext

text1,text2

R
```

But when you invoke *alltext* indirectly, as an argument, the CLI accesses the contents of *alltext*, thus:

```
R
TYPE @alltext@
(contents of text1)
.
.
.
(contents of text2)
.
.
.
R
```

You create either an indirect or macro file by creating a file (with EDIT or SPEED, for example) containing the desired CLI commands or text. When you invoke an indirect file or macro, whether it is a command or text file, the CLI performs the commands immediately and in sequence, as if each command were typed in at the terminal; the CLI does not echo the commands as it performs them, but does echo the results. Indirect files and macros are very handy for performing frequently-repeated chores, or for executing a lengthy or complex series of commands.

As CLI commands, the contents of indirect or macro files can consist of any legal CLI command format or punctuation, including parentheses, angle brackets, and templates. Indirect and macro files can reference other indirect or macro files. They can also contain variables, explained later in this chapter in the section Command Line Variables.

Invoking Indirect and Macro Files

Since the difference between the two types of files lies in whether the file is an argument (indirect text file) or command (macro or indirect command file), the CLI recognizes different methods of invocation for each type of file.

An indirect file is invoked by enclosing the filename of the indirect file in commercial at signs (@), in this way:

```
R

@cmndfile@
```

or

```
R

LIST @savefiles@
```

A macro file must have the extension .MC; the macro file is then invoked by simply typing the filename, in this way:

```
R

cmndfile
```

Note that you needn't type the .MC extension when invoking the macro. However, you can invoke a macro command file as an indirect file, in which case you must include the .MC extension; for example

```
R

@cmndfile.MC@
```

You can use directory specifiers and links to invoke macro and indirect files; in the case of links, intervening directories must be initialized for the link to work.

Examples of Indirect and Macro Files

Assume that you want to perform these housekeeping chores at the end of each CLI session in current directory *subdira*:

- Delete .BU files,
- List nonpermanent files,
- Determine free (usable) disk space,
- Release the current directory.

You can create either an indirect file called endday or a macro file called endday.MC containing these commands:

```
DELETE/V -.BU

LIST/E

DISK

RELEASE subdira
```

The above example shows that either the indirect or macro method is suitable for many command sequences. With either method, the CLI would perform each command in turn.

Both indirect files and macros can invoke other indirect or macro files. For instance, text1 and text2 might contain daily transaction records that you want printed at the end of each session. You can accomplish this by using BUILD to create file called alltext containing the filenames text1 and text2, and adding a line to endday.MC, thus:

```
DELETE/V -.BU

LIST/E/A

DISK PRINT @alltext@

RELEASE subdira
```

Command Interpretation Order

Because macro files are invoked as a command, confusion can arise if a macro file has the same name as a CLI command or utility, or the same name as a user program (.SV file) in the current directory. For this reason, the CLI follows these rules of precedence when interpreting a command.

1. When a filename is enclosed in @s, the CLI searches for a matching string in the current directory and executes any matching filename as an indirect command file.

2. When @s are absent, the CLI searches its own command and utility repertoire and executes any matching string.

3. When a CLI command does not match the string, the CLI searches for string.MC in the current directory and executes any matching macro.

4. When string.MC is not found, the CLI searches for *string*.SV in the current directory, and executes any matching program.

If you have both a macro (.MC) and a program (.SV) with the same name in the current directory, you can execute the program only by typing the .SV extension along with the filename.

These rules of command interpretation should be studied in conjunction with the Command Parsing Order section later in this chapter.

Batch Processing

Batch processing, like indirect and macro file processing, allows you to simulate a CLI session with a predetermined set of commands, to be invoked at your convenience. You submit the command sequence in a *job stream* (specifying one or more jobs) to the Batch Monitor with the CLI command BATCH. The Batch Monitor is a utility of the CLI; it has its own set of commands, most of which are very similar to CLI commands.

Batch Commands

Most Batch commands are adaptations of CLI commands and have similar formats. Each Batch command is preceded by an exclamation point (!), for example, !LIST. Batch commands are listed in Appendix D; those marked with an asterisk have no corresponding CLI command.

The !JOB command, which associates a label with a job, must be the first command in any stream and in any job in the stream. Some of the commands, for example !DKP (assign and name a disk) and !MTA (assign and name a magnetic tape drive), allow you to reference a symbolic device, deferring actual device assignment until job execution time.

Batch Job Input

Input to the Batch Monitor can be from a disk file or from one of these devices: card reader ($CDR), cassette unit (CT*n*), magnetic tape unit (MT*n*), paper tape reader ($PTR), or teletype reader ($TTR).

Figure 5.1 illustrates a batch job stream, input from cards, with two jobs.

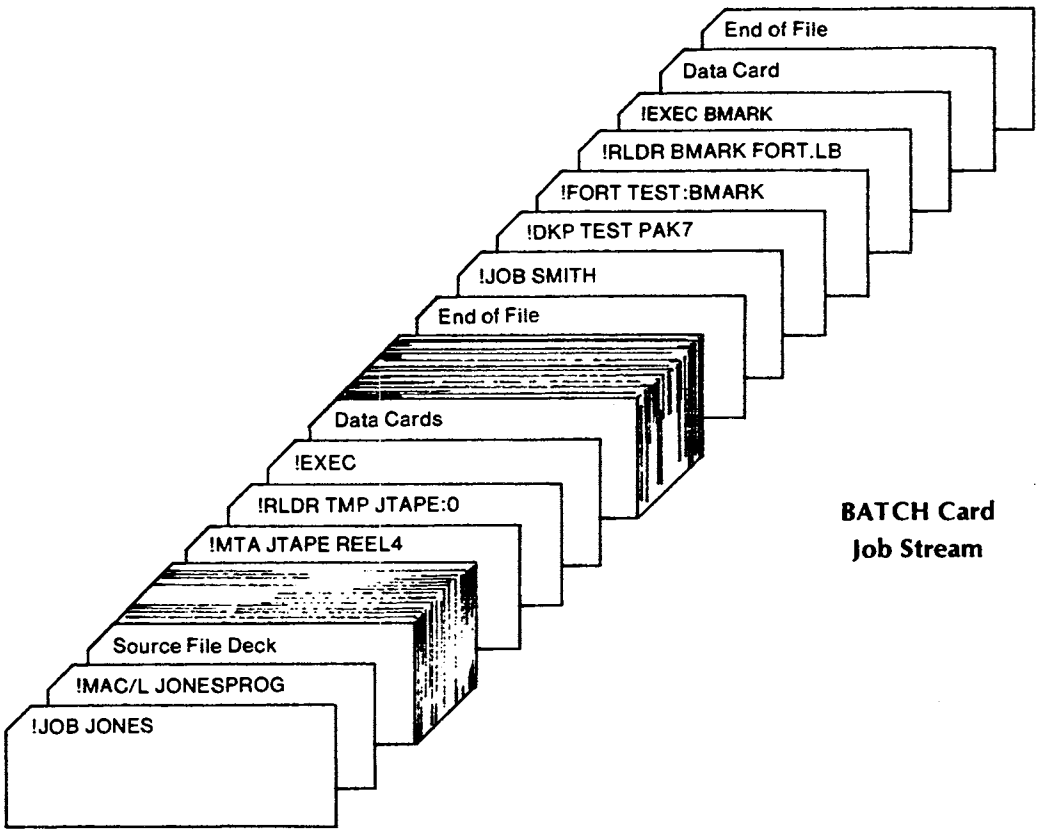


Figure 5.1 Batch card job stream SD-00665

When the job input is a disk file, the extension .JB is recommended, since the CLI will search for it. For the CLI BATCH command, the disk file's directory must be initialized, and the disk name (for instance, DK1) must be included in the directory specifier. Thus, to initiate a Batch job stream called MONTHEND.JB, the command could be

```
R

BATCH DP1:MONTHEND
```

To initiate the same job from the first file (file 0) on magnetic tape, the command is

```
R

BATCH MONTHEND.JB MT0:0
```

Batch Job Output

When you submit a job stream with the CLI command BATCH, the Batch Monitor takes over and executes the job commands. Transactions, results, and processing information about the job stream are recorded by the Monitor in two separate files:

- *SYSOUT* receives listing (program output) and error information. The name SYSOUT is reserved by RDOS for this purpose. The Batch Monitor, by default, writes SYSOUT to the line printer ($LPT), but you can change the SYSOUT destination to any output device or to a disk file with the BATCH command.
- The *log* file for the job stream receives the name of each job, as well as the date and time each job was begun and the date and time the job stream was completed. The default output device for the log file is the operator terminal ($TTO for a background program, $TTO1 for a foreground program). You can change the log destination to any output device or to a disk file with the BATCH command.

Command Line Variables

Variables are special words used in a CLI command to direct the CLI to return current information in place of the word. The CLI recognizes a pre-defined set of variables, listed in Table 5.1, consisting of commonly used RDOS values.

Variable	Value
%DATE%	Today's date, in the form mm-dd-yy
%GCIN%	The input terminal name, e.g. $TTI
%GCOUT%	The output terminal name, e.g. $TTO
%GDIR%	The current directory name
%LDIR%	The name of the previous current directory
%MDIR%	The master directory name
%FGND%	The character "F" if the CLI is executing in the foreground; nothing if the CLI is executing in the background
%TIME%	The time of day, in the form hh:mm:ss

Table 5.1 CLI Variables

Variables, which are enclosed in percent signs (%), can be used in any CLI command in which the returned value is logical and syntactically correct. For example, the command LIST 12:14:82/B directs the CLI to list all files created before 12 December 82 (today). The same effect is achieved with

```
R

LIST %date%/B
```

Variables are particularly useful in indirect and macro files and in Batch jobs. Because these mechanisms are usually applied to repetitive chores, the ability to include changeable information makes for more flexibility. For example, the macro file illustrated earlier in this chapter, Endday.MC, consisting of end-of-day chores, could be rewritten with a variable:

```
DELETE/V -.BU

LIST/E/A

DISK

PRINT @alltext@

RELEASE %GDIR%
```

With the variable %GDIR% included, this macro could be used in any directory.

The MESSAGE Command

The CLI MESSAGE command allows you to enter text for the CLI to display at the terminal. This command is particularly useful in indirect or macro command files, in which you might want to display a greeting, a reminder or an instruction to anyone who executes the indirect or macro file.

The format of the MESSAGE command is

```
MESSAGE ["]textstring["]...[textstring] ["]
```

If the text string is entered in quotation marks (" "), the CLI returns all characters literally, except for the quotes, which are text delimiters. Also, semicolons (;), ⟨CR⟩ and formfeeds (CTRL-L) are command delimiters and are not returned literally. A quoted text string can be up to 72 characters long.

If quotes are omitted, the CLI interprets special characters, such as percent sign (%) for variables and commercial at signs (@) for indirect files. Angle brackets (⟨⟩), parentheses (( )), commas (,) and slashes (/) are interpreted as they would be in a CLI command line. (To have these characters interpreted literally, they must be in a quoted text string.) Unquoted text strings are delimited as a CLI command, that is, with ⟨CR⟩ or semicolon (;). An unquoted text string can be up to 132 characters long.

When the MESSAGE /P switch is used in an indirect or macro file, the CLI *pauses* during execution and displays the *STRIKE ANY KEY TO CONTINUE* message. In this way you can ensure acknowledgement from the user before the command file proceeds.

The MESSAGE command can be illustrated by enlarging upon the enday.MC macro example shown earlier in this chapter:

```
MESSAGE ENDING SESSION AT %TIME%

DELETE/V -.BU

LIST/E/A

DISK

PRINT @alltext@

MESSAGE/P WAIT UNTIL PRINTING STOPS THEN

RELEASE %GDIR%
```

When this macro is invoked, the CLI displays the first message, including the time of day. The second message appears this way:

WAIT UNTIL PRINTING STOPS THEN

STRIKE ANY KEY TO CONTINUE

The CLI pauses until the user presses any terminal key, indicating that the CLI should proceed with the macro execution.

Command Parsing Order

The CLI evaluates the parts of a command string in the following order:

1. Text strings in quotes (MESSAGE command)
2. Indirect files and variables (as they appear sequentially)
3. Angle brackets
4. Parentheses
5. Numeric switches
6. Text strings outside quotes (MESSAGE command)

Understanding of these rules is especially important when using the MESSAGE command, where characters can be interpreted literally or with their special RDOS meanings. These parsing rules should be studied in conjunction with the Command Interpretation Order rules discussed earlier in this chapter.

The Assembly Language-CLI Interface

Though primarily an interface between the user, the operating system and the computer, the CLI can be used from within an assembly language program to execute commands and to pass arguments to the program. Certain delicate operations, such as file management, can be tedious to program in assembly language, while the CLI is built to handle such operations smoothly.

For the new user, the concepts in the following discussion can be better understood in conjunction with a reading of Chapter 7, on memory management, and Chapter 9, on assembly language programming. The subjects of this section, CLI.CM, COM.CM and .ERTN, are covered in *RDOS/DOS Command Line Interpreter*, (DGC No. 069-400015).

Use of the CLI from an assembly language program requires understanding of the CLI's command interpretation mechanism. The CLI uses two files, CLI.CM and COM.CM, to communicate with utility and user programs. The system call .ERTN, return from a swap with error status, can be used in an assembly language program in order to use the CLI to interpret error codes.

> **NOTE:** *If the CLI is running in the foreground of a mapped machine, these and other special CLI files are prefixed with the letter F, for example FCLI.CM and FCOM.CM. These are otherwise identical to CLI.CM and COM.CM. In the following discussion, the F prefix should be understood for foreground processing.*

CLI.CM

The CLI uses CLI.CM to deposit certain unedited commands for its own use. For example, the utility SYSGEN, which generates RDOS operating systems, stores a RDLR command line in CLI.CM (RLDR builds executable programs).

A user program can use system calls to store CLI command lines in CLI.CM, then *swap* or *chain* to the CLI to execute the stored commands. When a program is swapped out, RDOS saves the memory image of that program and brings the specified program into memory and executes it. This is also called a *push*. When a program chains, RDOS overwrites the executing program with the specified program.

The user program swaps or chains to the CLI using the system call .EXEC with a non-zero in Accumulator 2 (AC2). Thereupon, the CLI executes the commands in CLI.CM just as if they had been input from the terminal. To return to the user program, a swapped program can have a final CLI POP command (which returns you to the next higher level program) in CLI.CM, or you can issue a POP from the CLI. To restart a chained program from the CLI, you execute the program directly, either from CLI.CM or from the CLI itself. You cannot use a POP to return to a chained program.

COM.CM

The CLI uses COM.CM to hold edited command lines and switches from utility or user programs. When the CLI reads a command line and finds that it is not a CLI command, it assumes that the command involves a utility or user program. The CLI then builds COM.CM to contain the command line, including filenames, switches and arguments in an edited format. The CLI then invokes the appropriate utility, which always examines COM.CM, or the user program, which can examine COM.CM. You can use COM.CM to pass instructions and arguments to your program.

The COM.CM file built for any utility has a specific format that the utility expects. The file built for a user program also has a specific format that the user program must anticipate.

In a user command COM.CM, each character of the user command or argument occupies a byte, terminated by a null byte.. Four bytes each are reserved for command global switches and for argument local switches. Each letter switch sets a bit. The CLI uses 377_8 for the terminal byte.

The CLI does not interpret switches when it builds COM.CM, so you can write programs that interpret switches and act on what they read. To read arguments in COM.CM, use the system call .RDL, which reads a line from a disk file and terminates on a null byte. To read switches in COM.CM, use the system call .RDS, which reads the specified number of bytes whether they are null or not.

COM.CM can be examined from the CLI with the command FPRINT/Z/B COM.CM, which prints the file in byte format starting at location zero.

.ERTN

You can include the system call .ERTN, return from a swap with error status, in assembly language programs to instruct the CLI to interpret errors. When the program encounters an error that prevents it from proceeding, the CLI interprets the error code that is always returned in AC2 and displays the error message on the terminal.

The CLI attempts to describe the argument that caused the error. However, the argument named in the error message will always be the name of the program that took the error return, not necessarily the item that caused the error. This is so because the CLI is not active while the program is executing and encounters an error.

For example, assume that program UPROG.SV needs to read file Data.01. If UPROG cannot find Data.01, and takes the error return through .ERTN, the CLI displays the message

FILE DOES NOT EXIST:UPROG.SV

In this message, *FILE DOES NOT EXIST*: is the interpretation of the error code in AC2, *UPROG.SV* is the name of the program that took the error return.

It was the file Data.01 that was not found, not UPROG.SV, but UPROG is named as the cause of the error. Because this can sometimes be confusing, caution should be used when using .ERTN to track down your program errors.

Alternatively, a user program can take .ERTN with mnemonic EREXQ (code 17_8) in AC2. When an error is encountered and the CLI examines AC2, EREXQ instructs it to examine CLI.CM; the CLI then executes the contents of CLI.CM as described above. In this way you can specify action to take in an error condition.

Chapter 6

Input/Output

Input/output, or I/O, refers to the movement of data between the processor and any peripheral device: disk, terminal, magnetic tape, line printer, or any other device supported by the operating system. RDOS manages most I/O invisibly (for example, terminal or memory I/O) but allows the user to initiate I/O through CLI commands or through system calls in a program. RDOS automatically *spools* output to slow devices, temporarily storing data on disk so as not to slow the system as the device works.

Many CLI commands relating to disk are discussed in Chapter 3. This chapter discusses RDOS peripherals, CLI I/O commands for disk and other peripherals, and system calls for I/O.

Table 6.1 contains a list of devices supported by RDOS.

Device Name	Device
$CDR	Punched card reader; mark sense card reader.
CT*n*	Data General cassette unit *n*, first controller (*n* is in the range 0-7).
DP0	Data General model 6001-6008 fixed-head disk, first controller.
DP*n*	Data General moving-head disk pack, first controller, unit *n* is 0, 1, 2, or 3.
DP*n*F	Top loader (dual-platter Disk Subsystem. For the first controller, unit *n* is number 0, 1, 2, or 3. This unit has two disks. The top (removable) disk is DP*n*, the fixed disk is D*P*nF. This controller also supports diskette drives.
DS*n*	Data General Model 6063/6064 fixed head disk. The 6063 is single-density, the 6064 double density. *n* is 0, 1, 2, or 3.
DZ*n*	6060 series disk unit, first controller. *n* is 0,1, 2, or 3. The 6060 uses single-density disks, 6061 uses double-density disks.
$DPI	Input dual processor link.
$DPO	Output dual processor link.
$LPT	80- or 132-column line printer.
MCAR	Multiprocessor communications adapter receiver.
MCAT	Multiprocessor communications adapter transmitter.
MT*n*	First controller, 7- or 9-track magnetic tape transport. *n* is in range 0-7).
$PLT	Incremental plotter.
$PTP	High-speed paper tape punch.
$PTR	High-speed paper tape reader.
QTY	Asynchronous Line Multiplexor (ALM), asynchronous data communications multiplexor (QTY), or Universal Line Multiplexor (ULM).
$TTI	Teletypewriter or display terminal keyboard*.
$TTO	Teletypewriter printer or CRT display.
$TTP	Teletypewriter punch.
$TTR	Teletypewriter reader.

Table 6.1 RDOS device names

Table 6.1 RDOS device names (continued)

*For most devices, RDOS supplies an end of file mark. On $TTI and QTY input, however, you must indicate an end-of-file with CTRL-Z.

RDOS Peripherals

When you create an operating system with SYSGEN, you declare the RDOS-supported devices you have on your system, including disk drives and tape units. The disk on which you generate your first RDOS system holds the finished system and becomes the device containing the master directory. This can be either a fixed or moving-head disk. Later, if you choose, you can generate different systems on the original disk or other disks.

RDOS supports many different disk types, including diskettes; it can manage two controllers for each type. One type of controller can run both moving-head disks and diskettes.

One to sixeen magnetic or cassette tape drives can be included on an RDOS system. Magnetic tape units run at a density of 800 or 1600 bits per inch (bpi); both 7 and 9 track drives are allowed. You access individual tape files by means of either the CLI or a program by specifying the transport drive and file number.

During SYSGEN, you can configure RDOS to support up to two data channel or non-data channel line printers (80 or 132 column), two paper tape readers/punches, two card readers, two plotters, and a second terminal. Since RDOS routines (drivers) for these devices are re-entrant, allowing several tasks to use the routines without interfering with each other, you can add more devices later by creating device control tables (DCTs), which point to existing driver routines. See Chapter 8 for a discussion of user device drivers.

RDOS can also support two processors by way of an Interprocessor Bus (IPB), or multiple processors with a Multiprocessor Communications Adapter (MCA). These facilities are covered in Chapter 8.

For communications, RDOS supports two types of multiplexors: the Asynchronous Line Multiplexor (ALM) and Asynchronous Communications Multiplexor (QTY). Each is unit-expandable and can accommodate from 1 to 64 full-duplex lines at speeds up to 9,600 baud.

Each multiplexed line is recognized by RDOS as a file name in the form QTY:*x*, where *x* is the multiplexor line number (0 to 64). I/O operations on each line are either in line or sequential mode. (For an explanation of these modes, see the section ahead in this chapter entitled File I/O Reading and Writing Modes.)

Spooling

Because all devices are slower than the processor and some devices are slower than others, RDOS manages I/O in a way to maximize CPU usage. When data is output to a slow device like a terminal screen, the device's buffer, or temporary storage area, can become full, thereby suspending the task or command that initiated the transfer. RDOS helps to avoid this by storing data output to certain devices in a temporary disk file, then restarting the data flow when the device is free. This method, called spooling, frees the CPU for further processing and allows RDOS to queue output without storing it in memory.

Spooling frees the user from the need to organize output instructions because RDOS automatically spools data output to such devices as terminals and lineprinters. The devices for which RDOS spools are:

$DPO
$LPT
$LPT1
$PLT
$PLT1
$PTP
$PTP1
$TTO
$TTO1
$TTP
$TTP1

CLI commands and system calls allow you to control spooling to these devices, for instance, to disable and re-enable spooling. Table 6.2 shows the CLI commands and Table 6.3 shows the system calls.

Command	Function
SPEBL devicename ...	Enable spooling to devicename.
SPDIS devicename ...	Disable spooling to devicename.
SPKILL devicename ...	Stop spooling, delete spool files queued to devicename.

Table 6.2 CLI spooling commands

Command	Input	Function
.SPEA	Byte pointer to device	Enable spooling to device.
.SPDA	Byte pointer to device	Disable spooling to device.
.SPKL	Byte pointer to device	Stop spooling, delete spool files queued to device.

Table 6.3 Spooling system calls

Spooling requires that the operating system contain at least two system stacks in a single program environment and at least three system stacks in a foreground-background environment. You generate these stacks when you create an operating system with SYSGEN.

Spooling also requires disk buffering. RDOS allocates space for this dynamically from the master directory. If RDOS needs more disk space for spooling buffers and none is available, RDOS temporarily disables spooling.

I/O Commands of the CLI

The CLI methods to control I/O include commands that ready and release devices, or transfer data in different formats and between different devices, or activate a specific device. Data is always transferred in the form of a file. Peripheral devices are considered files and are addressed as such in CLI command lines.

Some commands pertain to several devices, some commands pertain to only one device. Table 6.4 lists CLI I/O commands, in the order in which they are discussed, and their pertinent devices.

Command	Devices	Function
INIT	Disk directory, MT*n*	Introduce device to system, enable for I/O.
RELEASE	Disk directory, MT*n*	Disable device from I/O, remove from system recognition.
XFER	Any RDOS device	Copy the contents of a file to another file.
DUMP	Disk file (source device) to disk file, MT*n*, CT*n*, $PTP	Copy one or more disk files in DUMP format, preserving SYS.DR information.
LOAD	Disk file, MT*n*, CT*n*, $PTR (source devices) to disk file	Load DUMPed files into files in current directory.
FDUMP	MT*n*	Fast-dump all files in current directory to MT*n*.
FLOAD	MT*n*	Fast-load all MT*n* FDUMPed files to current directory.
FPRINT	$TTO, $LPT	Print a disk file on the terminal or line printer in the specified format.
PRINT	$LPT	Print an ASCII file on the line printer.
VFU	$LPT	Create or load a format control file for a data channel line printer.
PUNCH	$PTP	Punch an ASCII file.
BPUNCH	$PTP	Punch a binary file.

Table 6.4 CLI I/O commands

Tape File Organization

Before using many of the CLI I/O commands, RDOS tape file organization must be understood. Tape files, whether magnetic or cassette tapes, are generally, but not necessarily, collections of files; that is, one tape file can comprise a group of files, for example all the files in a disk directory. The XFER command can copy single files to magnetic tape or cassette and thus the single file (whether disk file or paper tape) would constitute a discrete tape file. The DUMP command for magnetic or cassette tape transfers either single files or groups of files to the destination. The FDUMP command for magnetic tape transfers only groups of files (directories). For each DUMP or FDUMP command transferring a group of files, a single file is created on tape to contain the group of files.

Files are placed on tape in numeric order, beginning with file 0 and continuing up to file 99 (100 files can be placed on a tape, given small files and a long tape). RDOS writes two end-of-file (EOF) marks at the beginning of a tape when it is fully initialized with INIT/F. In later writing, RDOS starts writing at the first double EOF it finds, overwrites the second EOF in the pair, writes the file, and signifies the end by writing another double EOF mark.

You access tape files by using a command followed by a specifier which consists of the unit name, a colon, and the tape file number; for example,

```
R

PRINT MT0:5
```

This prints the sixth file of the tape mounted on magnetic tape transport 0.

Files must be written to tape in numeric order; you cannot write tape file 3 without having written tape files 0, 1 and 2. As you write files to tape, you should note their numbers and contents. Otherwise, you could inadvertently overwrite a file, thus losing that file and subsequent files on the tape. In addition, in later LOAD and FLOAD operations, you need to be confident of the files you are transferring to disk or other media.

INIT and RELEASE

The INIT command serves to introduce a disk directory or a magnetic tape transport to the operating system, enabling it for I/O. Without initialization, neither device can be accessed. The RELEASE command removes the specified device from the operating system, disabling I/O. These commands are discussed in Chapter 3 in relation to disks and directories and below in relation to magnetic tape drives.

Initializing and Releasing Magnetic Tape Transports

Initialization of magnetic tape transports consists of either *full* or *partial* initialization, similar to INIT/F for disks or INIT for directories. Full initialization, with the /F switch, prepares a magnetic tape for use under RDOS and effectively erases old data. Partial initialization opens the tape for use, without deleting files.

To first access a new or used magnetic tape on your system, you use the INIT/F command, for example,

```
R

INIT/F MT0
```

This command writes two EOF marks at the beginning of the tape, readying it to receive files. If the tape is not positioned at its beginning when this command is issued, INIT/F also rewinds the tape to the beginning. The /F switch effectively erases all files on the tape, and thus should be used with caution. The CLI issues a *CONFIRM ?* query when you use INIT/F. If you type a Y, the CLI echoes a *YES* and proceeds with full initialization. If you type any other character, the CLI issues a *NO* and cancels the INIT command.

To introduce to the system an existing RDOS tape whose contents you want preserved, you issue a simple INIT MT*n* command.

To logically remove a magnetic tape transport from the system, you use the RELEASE command, for example,

```
R

RELEASE MT1
```

This command rewinds the tape on the drive and closes the drive to I/O.

XFER

The XFER command copies a source file on any device to a destination file on any device. When a file is XFERed to disk, XFER creates the destination file name and copies the contents of the source file into it; the old file's attributes and creation dates are not copied. Directory specifiers pointing to files can be included in the XFER command line, but directories themselves cannot be XFERed. When a source or destination is a device, such as $TTI for input or $PTP for output, the device is named in the command line.

XFER can create a file from characters input at the terminal and terminated with a CTRL-Z. For example,

```
R

XFER/A $TTI newfile

IT'S ALL DONE WITH MIRRORS.

↑Z

R
```

This sequence creates a disk file named newfile containing the sentence entered at the terminal keyboard. The /A switch indicates an ASCII transfer. The CTRL-Z is not part of the file, but terminates input to XFER.

Switches to XFER allow you to specify an ASCII transfer, or to append files. For disk XFERs, you can specify random or contiguous organization for the destination file. The XFER default disk file block organization is sequential mode.

DUMP and LOAD

The DUMP and LOAD commands allow you to copy and retrieve disk files between various devices, preserving the file's name, access and link attributes, creation date, characteristics, and other information in SYS.DR. Disk files can be dumped to disk, or to magnetic, cassette, or paper tape. Dumped files can be loaded to disk from disk, or from magnetic, cassette or paper tape.

Files, partitions or subdirectories can be dumped and loaded. DUMP copies all nonpermanent files in the current directory, so an entire disk can be copied by making the primary partition the current directory. Templates can be used in file specifications.

Switches used with the DUMP command can control the copying of permanent files, link files, and files created before or after a certain date; with switches you can also rename files, exclude certain files, and list or verify the files dumped.

The LOAD command loads all files in the specified DUMP filename into the current directory. Switches used with LOAD allow you to select certain files for loading or simply to display all the dumped filenames without loading them.

The LOAD command can load only files that were previously dumped; it cannot load files that were transferred (XFER) or fast-dumped (FDUMP).

FDUMP and FLOAD

The CLI commands FDUMP and FLOAD invoke the corresponding utilities, which fast-dump and fast-load disk directories between disk and magnetic tape. These two utilities are designed specifically for magnetic tape; because RDOS writes larger records with FDUMP and FLOAD, these utilities are faster and use less tape than DUMP and LOAD. FDUMP and FLOAD also preserve SYS.DR information such as filename, access and link attributes, creation date, and characteristics.

FDUMP copies all permanent and nonpermanent files in the current directory; specific files cannot be named. FDUMP automatically initializes the drive specified and releases it after command execution. You can specify a second drive to continue to receive the dump when space on the first tape is exhausted. If you do not specify another tape drive, FDUMP prompts you to mount another tape when the tape on which it is writing becomes filled. Fast-dumped files can only be loaded with FLOAD.

FLOAD loads all files in the tape file number specified into the current directory; individual files cannot be specified. If a file on the tape has the same name as a file in the current directory, FLOAD prints the message *FILE ALREADY EXISTS*, does not load the file, and continues the load. FLOAD automatically initializes the specified drive but does not release it when through. If you have fast-dumped a tape file onto multiple reels on the same drive, FLOAD requests the next tape when it has loaded all data from the current tape.

With switches to FDUMP and FLOAD, you can list affected files on the terminal or line printer, or list fast-dumped files without fast-loading them.

FPRINT

FPRINT can be used to output a disk file to the terminal or the lineprinter in byte, decimal, hexadecimal, or octal format. Printable ASCII characters are output on the right, and any nonprinting character is output as a period (.). Used without switches, FPRINT produces octal format on the terminal. You can specify a first and last location with local switches, or send output to a file.

PRINT

The PRINT command copies ASCII files to the line printer. Multiple files can be given in a command line, and local numeric switches can specify multiple copies of a file.

VFU

The VFU command invokes the Vertical Formal Unit utility, which allows you to create and edit format control files for data channel line printers, and to load the format files into the printer's memory. VFU is useful for printing on non-standard forms. Three settings for printer forms can be specified:

- tab stops,
- form size in lines,
- multiple line-number/channel-number pairs.

The line-number/channel-number pairs allow you to control the starting and ending lines on a non-standard form.

VFU switches permit you to create, edit, and display format control files; the utility appends the extension .VF to the filename you specify. You use VFU to load a .VF file into your printer's memory.

The VFU command and utility is documented in *RDOS/DOS Command Line Interpreter* (DGC No. 069-400015) and *RDOS/DOS Sort/Merge and Vertical Format Unit Utilities* (DGC No. 069-400021).

PUNCH and BPUNCH

PUNCH and BPUNCH direct files to be output to the paper tape punch. PUNCH produces ASCII output and BPUNCH produces binary output. Input can be from any device.

I/O through System Calls

RDOS system calls allow a user program to communicate directly with the operating system. Issued from an assembly language program, system calls instruct RDOS to perform a function privileged to the operating system, such as I/O. Some arguments to the instructions are placed in accumulators as input to the call; for instance, call input could be a pointer to a disk file or memory location containing the real argument. The programmer anticipates that output from the call will be returned in particular accumulators; both an error return and a normal return must be anticipated.

Many of the CLI file, disk and I/O commands discussed so far have corresponding system calls. Functions such as creating, renaming, and deleting files, or creating, initializing, and releasing directories, can be accomplished through system calls. Even the CLI variables, like %MDIR%, are duplicated in system calls. RDOS system calls are listed in Appendix E and are documented in *RDOS System Reference* (DGC No. 093-400027).

This section describes the system call format and the system calls for file I/O and console I/O.

System Call Format

Each system call in an assembly program occupies four lines of code, comprising a system directive, the call, an error return and a normal return. Each call begins with the mnemonic .SYSTM, enabling the system to respond to the command. The form of a call is

```
.SYSTM.
call name
error return
normal return
```

After the system has obeyed a call, it takes a normal return to the second instruction after the command word. If it detects an exceptional condition, it takes the error return to the first instruction following the command word. System calls always pass the error code in AC2.

File I/O

File I/O involves assigning a channel number to the target file when opening the file for I/O; the open calls provide different levels of exclusivity for file access. Other calls allow you to determine and set the file pointer position in an open file. Various read and write calls are suitable to particular data organization modes. When finished writing or reading, you close the file and release the channel.

Table 6.5 lists system calls for file I/O.

Call	Function
.GCHN	Get the number of a free channel.
.STAT	Get a file's current directory status.
.RSTAT	Get a resolution file's current directory status.
.CHSTS	Get the file directory information for a channel.
OPEN *n**	Open a file for I/O.
ROPEN *n*	Open a file for reading only.
EOPEN *n*	Open a file for exclusive writing.
APPEND *n*	Open a file for appending.
MTOPD *n*	Open a tape file for free format I/O.
.GPOS *n*	Get the position of the file pointer.
SPOS *n*	Set the position of the file pointer.
.RDL *n*	Read an ASCII line from a file.
.WRL *n*	Write an ASCII line to a file.
.RDS *n*	Read sequential bytes from a file.
.WRS *n*	Write sequential bytes to a file.
.RDR *n*	Read a 64-word record from a file.
.WRR *n*	Write a 64-word record to a file.
.RDB *n*	Read one or more disk blocks from a file, with no system buffer.
.WRB *n*	Write one or more disk blocks to a file, with no system buffer.
.ERDB *n*	Read one or more disk blocks of a file into extended memory.
.EWRB *n*	Write one or more blocks from extended memory to disk.
.MTDIO *n*	Write or read data to or from a tape in free format.
.CLOSE *n*	Close the file on channel *n*.
.RESET	Close all files.

Table 6.5 File I/O system calls

**n* represents channel number

I/O Channel Numbers

Before you can access a file for I/O, you must open it with one of the open calls, assigning it an I/O channel number. While the file is open, it retains this channel number, and you access it by means of the number instead of the filename. When you close the file, the number is released.

The maximum number of channels on an unmapped system is 256. For mapped systems, you specify during SYSGEN the maximum number of foreground and background channels, at a maximum of 256 each.

For a single-task program, RLDR allots eight I/O channels, numbered 0 through 7. Usually, this is enough. If you need more channels, use either the RLDR local /C switch, or the assembler pseudo-op .COMM TASK.

You can assign a channel number to a file in two ways: by issuing the number directly with the open call, or storing the channel number in an accumulator. You can ascertain the number of a free channel by issuing the call .GCHN.

System calls .STAT and .RSTAT allow you to get a copy of a file's current directory status information. These calls return the file's user file definition (UFD), which contains filename, extension, attributes, access information, and device name. Similarly, the .CHSTS system call returns UFD information on whatever file is currently open on the specified channel.

Opening Files for I/O

You can control access to a file by opening it for a specific type of access. The most common way of opening a file is by system call .OPEN, which provides the user with a non-exclusive capability to read and modify the contents of a file. An .OPEN does not prevent the file from being opened by other users, nor does it forbid any users from modifying the file's contents.

Another system open call, .ROPEN, also permits several users to open a file for reading simultaneously, on different channels, yet prevents any of the users from modifying the file's contents.

A third system open call, .EOPEN, is more restrictive than the others. A program that .EOPENs a file has read access and exclusive write access to the file; other users can .ROPEN the file for reading only. The .APPEND call is identical to .EOPEN except that it opens a file specifically for appending.

The call .MTOPD opens a magnetic tape or cassette unit and files for free format I/O. Free format I/O is explained ahead in the section File I/O Reading and Writing Modes. You can specify a file on the tape with .MTOPD or you can access all files on the tape.

Positioning the File Pointer

Once a file is open, you can use the call .GPOS to determine the next character position within a file where program writes or reads will occur.

With the call .SPOS you can reset the current system file pointer to a new character position. For magnetic or cassette tapes, you can specify only position 0, the file starting location.

.SPOS allows you random access to characters and lines within any block of a file. You can read a character after writing or rewriting it simply by backing up the pointer to its previous position.

File I/O Reading and Writing Modes

RDOS provides five basic modes for reading and writing files: line, sequential, random record, direct block, and free format (tape only).

Line Mode

In line mode, the data consists of ASCII strings up to 132 characters in length, terminated by a carriage return, form feed, or null character. Data reading or writing continues until RDOS detects one of these three characters. The system edits all device delimiters at the device driver level. For example, it ignores line feeds on input devices, and supplies line feeds after carriage returns on all output devices. Furthermore, neither reading nor writing requires byte counts, since reading continues until RDOS sees a terminator, and writing proceeds until RDOS writes a terminator.

The line mode calls are .RDL and .WRL.

Sequential Mode

Sequential mode provides unedited data transfers. In this mode, RDOS assumes nothing about the data; thus you always use sequential mode for processing binary data. You can also use it for processing ASCII data, provided you require no editing of this data. Sequential mode transfers require specific byte counts to satisfy read or write requests. You can use any I/O device for sequential data transfers.

The calls for sequential mode are .RDS and .WRS.

Random Record Mode

This mode gives you random read/write access to 64-word segments within random or contiguous disk files. Data transferred in this mode can be either ASCII or binary, though no device-level editing occurs.

The random record calls are .RDR and .WRR.

Direct Block Mode

In direct block mode, RDOS transfers binary data directly between a disk file and a specified memory area, without using the system buffer required for all other modes. Because they require no system buffer, direct block transfers are faster than any other type of transfer. RDOS uses sequential memory locations in the operation, and it transfers 256 word blocks of data between memory and disk. Extended direct block mode, similar to direct block mode, permits you to transfer binary data between a disk file and mapped extended memory.

You can execute direct block I/O with the calls .RDB and .WRB. Extended direct block I/O calls are .ERDB and .EWRB

Free Format Mode for Tapes

Free format I/O allows you to access magnetic tape and cassette units on a machine level. In this mode, you can read or write data in variable length records, from 2 to 4096 words within a record. You can also space forward or backward from 1 to 4096 data records or to another file, and execute similar machine level operations.

To perform free format I/O, you use .MTDIO.

Closing Files

After completing write or read operations, you must issue a .CLOSE in order to update or delete a file's UFD information, or to release its directory or device. Closing also releases the channel number associated with a file.

An alternative call, .RESET, closes all open files after writing out any partially-filled system buffers. In a multitask environment, you issue .RESET only when no other task is using a channel.

Console I/O

Console (terminal) I/O calls control the sending and receiving of data to and from the terminal. Console I/O does not involve files, merely single characters. The calls for console I/O reference $TTI/$TTO or $TTI1/TTO1; the terminal is always available to the calls, and no channel number or open call is needed. The console I/O system calls are listed in Table 6.6.

These calls work on foreground and background terminals only, not on terminals connected to a multiplexor.

Call	Function
.GCHAR	Get a character from the terminal.
.PCHAR	Write a character to the terminal.
.GCIN	Get the operator input terminal name.
.GCOUT	Get the operator output terminal name.

Table 6.6 Console I/O system calls

The .GCHAR call gets a character typed in at $TTI or $TTI1. RDOS does not echo the character at the terminal when you type it.

.PCHAR puts a character, that is, displays on $TTO or $TTO1 the character contained in an accumulator.

The calls .GCIN and .GCOUT allow you to determine the appropriate console at run time for a foreground/background system. $TTI is the name of the background input console, and $TTI1 for the foreground; likewise for the output devices $TTO and $TTO1. .GCIN returns the input terminal name, and .GCOUT the output terminal name.

Chapter 7

Memory Management and Multitasking

RDOS merges the speed of a disk operating system with the speed of a memory resident system, a combination that optimizes real-time control. The RDOS *executive*, the heart of the operating system, is resident in memory while it manages the execution of the CLI, system utilities, and user programs, calling RDOS modules from disk into memory when necessary. Understanding the methods of RDOS memory management, and the tools RDOS offers the user to control memory, is central to using system resources to greatest effect.

RDOS runs on machines configured with a minimum of 40 Kbytes of memory (NOVA) to a maximum of 2 megabytes of memory (ECLIPSE). Some RDOS systems are equipped with the Memory Allocation and Protection unit (MAP), a hardware option; mapped systems can support up to 256 Kbytes of memory (NOVA) or 2 megabytes of memory (ECLIPSE). The operating system you build with SYSGEN consumes some of this memory, the amount depending on the peripheral devices and software structures you specify in SYSGEN.

For user programs, you can balance available memory resources with disk storage through such means as swaps, chains, overlays, or foreground-background processing, extending the reach and speed of any user application. In the program development cycle, the Relocatable Binary Loader, or RLDR, helps you to direct the memory usage of your application.

Another tool that helps you to increase system efficiency is multitasking. In a multitasking program, one task can start while another task is in a suspended state, awaiting the completion of a system call or an I/O operation. Multitasking thus optimizes use of CPU time, peripherals, and memory resources for real-time processing.

This chapter covers the organization of memory for both RDOS and user programs. We discuss the program development cycle, including the use of RLDR and the system library. The chapter also introduces memory conservation techniques, including swaps, chains and overlays. Foreground-background processing for both mapped and unmapped machines is covered. The discussion of multitasking includes sections on task states and priorities, Task Control Blocks, task calls, and task timing control.

This chapter can be read in conjunction with Chapter 9, Assemblers and Program Utilities, and Chapter 10, RDOS High Level Languages. All the material in this chapter is documented in *RDOS System Reference*, (DGC No. 093-400026) and *RDOS/DOS Assembly Language and Program Utilities*, (DGC No. 069-400019).

Appendix E summarizes system calls, and Appendix F lists task calls.

Memory Organization Under RDOS

The RDOS executive is the main framework of the operating system, and it must reside in main memory for RDOS to process data. This resident portion handles interrupts, manages system buffers and system overlays, and executes system calls. The executive brings system overlays into main memory from disk storage to perform such specific functions as full or partial system and device initializations, and file maintenance operations like opening, closing, renaming or deleting files, and spooling control.

Figure 7.1 illustrates RDOS memory organization.

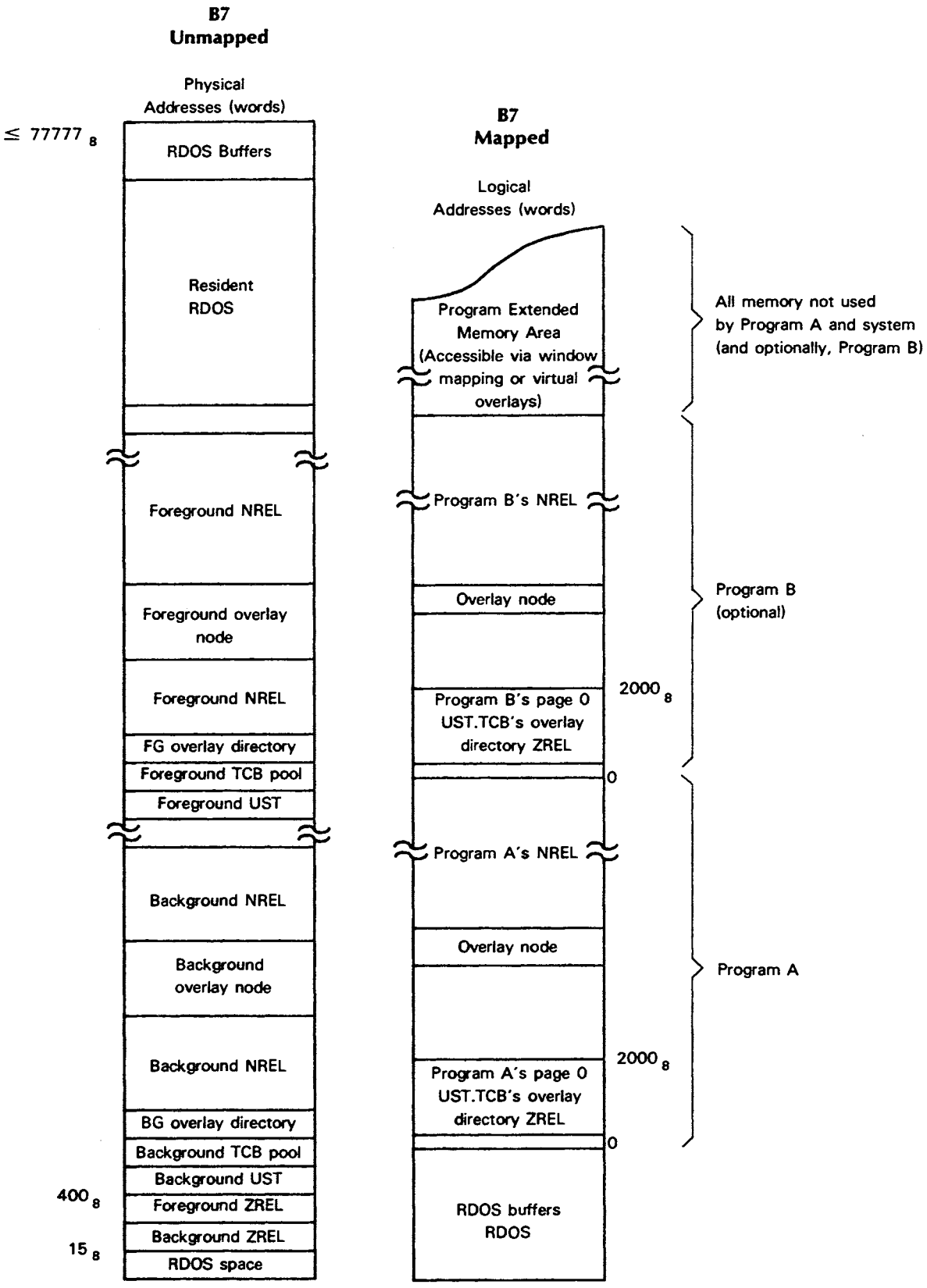


Figure 7.1 RDOS address space

ID-00334

In an unmapped system, the RDOS executive resides at the top and bottom of memory. Locations 0 through 15_8 contain program and interrupt entry points into the top area of RDOS. In a mapped system, resident RDOS begins at location 0 and extends to the highest address required; it is invisible to your programs.

Above resident RDOS in all systems (at the very top of memory in unmapped systems) is a series of system buffers, which handle buffered I/O transfers and hold system overlays and directories from disk.

The portion of *page zero relocatable memory* (ZREL) available for user programs begins at location 16_8, called the *User Stack Pointer*, or USP. The USP is a special location preserved by RDOS whenever program control is transferred from one task to another. RDOS uses location 17_8 as an indirect address to the call processor, to process system calls. After the USP, user ZREL skips to locations 20_8 through 37_8 then extends from 50_8 through address 377_8. In an unmapped system, these are physical addresses; in a mapped system, they are logical addresses.

Above program ZREL, *normal relocatable memory* (NREL) begins at address 400_8. As with ZREL, NREL locations are physical addresses in an unmapped system, and logical addresses in a mapped system. The Relocatable Loader, RLDR, builds a *User Status Table* (UST) for your program at location 400_8. The UST describes, among other things, your program's length, number of tasks required, and number of I/O channels needed.

Above the UST, RDOS reserves an area for a pool of *Task Control Blocks* (TCBs), which RDOS uses to store task state information. If you define *overlays*, routines called from disk as needed, in your RLDR command line, then an overlay directory is placed above the TCBs. RLDR reserves a *node*, or vacant space, in your program for each overlay segment you define. The remainder of your program resides in NREL space above any overlay directory.

Above your program, still in NREL memory, are the task modules required to support multitask calls issued by the program, and the appropriate *Task Scheduler*, which switches control from task to task during program execution. Both the task processing modules and Task Scheduler are obtained from the *system library*, SYS.LB, a collection of commonly-needed routines, and loaded into your program by RLDR.

Program Development

Depending on your programming language interests, you use one or more RDOS system utilities to develop your programs, invoking the utilities through commands to the CLI. Between any two steps, you can use other CLI commands to maintain, protect and examine the files produced. Both the Relocatable Loader and the system library, are particularly important in program development and are discussed in the following pages. The cycle for program development is outlined below.

- Assembly language

  Write or change one or more assembly source files, using SPEED or EDIT. System and task calls can be included in the program.

  Assemble the source file(s) (ASM or MAC command) to produce one or more binary files (.RB).

  Load the binary file(s) with RLDR, producing an executable program (.SV). To debug a new program, you can load it with a Symbolic Debugger.

- BASIC

  The Extended BASIC interpreter or Business BASIC interpreter allows you to produce a finished program without using the RDOS utilities directly.

- Compiled languages (FORTRAN IV, FORTRAN 5, DG/L)

  Create or edit one or more program source files, using SPEED or EDIT.

  Compile the source file(s) (.SR) with the appropriate compiler utility. Compilation produces one or more binary files (.RB).

  Load the binary file with RLDR, including any required modules from the language libraries. The result is an executable file (.SV).

The RLDR Utility

RLDR processes one or more binary files into an executable program, as shown above in the program development cycles. When you load a program, RLDR builds into it the tables, modules, directories and Task Scheduler (single or multitask) that you specify.

For system calls, RLDR loads the necessary code from RDOS modules. For task calls, the utility searches the system library, SYS.LB, for the task call processing modules. Note that an important difference between system and task calls is that system calls execute in system memory space, while task calls execute in user memory space. Thus task memory usage must be counted in the total memory needs of a program.

RLDR assumes a single-task program unless you specify a multitask program in the RLDR command line or issue a .COMM TASK statement in the program .RLDR loads either the single Task Scheduler or multitask Task Scheduler from SYS.LB.

You introduce each task call your program uses with an .EXTN statement at the start of the program. RLDR then inserts the processing modules into the program.

The loader also allots eight I/O channels by default for any program. More channels can be specified with a local switch to RLDR.

Program overlays are defined in the RLDR command. You can conserve memory space by coding certain program segments as overlays, which are read into memory one by one, as the program needs them.

These and other aspects of RLDR are touched on in the following sections. For further discussion of RLDR, see Chapter 9, Assemblers and Program Utilities.

The System Library

The system library, SYS.LB, is a collection of program modules that support user programs. You specify the modules you need to RLDR, which extracts the necessary code from SYS.LB. The modular organization of the system library thus helps to conserve memory.

Among the modules in SYS.LB are

- single Task Scheduler,
- multitask Task Scheduler,
- Symbolic Debuggers,
- BFPKG, the buffered I/O package,
- command processing modules for each task call type.

The Symbolic Debuggers help you to detect, locate, and remove program errors while executing the program, allowing you to set breakpoints in order to stop and resume action.

BFPKG can help speed up line and sequential I/O operations by providing extra buffering within multi-user program space.

Memory Conservation Techniques

A user program can stretch available memory space by means of several techniques provided by RDOS. They are *swapping*, *chaining*, and *overlays*, described below, and *extended memory*, covered under Foreground-background Processing on Mapped Systems later in this chapter.

Program Swaps and Chains

Any program running under RDOS can suspend its own execution and either invoke another distinct program (swapping) or call for a new section of itself (chaining).

RDOS offers five levels of program execution, levels 0 through 4, where the CLI normally executes on level 0. When a program swaps out, it calls another program into execution on the next level (1, 2, 3, or 4). When the swapped-in program terminates, the last swapped-out program is reactivated. The program whose memory image is overwritten during a program swap operation is stored temporarily on disk. Information saved by the swap facility enables the original program to continue upon return from the swap.

When a program chains, it calls a new program into execution on the same level. Chained programs run serially. To use chaining, a program must be written in serially executable segments, each of which calls the next segment. In a chain, RDOS does not save the core image, but brings the new program into core and executes it.

Both swapping and chaining allow you to run programs that require more memory than is ever available at one time.

Every new program (or section of the same program) to be swapped or chained must exist on disk as a save file (.SV), and contain core images extending from address 16_8, the User Stack Pointer (USP), through the highest address in the user program, or NMAX.

Swapped and chained programs can pass parameters and results to one another by means of disk files with agreed-upon names.

Overlays

In many cases, an application program is too large for user address space. Program swaps and chains can handle some of these situations; for others, RDOS offers user overlays. User overlays are routines of limited size that are called into memory from disk one by one during execution of the main program. The overlays execute in fixed-size memory nodes, and a program can have many nodes. Thus you can segment your program into parts that fit into the fixed-size memory nodes.

User overlays differ from program swaps in that overlays are read into a fixed node within a root program. The root program remains active and memory resident while the overlay executes. Moreover, user overlays are more flexible than program swaps, since a program can execute part of itself in one or more currently resident overlays while a nonresident overlay is being loaded by the system.

In addition, if more than one user task requests a specific overlay, that overlay will be locked into memory until all requests for it have been satisfied. RDOS itself uses overlays to improve its own operations; these are called *system overlays*.

The complete set of user overlays used by a root program is called a *user overlay file*. You specify overlays to RLDR, which stores them in the overlay file, calling it *programname*.OL. Overlay files are organized contiguously. Contiguous file organization and memory image format allow a program to load overlays into a root program quickly and efficiently.

Associated with each memory node is a segment of the overlay file that contains one or more user overlays. RLDR reserves enough space in each overlay node to accommodate the largest overlay that will be called into that node. A single root program can have up to 128 user overlay nodes. Each node can receive up to 256 separate user overlays serially from its segment of the overlay file.

Foreground-Background Processing

Many computer installations require users to develop and run their programs at different times. That is, program development must be completed before any real-time program can be run. Moreover, once a real-time program is in execution, no other program can run concurrently, although the real-time program itself requires only a fraction of available system resources at any given moment.

To help eliminate this inefficiency, RDOS offers foreground-background operation, also called dual-ground processing or dual-programming, in which two different programs reside in memory, and appear to execute simultaneously. That is, the operating system switches control between one of two Task Schedulers according to a predetermined priority assigned to each by the user. Foreground-background processing permits RDOS to run unrelated collections of tasks between the grounds so that they share basic system resources.

A typical foreground-background system under RDOS might contain a real-time process control program in foreground memory, as well as assembly, compilation, and payroll programs at different times in background memory.

When you bootstrap your system, RDOS starts up in the background; all programs you execute run in the background until you command RDOS to execute one in the foreground. When two grounds are run, they share such system resources as CPU time and I/O devices.

Foreground-background processing can run on both unmapped or mapped systems, though it is designed for, and runs most efficiently on, mapped systems. A mapped machine has a hardware option called a Memory Allocation and Protection (MAP) unit, which provides a hardware separation between the two grounds. A distinct version of RDOS, called mapped RDOS, is run on mapped machines.

Communication Between Foreground and Background

Although the foreground and background programs are truly independent, RDOS does allow them to act in concert. System calls permit the two programs to exchange messages in 256-word areas of user address space. On systems equipped with the Multiprocessor Communications Adapter (MCA), foreground and background can communicate at data channel speeds. And on mapped systems, the two programs can communicate with window mapping by means of shared memory pages.

CLI Interface to Foreground-Background Processing

On a mapped system, you set the size of background memory with the CLI command SMEM, thereby allotting the remaining memory to the foreground. You can then execute a program in the foreground with the command EXFG *programname*.

On an unmapped system, you also use the EXFG command to start a foreground program, but you must have configured the program with specific address information beforehand. The program itself, not the SMEM command, allots foreground memory.

When you run two grounds, the system terminal communicates with the background program and you use another terminal with the foreground program. Dual programs can communicate through system calls; each can initialize and use directories and devices, and each can open and read files. Both programs can access the same directory simultaneously.

On a mapped system, you can display the current foreground/background memory allotment with the command GMEM.

To terminate the foreground program, type CTRL-F from the background; CTRL-F is not recognized from the foreground terminal. CTRL-F releases all devices and directories initialized by the foreground and restores total control to the backgound.

You can also terminate a foreground program (except a text editor) with CTRL-A or CTRL-C from the foreground terminal. These work similarly to CTRL-F; CTRL-C additionally creates a breakfile called FBREAK.SV to save the foreground memory image.

Foreground-Background Processing on Unmapped Systems

For an unmapped system, foreground and background memory usage must be assigned manually, with specific boundaries. You specify boundary addresses for the foreground program with RLDR; this boundary then sets the starting page zero relocatable (ZREL) and starting normal relocatable (NREL) addresses for the program. In an unmapped system, RDOS and the two programs share a maximum of 32 *pages* (64 Kbytes) of memory. A page is 1,024 words (2,048 bytes).

Though a system that runs two programs instead of one can use its hardware resources far more efficiently, there are drawbacks to dual-programming under unmapped RDOS. First, if one program fails, as it would if it accessed a word outside its allotted address space, it may bring down the other program. Second, both foreground and background programs must be tailored very carefully to fit into available memory. Finally, each ground must share page zero, restricting the amount of this resource available to each.

Foreground-Background Processing on Mapped Systems

The MAP feature allows you to specify addresses as you do in an unmapped system, but the system remaps them into pages of 1,024 words. Addresses in mapped systems are called *logical addresses* instead of physical addresses. Figure 7.2 shows logical-to-physical address mapping.

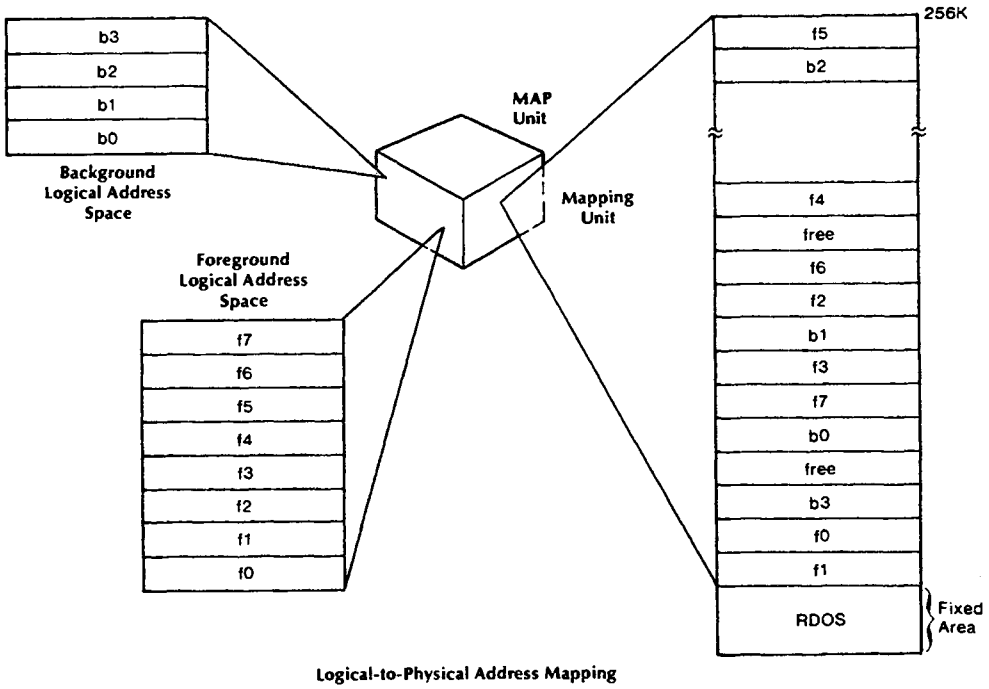


Figure 7.2 Logical-to-physical address mapping SD-00673

Foreground and background programs in a mapped system can each directly address up to 32 pages (64 Kbytes) of memory at the same time. Mapped RDOS helps protect itself by preventing unauthorized user access to system devices like the CPU, and offers several helpful CLI command and system calls for allotting and controlling memory in each ground. You can *checkpoint* a background program from the foreground, temporarily calling in another background program. You can also access *extended memory*, enlarging the total user program space.

In a mapped system, two addressing modes exist. In the first, absolute mode, only the lower 32 pages are directly addressable and the mapping device is not used. RDOS resides in these low physical memory locations and executes in absolute mode, while the user never deals with the system in absolute mode.

In the second, called mapped or user mode, the system maps up to 32 pages of memory to produce a *logical* 32 pages of continuous address space. Both the foreground and background programs execute in mapped mode and are essentially unaware of their actual memory locations.

Any program operating in user mode uses a complete logical address space including its private page zero and extending through its upper memory bound (NMAX). NMAX is determined by the requirements of the individual program and can extend as high as logical word address 32,767. Mapped RDOS assigns free memory blocks from its available pool to each program before executing it, manages the map, and builds the program in logical address space. Mapped RDOS also allocates the data channel map, which both system and user devices can access.

While a user program runs, it can communicate with mapped RDOS through all the standard RDOS system calls and through a few additional calls designed for a mapped system.

Checkpointing on a Mapped System

Mapped systems permit the foreground to send the current background to disk temporarily and to read in a different background program for execution. This procedure is called checkpointing.

One example of a checkpointing application runs a data collection program in the foreground and one of several program development utilities in the background. In this application, the foreground requires occasional use of a data reduction program to be executed in the background. The foreground then checkpoints the lower-priority background temporarily, runs the data reduction program, then restores the original background program.

Extended Memory on a Mapped System

Mapped addressing extends the total amount of resident memory, yet it does not itself permit any single user program to use more than 32 pages of memory. Since this restriction is unacceptable for some application programs, RDOS provides extended memory. Extended memory, also called extended address space, can raise the total user program space up to 2 megabytes of memory. The actual amount of memory depends on the size of your RDOS system.

Extended memory can be used under mapped RDOS in both a single program and dual program environment. You allot the total (both logical and extended) address space to the program with the CLI command SMEM.

The two means provided by mapped RDOS for accessing expanded address space are *virtual user overlays*, and *window mapping*; with window mapping, you can use *extended direct block I/O* and *shared memory pages*.

Both virtual user overlays and window mapping create extended address space by storing data in memory blocks outside the logical 32 page address space. When your program wants to access this data, it issues a system call to remap the desired blocks into the logical 32 page address space. It is in the precise methods of storing the data and enabling the memory management unit that these two methods differ.

A virtual overlay is an overlay that you identify as virtual in the RLDR command line. RDOS then loads it into extended address space when your program opens the overlay file. Your program can then access the overlay by means of the same system or task call that loads an ordinary user overlay from disk. Access to virtual overlays is very fast because they reside in memory; ordinary overlays require a disk access and actual transfer of data.

Window mapping allows your program to access the map directly. Data is mapped in 1-page (2-Kbyte) blocks to and from a fixed area in logical address space (called a window), and extended memory. Windows, like virtual overlays, are defined in integer multiples of 1-page blocks and are block-aligned. Window mapping is most useful for extended data storage; your program defines a window map.

Once a program defines a window map, it can use extended direct-block I/O. This is similar to direct-block I/O described in Chapter 6. The extended direct-block I/O calls can transfer 256-word data blocks between the map in extended memory and disk files, providing a quick means of altering data in the extended memory area.

Window mapping also facilitates the use of shared memory pages between foreground and background. During SYSGEN, you allot the number of pages for sharing, then specify shared memory pages when defining a window in your program.

Window mapping and virtual overlay loading are similar in that no true data transfer occurs. The *addresses* of the data are changed. This accounts for the high speed of extended memory access.

Memory Protection on a Mapped System

Mapped RDOS allows you to write-protect memory in 2 Kbyte blocks, a feature that can be very important in multitasking. Write-protected memory can be overwritten by an overlay, swap or chain, but by no other instruction; with this feature you can maintain the integrity of your overlay nodes as your program executes.

Multitasking

A task is a complete, self-contained execution path through a program, subprogram or overlay that demands system resources, such as the CPU or I/O devices. In a single task environment, a program has a single unified path connecting all its program logic, no matter how complex the logic branching. During execution, the processor idles when the program initiates an I/O operation; the program continues executing when the I/O operation is complete. Most FORTRAN programs, most user assembly level programs, and even system utility programs like an assembler or editor, are all examples of single task programs.

However, most computer applications must process a multitude of unrelated tasks. Under RDOS, one real-time program can have many logically distinct tasks, each with its own priority.

Each task performs a specified function asynchronously and in real time (that is, as close to instantly as possible). The RDOS Task Scheduler allocates CPU control to the highest priority task that is ready to perform or continue performing its function. Many tasks can run in the same re-entrant code path.

Typical multitask applications are airline reservation systems, inventory control systems, process control operations, and communications networks with message queueing and switching. Individual tasks within an inventory control system might continuously update inventory totals using data received from remote terminals, and reorder items whose quantities fall below specified reorder points.

Appendix F contains a list of RDOS task calls.

Task States and Priorities

Each multitask program that you run under RDOS has one task, the default task, initiated for it automatically. To create another task, your program, which is the default task, specifies a priority for the new task, then issues the appropriate task call (.TASK or .QTASK, described later).

Then, the RDOS Task Scheduler decides which task should be executing. The Scheduler chooses according to the priority of each task; this priority can later be changed. RDOS allows 256 levels of task priority (0 to 255), with priority 0 being the highest. Several ready tasks can exist at the same priority level, and the scheduler services these in round-robin fashion.

A task can be in one of three active states—executing, ready, or suspended—or in a dormant state:

- Executing — A task has CPU control and is executing its assigned instruction path
- Ready — A task is waiting for execution, but cannot reach execution state until it becomes the highest-priority ready task when the currently-executing task is reduced to some other state
- Suspended — A task is awaiting the occurrence or completion of some system or task call, or it is awaiting a specific real time event
- Dormant — A task not known to the system; the program has not initiated the task or has killed the task

The default task begins executing at the highest priority and since it is the only task in the system, it receives control. When this task invokes the second or subsequent tasks, the Task Scheduler puts the highest priority task into *execution*. Other tasks that have been initiated but are of lower priority are *ready* to run. The Task Scheduler always gives CPU control to the highest priority task that is ready. Many tasks can be ready, but only the highest priority task receives control.

The executing task becomes *suspended* when it issues a system call, or certain task calls. The task remains suspended until the call is completed, at which time it becomes readied.

Actually, there are two different types of task suspension, and a task can become doubly suspended, in which case it won't become ready until both reasons for its suspension are resolved. One type of suspension is caused by a task issuing either a system call or a transmit/receive message task call, or by requesting an occupied overlay node. Another type of suspension occurs when a task is suspended by one of the three suspend-task calls: .SUSP, .ASUSP, or .TIDS. No task can be raised to the ready state until both types of suspension are removed.

A task is said to be *dormant* when your program has not initiated the task, or has killed the task. A dormant task is unknown to the operating system, even though its code might be core resident.

Task Control Blocks (TCBs)

When a task is active (executing, ready or suspended), RDOS maintains certain status information about it so the Scheduler can keep the highest priority ready task executing. This information is kept in a *Task Control Block* (TCB). TCBs contain active registers and other priority and status information when a task exists in either the ready or suspended state.

RDOS maintains one TCB for each task in the active state (no TCB for a dormant task). RDOS reserves the TCB of the executing task for the task, but doesn't use the TCB until the task loses control of the CPU. When a task becomes ready or suspended, RDOS uses its TCB to store its status information. If, on the other hand, the task is killed or kills itself, it becomes dormant, and RDOS places its TCB in a pool of available TCBs.

RDOS links the TCBs of ready, executing, and suspended tasks in a chain, organized by decreasing task priority. Each TCB in the active chain is connected to the next TCB in the chain. Thus the Scheduler needs look only to select the highest-priority ready task. RDOS executes equal priority tasks in a round-robin fashion. The last TCB in the active chain has a link of -1.

Unused TCBs in the system are also linked together to form an inactive chain of available TCBs. Except for the link words, these TCBs are empty until a task is initiated; then RDOS removes a TCB from the free chain, fills it with information about the task, and places it on the active chain.

Task Calls

Unlike .SYSTM calls, task calls consist of single words that create or modify tasks. The four essential differences between task calls and system calls are:

- Task calls have no .SYSTM mechanism: each call uses a module from the system library, and therefore you must declare each call included in a program as external, in an .EXTN statement. Otherwise, RLDR won't load the call's module and the call won't work.
- RDOS executes all system calls in RDOS space, but executes all task calls in user space. Thus the use of task calls in a program affects its size, whereas the use of system calls does not.
- Most task calls do not have error returns, and hence do not reserve an error return location.
- You use accumulators to pass all arguments to most task calls.

Each task call changes a task's state. Most task calls cause the issuing task to move from the executing state to the ready state. However, other task state transitions are possible.

RDOS offers a complete set of task calls to monitor and control a real-time environment. These calls divide into two broad categories: calls your program issues, and task commands you issue through the console.

Program-level Task Calls

There are six categories of program-level task calls:

- Task initiation,
- Task execution control,
- Task control by identification number,
- Intertask communication,
- User clock and user interrupt control,
- Task Scheduler control.

Before a multitask environment can be created, dormant tasks within the user program must be *initiated*, or raised from the dormant to the ready state. Two task calls, .TASK and .QTSK, raise dormant tasks and initiate the multitask environment. The default task (program) issues at least the first of these calls; after more than one task is running, any task can issue them. The executing task that issues a .TASK call is reduced to a ready state.

Having initiated several tasks, you can control them with *task execution control calls*; these let you ready, suspend or kill a task. If the task resides in an overlay, or uses mapped extended memory, you can also use task execution control calls to load the overlay, or to remap it into extended memory.

You can control tasks by individual task ID number by using the *task ID calls*. These permit you to ready, suspend or kill a task as well as to change task priority and get the status of a task.

Intertask communication calls allow tasks to communicate by sending and receiving one-word messages in agreed-upon memory locations. One word messages can, of course, be pointers to larger messages if you coordinate this beforehand.

A transmitting task can simply deposit a message in an agreed-upon location (.XMT), or the caller can deposit the message and wait until its receipt (.XMTW). To receive such a message, another task issues a .REC task call. If the transmitting task has not yet sent the message when the receiver's task issues .REC, RDOS suspends the receiving task until the message is sent. If the message has already been sent, the receiving task is readied, and accepts the message. If the .XMTW task call sent the message, RDOS readies both receiving and transmitting tasks before sending control to the Task Scheduler. A single message can't be transmitted to more than one task.

You can also send a message to a receiving task from a user interrupt service routine by means of the .IXMT call.

User clock and user interrupt task calls apply multitasking concepts to nonstandard user devices. Included are calls that provide special power fail/restore service and a *system* call that creates a logical real time clock, which passes control to a specified routine at given intervals. See the section ahead entitled Task Timing Control for more on user clock calls.

Finally, under special circumstances, you may want to disable the *Task Scheduler*. Task calls .DRSCH and .ERSCH disable and re-enable it.

Task-Operator Communications (OPCOM)

The intertask communication calls also facilitate the passing of messages between the console and a task. You enable the Task-Operator Communications Package (OPCOM) by using task call .IOPC. OPCOM permits you to create, examine, modify or kill running tasks from the console with special commands. OPCOM commands give you truly dynamic control over the multitask environment, possibly eliminating source level changes, assembly, and binding of a new program.

Task Timing Control

A task can suspend itself for as long as it wants to by issuing the *system* call .DELAY. This allows you to implement your own time slicing or round-robin allocation of CPU control.

RDOS also maintains a system clock and calendar for those tasks that should be scheduled on a time-of-day basis. Tasks can examine the frequency of the system clock, and can obtain or set the date for the correct time in seconds, minutes, and hours. Moreover, you can schedule any task to gain control at periodic intervals with the .QTSK call. Using these features, you can schedule tasks for such functions as hourly scans or shift reports.

Finally, RDOS provides a pair of system calls, .DUCLK and .RUCLK, that let you define and remove a user clock, driven by the system clock. Your user clock generates interrupts at the intervals you specify. When one of these interrupts occurs, both the Task Scheduler and task environment are frozen, and control goes to your own special routine outside the multitask environment.

Chapter 8

Special RDOS Features

RDOS offers many features that you can use to optimize your system's efficiency: software or hardware options that allow you to attach your own devices to the system, streamline system throughput, extend processing power, or enlarge the process environment. You enable, include or manipulate many of these features in your operating system with the system generation program SYSGEN.

You can implement user *device drivers*, software controlling non-standard devices, either by adding the driver to the operating system with SYSGEN or by introducing the device at runtime. In addition, you can use the RDOS *interrupt priority* system to set up your own scheme for device servicing.

RDOS provides an *automatic restart on power failure* option, which saves the processor state and restarts some system devices. You can extend restart support to other devices with the device driver interrupt software.

Two hardware options, the *Interprocessor Bus (IPB)* and the *Multiprocessor Communications Adapter (MCA)*, permit the configuration of multiple processor RDOS systems. Two processors connected with an IPB can share a disk and monitor each other, running up to four programs at once and providing backup in the event of one processor's failure. With an MCA, up to 15 processors can communicate with each other by way of their data channels.

With the RDOS *tuning* option you can collect statistics on system efficiency and either generate a new system tailored to your demonstrated needs, or direct SYSGEN to *self-tune* on the basis of the statistics. The system elements that tuning surveys are stacks, cells, buffers and overlays.

Each of these features is discussed below; further documentation is in *RDOS System Reference* (DGC No. 093-400027).

User Device Drivers and Interrupts

Many real time applications of RDOS require the use of non-standard equipment, devices that RDOS doesn't normally recognize. RDOS can support user-written *device drivers*, or device service software, for non-standard equipment. RDOS also allows the user to control *interrupt* handling (when a device needs servicing, it generates an interrupt) for system and user devices. You can include your device drivers (also called interrupt service routines), as part of the resident RDOS operating system, or you can implement them as part of your application program.

The following sections outline the RDOS procedures for interrupt servicing, including interrupt priorities; the use of interrupts in a task environment; and device driver implementation in the operating system or at runtime.

Interrupt Servicing

Interrupts can be generated by conditions internal to the computer hardware, for example a power monitor, or by events that originate in the environment that is being controlled, such as an external hardware interrupt like an analog-to-digital converter completion. With the RDOS *interrupt priority* scheme, you can assign various levels of priority to devices on your system, thereby keeping I/O devices active and improving job throughput.

One method of interrupt priority is a function of the hardware itself: when two devices request interrupts simultaneously, the device that is physically closest to the processor on the I/O bus is at the highest priority. Superseding that method, you can specify which devices can interrupt a running service routine by using a 16-bit *priority mask*. With a mask, you can disable any peripheral from interrupting your interrupt while enabling others to do so, thus creating a priority interrupt system.

When the CPU detects an interrupt, it suspends the currently-executing program and gives control to the *interrupt dispatch* program, INTD. INTD is an integral part of the RDOS system and resides in memory at all times. The functions of INTD are:

1. Identify the interrupting device *
2. Save the machine status and all addressable registers *
3. Set up the new priority mask *
4. Direct control to the proper servicing routine *
5. Restore the previous priority mask *
6. Restore the machine status and registers
7. Return to the interrupted program

* The first five functions are performed by the Vector (VCT) instruction on ECLIPSE systems.

INTD directs control to the correct service routine by using the device code received on the interrupt as an index into the *interrupt vector table*, ITBL. The entry in ITBL is the address of the appropriate *device control table* (DCT) for the interrupting device. DCTs, which are built by RDOS for every device that is generated with SYSGEN, contain the mask, the address of the device driver, and device information, such as characteristics, that can be used by INTD.

The interrupt process is demonstrated in Figure 8.1.

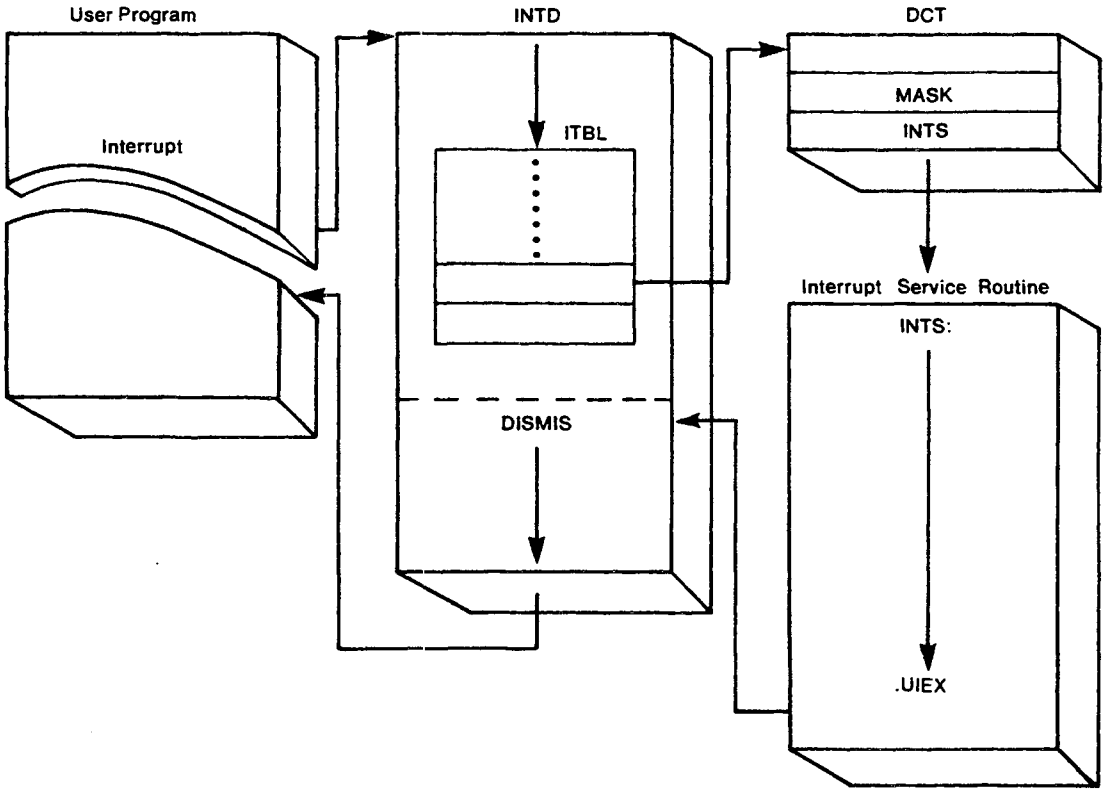


Figure 8.1 Flow of control during interrupts

SD-00685

Interrupts in a Task Environment

Although an interrupt freezes the task environment, you can ready a task for execution by sending it a message from your interrupt service routine. If the task is awaiting a message, it will be readied despite the suspension. If no task awaits the message, your routine will simply continue after posting the message. Task call .UIEX, which you must use to end your routine, offers a choice upon return to the normal environment: depending on the contents of AC1 at .UIEX, task rescheduling will occur, or tasks will resume their former states, upon return from your interrupt.

Device Driver Implementation

You can introduce user devices to an RDOS system in one of two ways:

- Incorporate the driver into the operating system at the source level,
- Identify the driver to the operating system at runtime with the system call .IDEF.

Because a driver built into the operating system has access to system routines and databases, the first method provides more efficient device interrupt service than the second. Implementation at source level, which requires extensive understanding of the operating system and device driver structure, comprises these steps:

1. Declare an entry for the device in the Device Control Table (DCT).
2. Enable entry of the device name in SYS.DR.
3. Update the system libraries.
4. Create a system queue entry.
5. Perform a system generation (SYSGEN) and load.

The second method of driver implementation, introducing the device with. IDEF, is easier, though less runtime efficient because the driver must duplicate some system functions. With .IDEF, you can introduce both character and data channel devices. Up to 64 user devices can be identified on a mapped system at any moment.

Before using .IDEF for your device driver, you must choose a mask and construct a DCT for the device. For drivers introduced at runtime, only the first three words of a DCT are needed. They are:

Word	Mnemonic	Purpose
0	DCTBS	Reserved for system use.
1	DCTMS	Interrupt mask.
2	DCTIS	Address of device interrupt service routine.

You then identify your device to RDOS with system call .IDEF. .IDEF places an entry in the interrupt vector table, ITBL, thereby allowing the interrupt dispatch program, INTD, to reach the DCT, and thence your device interrupt routine.

Power Fail Handling

RDOS provides automatic restart at power failures, an option that you enable with SYSGEN. If you enable automatic restart and the CPU key switch is in the lock position, the system, when it detects a power loss, transfers control to a power fail routine that saves the state of the accumulators, the program counter, and the carry bit. Refer to the appriate CPU manual for details on automatic restart at power failure.

> **NOTE:** *For systems with semiconductor memory, you can enable automatic restart only if the system also has battery backup.*

RDOS gives power-up restart service to these system devices:

Teletypewriter and CRT,
Disk,
Multiplexor,
Line printer,
Paper tape punch/reader,
Card reader,
Plotter.

You can provide power-up service for magnetic or cassette tapes, or for your own device, by writing an interrupt service routine using .IDEF. (See the previous section entitled User Device Drivers and Interrupts.) To exit from your power-up service routine you use task call .UPEX, which forces rescheduling.

Multiple Processor Systems

Two hardware options allow the configuration of multiple processor systems running RDOS: The Interprocessor Bus (IPB) allows two processors to exchange line and sequential I/O, to share a disk, and to monitor each other with an interval timer. The Multiprocessor Communications Adapter (MCA) allows up to 15 processors to communicate by means of their data channels; the MCA also allows foreground and background programs to communicate at data channel speeds.

In one type of dual processor application that greatly extends processing power, the two CPUs run independently; each can run a foreground and background program, and all four programs can communicate with one another through the shared disk.

Another dual-processor application is a backup system, allowing real-time operations to continue if one CPU fails. In such a system, the main system controls or monitors some process (or series of processes) continuously, and the backup system stands ready to assume the main system functions in the event of failure in the main system. The IPB interval timer detects any main system failure and signals the backup system to take control. While in the standby state, the backup system can handle lower-priority tasks such as data analysis or program development. Both the main and the background systems can run foreground and background programs, or simply a background program.

A multiple processor application with more than two processors typically involves a master-slave arrangement among the CPUs. Each slave CPU in such a system can run unique or redundant tasks, and all are managed by a supervisory CPU. All processors in the system can run foreground and background programs, or only background programs. Each processor can communicate with the others through the MCA. In a variation of this configuration, two master CPU's, linked by an IPB, can monitor each other and either can take control if the other fails.

To run a multiprocessor system with either an IPB or MCA, each CPU must boot an operating system from a separate disk partition, and each partition must have its own copy of the RDOS system and CLI.SV, CLI.OL and CLI.ER. You configure an RDOS system to support either an IPB or MCA, or both, with SYSGEN.

The capabilities of each of the devices are outlined below.

The Interprocessor Bus (IPB)

The IPB (Model 4240), for linking two processors, provides one full-duplex line for sequential and line I/O between two processors. The read and write operations take place through special device filenames, $DPI for input and $DPO for output; each $DPO is a spoolable device. An additional half-duplex line allows RDOS to ensure that processors sharing disk partitions do not simultaneously modify the SYS.DR or MAP.DR of any partition.

The IPB interval timer, for system monitoring, generates an interrupt request if either processor fails to service its real-time clock every second. Where two processors are linked by an IPB and share a disk, you can issue system call .BOOT to shut down the current RDOS system and bootstrap another operating system or a suitable user program. With special programming measures, you can .BOOT a system without operator intervention, and register the correct date and time for a real-time process.

The Multiprocessor Communications Adapter (MCA)

The type 4206 MCA receiver/transmitter (MCAR/MCAT) allows programs to communicate over full duplex lines, in blocks of up to 8192 bytes, through the data channel. Each program can exist within the program space of a single CPU, or within up to 14 other CPUs, or both. A second 4206 receiver/transmitter (MCAR1/MCAT1) provides as many as 15 additional communications links. Each CPU can communicate with any other CPU.

Each MCA line has a filename, differing according to whether it is transmitting or receiving. Your program can access the line filenames by system calls, and you can access them by some CLI commands.

With the CLI command MCABOOT, you can transfer and bootstrap a copy of an RDOS system, or a suitable standalone program, to another processor's disk. Additionally, before sending the system, you can fully or partially initialize the receiver's disk with MCABOOT.

Tuning an RDOS System

RDOS provides a feature called *tuning* that permits the operating system to monitor its own efficiency and to suggest adjustments that will produce a more efficient system. You enable tuning in SYSGEN, then turn tuning on and off at runtime with CLI commands or system calls. The tuning mechanism, when on, records in a disk file the number of instances when a system stack, cell or buffer is unavailable to RDOS because too few of them were allocated during system generation. Additionally, the tuning function can record the number of times RDOS requires a system overlay but must read it from the overlay file because it is not core-resident.

The tuning report can be used in two ways. First, you can analyze the report and generate a new operating system based on your judgment of whole-system needs. Second, with the feature called *self-tuning*, you can generate a new system in tuning mode, where SYSGEN examines the tuning report file and builds a system with more appropriate numbers of system stacks, cells, and buffers, but not overlays. With self-tuning, SYSGEN has a limited view and can't know the impact of the modified numbers of system stacks, cells, and buffers on other factors in your application. You can produce the most efficient system through comparative timings of different system configurations, guided by the tuning report.

The following sections discuss system stacks, cells, buffers and overlays and describe the enabling of tuning in SYSGEN, how to turn the feature on and off, and the tuning report file.

System Stacks, Cells, Buffers, and Overlays

Because RDOS is partly core-resident and partly disk-resident, it has features ordinarily found on larger operating systems, while its total memory needs are modest. Stacks, cells and buffers are memory-resident parts of RDOS, and system overlays are called into memory from disk when needed. Tuning, therefore, can help you to reduce total memory needs where these system elements are overabundant, or to increase system efficiency where more elements will reduce delays.

RDOS uses system stacks as databases in the execution of system calls, disk I/O and spooling. Concurrent system calls and concurrent disk I/O require multiple stacks; at a single moment, RDOS services only as many requests as it has system stacks available, in the order that these calls are made. SYSGEN allows you to specify one to ten system stacks.

System cells are control tables that RDOS uses primarily to store system task state information. RDOS also uses cells to manage read/write block operations, stacks, the Interprocessor Bus (IPB), spooling, and active system calls. SYSGEN automatically allots cells for an IPB, if you have specified one, and for read/write block operations; you can add up to 64 more cells.

RDOS uses system buffers for two functions: first, to receive system overlays, which are segments of operating system code that are stored on disk and called into memory when needed; and second, to buffer all I/O except read/write block operations. When RDOS needs a buffer, it flushes the contents of the oldest idle buffer; however, if extra buffers are available, fewer are flushed and their contents remain available. Extra buffers accelerate access time to buffered data and make system call execution faster, but reduce the total amount of memory available to your programs. SYSGEN automatically allots a minimum of two buffers per system stack, or six buffers total, whichever is greater. You can specify as many extra buffers as core memory allows.

Initiating Tuning

SYSGEN asks two questions about tuning; in answer to the first, you enable the system to report statistics on system stacks, cells, and buffers; in answer to the second SYSGEN question, you enable the system to append system overlay statistics to the report.

You turn tuning on and off with the CLI commands TUON and TUOFF, or with system calls .TUON and .TUOFF. No statistics are collected until you activate tuning. Having the ability to turn tuning on and off, particularly with system calls, permits you to pinpoint suspected areas of your application. When active, the system stack, cell, and buffer reporting function consumes one system buffer, and the system overlay reporting consumes two additional system buffers.

The Tuning Report

Tuning reports its statistics in a contiguous disk file in the master directory. The file is called Sys.TU or Sysname.TU, where Sysname is the descriptive (non-default) name of the running operating system. The tuning report consists of one disk block if you have specified only system stack, cell, and buffer reporting; if you have specified system overlay reporting, two blocks are added to the file.

The first block of any tuning report summarizes this information on system stacks, cells and buffers:

Number of resource in system,

Number of requests for resource,

Number of faults (unsuccessful requests for resource).

If you have specified system overlay reporting in SYSGEN, tuning appends these statistics to the first block of the report, in a format identical to the others. Further, tuning adds two more blocks that detail the overlays separately; up to 128 system overlays can be detailed in the report.

With the report, you can judge the ratio of requests to faults as a measure of system efficiency, and modify your operating system accordingly. You can print the tuning report with the CLI command TPRINT; you can also examine it from a program with the system call .RDS.

Chapter 9

Assemblers and Program Utilities

Assembly language programming in RDOS gives you the use of the instruction set of your computer and an easy interface with the operating system, both advantages in any complex, real-time or machine-level programming.

The symbolic instruction code and symbolic addresses of user source code are translated into machine language by either of the two RDOS assemblers, the Extended Assembler (ASM) or the Macroassembler (MAC). Macroassembly permits you to write a set of general or frequently-needed code, and then to call that code at any time, inserting specific information where needed.

Assembler pseudo-ops allow you to direct the operation of the assembler. Output from the assemblers includes a source program listing and an error listing in addition to the assembled code.

The Relocatable Loader, RLDR, loads assembled code modules into executable code. The Library File Editor, LFE, lets you store assembled modules in a library for later use with RLDR. The Overlay Loader, OVLDR, lets you replace existing overlays with different overlays.

Debugging tools include the Symbolic Debuggers (DEBUG and IDEB), which facilitate breakpoint debugging; the Symbolic Editor (SEDIT), which handles symbolic editing of disk files; the stand-alone Disk Editor (DISKED),which enables disk word searches, reads and writes in a variety of formats; and the patching utilities (ENPAT and PATCH), which help you to encode and then patch save and overlay files.

This chapter can be read in conjunction with Chapter 7, Memory Management and Multitasking. All the material in this chapter is documented in *RDOS/DOS Program Utilities*, (DGC No. 069-400019) and *RDOS/DOS Debugging Utilities* (DGC No. 069-400020).

An example of assembly language source code appears in Appendix G, along with source code examples of the RDOS high level languages.

The Assemblers

You can choose between the two assemblers, the Extended Assembler (ASM) and the Macroassembler (MAC). Both are similar in their operation, though MAC is more powerful and more flexible, while ASM is faster. MAC allows you to write frequently-needed code into a set of instructions that you can call at will. The term assembler in this chapter applies to both ASM and MAC.

The assembler translates your symbolic instruction code and symbolic addresses into machine-readable numeric codes and numeric addresses.

Assembly language source code employs symbolic instruction codes (such as LDA 0,2 for "load the contents of location 2 into accumulator 0") and operating system calls (such as .RDL for "read a line"). You add your own symbols in source code in the form of memory location names (such as TEMP), which you subsequently reference by name. Source code also uses assembler pseudo-operations (pseudo-ops), which direct the operation of the assembler, but which do not become executable code.

Most Data General machine instructions assemble into one 16-bit storage word. There are three types of instructions:

- Memory Reference Instructions (MRI), which access memory locations or their contents by means of one of four different indexing modes.
- Arithmetic and Logical Instructions (ALC), which can add and subtract values, shift bits, swap bytes, use the overflow (Carry), AND values to mask words, and redirect program execution.
- I/O instructions, which govern the operation of all system devices. Generally, you use these instructions only to write device interrupt handlers.

Macroassembly

Programs often use the same set of instructions many times. Macroassembly permits you to write a set of instructions only once, and call this set wherever you wish during assembly of the source with MAC. The basic steps of macroassembly are:

1. You write a set of symbolic instructions, called a macro definition, and give the macro definition a name.

2. Wherever you want that set of instructions in your source file, you insert a macro call. At minimum, the macro call contains the name of the macro definition.

3. The assembler contains a macro processor that substitutes the sequence of instructions (macro definition) for the macro call. This substitution is called macro expansion.

The macro facility offers additional sophisticated features. For example, the same set of instructions, differing only in accumulators and addresses, might be used many times within a program. If so, you can write formal (dummy) arguments for the accumulators and addresses into the macro definition. You add the actual arguments to your macro call in the source. During assembly, MAC substitutes the actual arguments for the dummies. Thus, a macro definition is usually a skeleton of the actual instruction set that results from macro expansion.

Pseudo-operations

A pseudo-op directs the operation of the assembler and it is called a pseudo-operation because your program never actually executes it. You use pseudo-ops to indicate such things as absolute, normal relocatable and page zero relocatable code (.LOC, .NREL and .ZREL), or to declare entry points or symbols to interface with other modules (.ENT, .EXTD, .EXTN).

Conditional assembly pseudo-ops provide you with the capability to select portions of source code for the assembly process. Depending on the evaluation of an absolute expression, the assembler either assembles or bypasses a section of source code.

Interprogram communication facilities define data addresses and constants in one program and reference them in another. By using the interprogram communication pseudo-ops, you can write related subprograms without concern for the absolute locations of data and addresses shared by the programs at runtime.

Assembler Input and Output

You input your source code to the assembler, which reads the source line-by-line and translates it into binary machine language code. The assembler reads each source program twice; each read is called a pass. At the end of the second pass, the assembler produces one or more of the following:

- Relocatable binary file (.RB),
- Source program listing file,
- Error listing.

Relocatable Binary File (.RB)

When translating source code into binary output, the assembler builds syntactically recognizable elements called atoms. Atoms consist of numbers,symbols, operators, break characters, or special characters.

The binary output is a translation of source lines into blocks of binary code. Most lines of source input translate into single 16-bit (one word) binary numbers for input to RLDR. The assembler gives each number an address, which can be a relative address that RLDR will relocate. The assembler produces as part of the binary file the information that RLDR needs to map each address and its contents.

A relocatable binary file can have three different sections of code: absolute, ZREL (page zero relocatable) and NREL (normal relocatable), each of which is specified in your source program with pseudo-ops.

You can choose not to output a binary file in the assembly process.

Source Program Listing

The program listing permits you to compare your input against the assembler's reading of it. For each line of source code, a line of a program listing can contain the following elements:

- Program line number, or, if there are errors, up to three alphabetic error codes;
- Location counter and relocation mode, if relevant;
- Relocation flag pertaining to the location counter;
- Data field and relocation mode, or value of equivalence expression, or pseudo-op argument;
- Relocation flag pertaining to the data field;
- Source line as written, or as expanded by macro calls

A program listing always includes a cross-reference listing of the symbol table. Cross-reference listings display your program's symbols, symbol addresses and types, relocation

flags and the page and line where the symbols appear. Cross-reference symbols include user and macro symbols as well as symbols for these pseudo-ops: entry (.ENT), external normal (.EXTN), external displacement (.EXTD), overlay entry (.EXTO) and named common (.COMM).

In the assembly process, you can choose to suppress certain lines of the listing, for instance macro expansion, and you can also choose not to output a program listing.

Error Listing

The error listing produced by assembly contains the title of the source program and all source lines that have been flagged with an error code. If a source program listing is output, the error listing is optional. During the second pass, all errors in the source program are included as part of the program listing. If a program listing is not output, a separate error listing is produced automatically. The error listing output on the second pass lists only those lines containing errors.

The format of error listing lines is the same as that of the source program listing. The error listing is useful in programs with very long listings, since it serves as an abstract, but it adds no further information to that given in the program listing.

Software Packages

Two Data General software packages that can be used with assembly language programs are the Sensor Access Manager (SAM) and the Satellite Processor Software Interface.

SAM is a library of device handlers and subroutines that manage process I/O. SAM works in conjunction with a chassis and a group of cards which are collectively called Data General/Data Acquisition and Control (DG/DAC). SAM can be used with FORTRAN IV and FORTRAN 5, as well as assembly language.

The Satellite Processor Software Interface allows one or more DG Satellite Processors (64 Kbyte ECLIPSE computers) to exchange information with a host ECLIPSE computer. SP software allows the host and the satellite processor programs to work in parallel and to transfer data to one another. The host program runs under RDOS while the satellite processors run under the Real-Time Operating System (RTOS). The SP Software Interface is able to be used by FORTRAN 5 and DG/L in addition to assembly language.

Invoking the Assemblers

You invoke the Extended Assembler with the CLI command ASM, and the Macroassembler with the CLI command MAC, citing in the command line the appropriate input filenames and switches to control output.

Programming Tools

The Relocatable Loader, RLDR, transforms assembled code (.RB files) into executable code. For efficient storage and convenience, you can store .RB modules in a library file (.LB) by using the Library File Editor, LFE. You can select modules from a library file for loading in the RLDR command line. The Overlay Loader, OVLDR, lets you replace existing overlays in an overlay file (.OL) with one or more different overlays.

RLDR

Both ASM and MAC are relocatable assemblers, meaning they assign to each storage word a relative location counter value. The Relocatable Loader, RLDR, takes each relative value and gives it an absolute memory address, producing executable code (.SV files).

The assembler produces output that is placed by RLDR for execution in either the absolute, the ZREL or the NREL sections of memory. The assembler assigns relative values by maintaining three separate counters for each type of relocatability (absolute, ZREL and NREL code).

The role of RLDR is to accept a number of assembled modules and form a non-overlapping save file for execution. RLDR accomplishes this by taking the assembler's relative addresses and making them into absolute addresses, using its own counters.

Like the assembler, RLDR maintains three counters for the three types of code. Using its counters, RLDR establishes an absolute address for each relative address of the modules it processes, incorporating into NREL the length of the User Status Table (UST), the number of tasks specified, and the size and number of overlay nodes in the program. RLDR assigns each symbol an absolute address by adding the symbol's relative address to the ZREL and NREL counters. After loading each module, RLDR updates its counters to include the number of ZREL and NREL words used by that module, thus setting up the starting addresses of absolute memory for the next module. In this way, RLDR loads any number of separately assembled modules together, without storage conflict.

See Chapter 7 on memory management for more information on RLDR.

Using RLDR

You invoke the Relocatable Loader with the CLI command RLDR, including on the command line the appropriate binary, overlay binary, and library filenames. You assign nodes to overlays in the command line. RLDR scans the system library, SYS.LB, at the end of the command line, and whenever you place its name in the command line; you can direct RLDR not to search SYS.LB at the end.

With global switches in your RLDR command line, you can

- Produce an alphabetical symbol listing,
- Create the .SV file to execute under RTOS,
- Include a debugger and symbol table in the .SV file,
- Display RLDR error messages at the terminal,
- Control warning messages for overlays,
- Produce hexadecimal numeric output instead of octal,
- Suppress system tables and alter NREL and ZREL starting locations,
- Retain the RLDR symbol table file,
- Suppress output to the terminal,
- Display the starting NREL address of each RB or library module,
- Place the symbol table in high memory.

With local switches you can further control RLDR output and manipulate starting locations. These switches also enable you to allot I/O channels, to allot Task Control Blocks (TCBs) for multitasking programs, and to load overlays as virtual overlays (mapped RDOS).

The Library File Editor (LFE)

A library file consists of a set of relocatable binary files (.RBs) that you store for later use. The Library File Editor helps you to build and maintain library files composed of binary files that you produce with ASM or MAC. The binary files are organized in logical records within a library, and each binary file is separated from others with special start and end blocks. RLDR can extract binary files from your libraries for loading into an executable program.

LFE allows you to perform the following functions:

- Create a new library from one or more binary files;
- Merge existing libraries into a new library;
- Insert binary files into a new or existing library;
- Replace logical records in a library with new binary files;
- Extract specific logical records from a library;
- Delete logical records from a library;
- Analyze a library, records within a library, or a binary file;
- List titles of records in libraries or binary files.

LFE is particularly important in helping you to order the sequence of relocatable binaries in your library, since RLDR uses this order to determine which programs to load.

You invoke the LFE utility with the CLI command LFE, adding the appropriate function keys and switches.

The Overlay Loader (OVLDR)

The overlay loader, OVLDR, allows you to replace existing overlays in an overlay file (.OL) with one or more different overlays. For this process you use OVLDR to create a file containing the replacement overlays, naming in your OVLDR command the associated save file, the node number or overlay descriptor of the overlays to be replaced, and a list of .RB files that are to replace the current overlays. Then you use the CLI command REPLACE, which uses the OVLDR-produced replacement file to replace individual overlays in the existing .OL file.

Debugging Tools

RDOS provides five utilities that assist you in program debugging and patching: the Symbolic Debugger, the Symbolic Editor, the Disk Editor, and the patching utilities ENPAT and PATCH.

Symbolic Debuggers (DEBUG and IDEB)

Debugging is the process of detecting, locating and removing mistakes from a program. To debug your program, you load a Symbolic Debugger with your program, using RLDR. You can then control program execution, causing the program to halt in the debugger at one or more points, so that you can examine the contents of memory locations and special registers (such as the accumulators), and correct the contents. Because the debugger is symbolic, you can examine memory and register contents in source language format, though you can also use octal format or other special formats.

The debugger lets you set up as many as eight active breakpoints within a program. Program execution halts just before the instruction at the breakpoint is executed, and you can then issue debugging commands. You can restart the program at the breakpoint instruction, or at any other location.

Data General supplies two versions of the debugger with RDOS. The first, called DEBUG, allows interrupts during debugging. The second, called IDEB, disables interrupts during debugging. The versions of DEBUG and IDEB are tailored to NOVA or ECLIPSE, mapped or unmapped configurations.

Symbolic Editor (SEDIT)

With the Symbolic Editor, SEDIT, you can edit disk file locations symbolically in any random or contiguous file created on RDOS or DOS systems. SEDIT is compatible with the RDOS debuggers (DEBUG or IDEB, NOVA or ECLIPSE, mapped or unmapped) and shares many commands with them.

SEDIT searches the disk file for symbolic values, and allows you to display and modify locations in octal, instruction, system call, byte, or ASCII format. You can display whole blocks of words, and use special registers to find and modify locations. SEDIT works best on text files and on programs that have been loaded with a program symbol table. SEDIT can also access the overlay file of a program from its save file.

Disk Editor (DISKED)

The stand-alone DISKED utility enables you to examine, modify and rearrange disk words. (A stand-alone program runs without an operating system.) With DISKED you can search for a specific word, fill different parts of the disk with a word, display any part of the disk, or dump part or all of the disk to an output device. DISKED offers seven different output formats and a choice of number bases ranging from 2 to 16.

Patching Utilities (ENPAT and PATCH)

A patch is a one-word change made in a save or overlay file. You can use ENPAT to systematically encode and enter data into a patch file, then install the patch file with PATCH. Patches are most often used to update operating system files; Data General supplies current patches with its system software. You can use ENPAT and PATCH for your own save and overlay files.

ENPAT checks syntax on all data input, while PATCH evaluates expressions. ENPAT allows you to encode a multi-word patch, and enables conditional installation of a patch based on the presence of a particular symbol in the loadmap. With ENPAT, you can also enter a question for conditional installation that is later output in the PATCH dialogue. Multi-word patches can be marked as a group for conditional installation.

PATCH installs the patch file created by ENPAT. When PATCH runs, it creates a patch dialogue file that records date and time, a list of all locations patched, and a string indicating whether the patch was installed.

Chapter 10

RDOS High Level Languages

High level languages remove the programmer from the machine level instruction set of assembly language, offering expressions and syntax that are more easily recognized, and suited for a variety of programming applications. The Data General high level languages available under RDOS are FORTRAN IV, FORTRAN 5, DG/L, Interactive COBOL, Extended BASIC, and Business BASIC.

FORTRAN IV, FORTRAN 5, and DG/L are compiled languages, for which your source code is translated first into assembly language and then to machine language by a special program, called a compiler. DG's FORTRAN IV compiler is largely compatible with the American National Standards Institute (ANSI) 1966 standards, and features many DG extensions. The FORTRAN 5 compiler is a superset of FORTRAN IV, with extensions increasing its power. DG/L is an ALGOL-derivative systems development language.

Interactive COBOL, Extended BASIC, and Business BASIC are interpretive languages, in which you write, edit, debug and run your program interactively within the language. Interactive COBOL is designed for business use, and features multi-user programming and sophisticated screen formatting. Extended BASIC features many DG extensions to the original Dartmouth BASIC, though it is not compliant with ANSI standards. Business BASIC is designed for commercial applications. Both BASIC interpreters permit multi-user programming under RDOS.

The RDOS languages run on some other Data General operating systems, and some applications software can be run directly or recompiled to run on different operating systems and hardware configurations. The Data General operating systems involved are the Real-Time Operating System (RTOS), the Disk Operating System (DOS), the Microproducts Operating System (MP/OS), the Advanced Operating System (AOS), and the Advanced Operating System/Virtual Storage (AOS/VS).

This chapter can be read in conjunction with Chapter 7, for information on memory management and multitasking, and Chapter 9, for information on program utilities and debuggers.

Examples of source code for each of these languages appears in Appendix G, along with an example of assembly language source code.

The FORTRAN Compilers

Data General offers two FORTRAN compilers for use under RDOS, FORTRAN IV and FORTRAN 5. FORTRAN IV largely conforms to the American National Standards Institute (ANSI) 1966 standard for FORTRAN, and contains some Data General extensions to increase the flexibility and power of the language. FORTRAN 5 is a superset of FORTRAN IV, in addition adhering more to the ANSI 1966 standard and containing more extensions. FORTRAN IV is a fast, one-pass compiler that performs some local optimizing, while FORTRAN 5 is a globally optimizing compiler that produces in-line code.

Uses of FORTRAN

FORTRAN was originally designed for technical scientific applications, but has become widely used in the fields of education, the social sciences, industry, and business. FORTRAN's easy syntax, computational abilities and formatted output, which make it an ideal scientific tool, are also useful in such endeavors as economic modeling, market analysis, forecasting, accounting, and clerical duties.

Data General's extensions to FORTRAN, in particular multitasking, enhance FORTRAN's commercial applicability, for example permitting one program to perform all aspects of inventory control. Our Commercial Subroutine Package (CSP) further enhances FORTRAN for the business world.

Multitasking, combined with FORTRAN's re-entrant code, also facilitate real-time uses, with such applications as industrial or laboratory process control, weather forecasting, navigation, and medical scanning systems.

Common Features of the FORTRAN Compilers

The FORTRAN statement function and function subprogram features allow you to define a formula as a function only once, subsequently using the function in place of the formula. You can also code a function as a separate program unit that other programs can call.

With FORTRAN subroutines, you can organize large programs in modules; the subroutines are easily updated, and can be called by other programs.

Data General's FORTRAN multitasking capability lets you take advantage of the RDOS task scheduling system. Through FORTRAN subroutine calls you define tasks and their relative priority. FORTRAN allocates separate space for each task through a runtime stack, which holds task-related data such as program variables. Because FORTRAN code is re-entrant, tasks can share code.

The entire RDOS file structure is available for FORTRAN users. You can create, delete and specify characteristics for sequential, random and contiguous files. System devices and file directories can be defined, initialized and released. Since files created by FORTRAN are identical to those created by other languages, they can be accessed mutually.

You can call assembly language subroutines from FORTRAN, a feature especially useful for device-level programming or special data manipulation. You can pass arguments to assembly language subroutines and give them access to data stored in FORTRAN shared common areas, blank or named. You can also use RDOS system calls from FORTRAN programs.

FORTRAN IV

FORTRAN IV supports standard formatted I/O, free-formatted I/O, and binary I/O. The language includes full sets of relational and arithmetic operators, and allows single- and double-precision arithmetic on real or complex variables.

You can insert lines of assembly language in your program, starting each line with an A in the first column. The assembly language statements pass directly to the assembler.

FORTRAN IV is designed for quick compilation with extensive error checking and local optimization. It uses subroutines to check errors, to check for underflow or overflow, to divide by zero, to decode addresses, and to save arguments.

FORTRAN IV runs under RDOS, DOS, RTOS and AOS. Under RDOS, it runs on the NOVA or ECLIPSE computer. You can program FORTRAN IV to use any hardware multiply/divide and floating-point instructions available on your computer.

For documentation see the *FORTRAN IV User's Manual* (DGC No. 093-000053), *FORTRAN IV Runtime Library User's Manual* (NOVA) (DGC No. 093-000068), and *FORTRAN IV Runtime Library User's Manual* (ECLIPSE) (DGC No. 093-000142).

FORTRAN 5

FORTRAN 5 is a superset of FORTRAN IV, conforms more closely than FORTRAN IV to the ANSI 1966 standards, and features more Data General extensions, particularly in multitasking and extended memory use.

With multitasking in FORTRAN 5, you can vary the stack size of each task according to your needs, and you can define events in order to synchronize tasks.

FORTRAN 5 allows you to use extended memory outside logical user space. You can use FORTRAN 5 calls to define a window map for storing and accessing large amounts of data.

Other extensions to FORTRAN 5 are mixed-mode arithmetic, bit and byte manipulation, and high level debugging aids.

The FORTRAN 5 compiler's global optimization produces more in-line code and ensures fast execution. The optimization performs such functions as removing unused variables and parameters from the final code, and moving unchanging code from DO loops.

FORTRAN 5 runs under RDOS, RTOS and AOS, and in 16 bit-mode under AOS/VS. FORTRAN 5's hardware requirements are a NOVA equipped with a hardware floating-point unit (FPU) *and* a hardware multiply/divide unit (HMPYD); or an ECLIPSE equipped with FPU *or* the floating-point instruction set (FIS).

FORTRAN 5 is documented in the *FORTRAN 5 Reference Manual* (DGC No. 093-000085) and *FORTRAN 5 Programmer's Guide (RDOS)* (DGC No. 093-000227).

DG/L Compiler

Data General's DG/L Systems Development Language is an ALGOL-derivative, structured programming language designed for a wide variety of applications. DG/L's multitasking runtime environment and memory management features suit it for a multiple-user application. Arithmetic and mathematical library functions facilitate computation tasks, while the string handling and string arithmetic features enable commercial programming. These features and others make DG/L ideal for the development of systems-level software such as compilers, assemblers, or sort/merge programs.

DG/L's globally optimizing compiler produces code that is re-entrant, recursive and load time relocatable. Multitasking capabilities allow you to access all the task creation, control and synchronization features of RDOS.

The structured nature of DG/L encourages top-down programming in which the program is logically divided into progressively subordinate functions. You can write DG/L code in segments called procedures, defining procedures within other procedures to develop a tree-structured program. A procedure can reference all other procedures at the same level, or any procedure subordinate to itself. Since all DG/L code is re-entrant and recursive, a procedure can also reference itself.

Memory management tools of DG/L include the ALLOCATE and FREE statements, which allow you to assign then release memory not taken by program code or stack space, thus enabling multiple tasks to share successively a large block of memory. The cache memory manager lets you handle data structures larger than your program's address space, allowing you to define the word size of elements, and then to access up to 64 K elements in a cache memory structure. The data is stored in a disk file and buffered through memory on a least-recently-used basis, thus speeding data access.

DG/L's variable scoping feature also helps to conserve memory by dynamically allocating and freeing storage areas required by variables local to a procedure, as the procedure is entered and exited.

DG/L permits you to manipulate storage addresses directly to create parameter packets, linked lists, and other systems programming constructs. You can also perform variable manipulation, by which a variable declaration requires no storage but uses an address pointer to access a storage template at runtime.

String management functions include substring and index runtime routines that allow chain linking and string parsing. DG/L supports string variables in arithmetic operations with automatic decimal point alignment and arbitrarily large precision.

DG/L supports single- and double-precision integer arithmetic. On computers with a hardware floating-point feature, single- and double-precision real arithmetic is optionally supported.

DG/L programs can be written, compiled and debugged on an ECLIPSE computer under AOS/VS, AOS or mapped RDOS, then transported and run on an ECLIPSE, NOVA or microNOVA computer under RDOS, DOS, RTOS, AOS/VS or AOS. This broad transportability is enabled by DG/L's comprehensive operating system interface, which allows you to program in a single form while the compiler and runtime library provide the proper object time system interface. Conditional compilation is possible, and any operating system function accessible from a system call in assembler language can be coded in-line with DG/L's SYSTEM runtime function.

Documentation on DG/L includes *DG/L Language Reference* (DGC No. 093-000229) and *DG/L Runtime Library User's Manual* (DGC No. 093-000124).

Software Packages

FORTRAN IV, FORTRAN 5, DG/L and assembly language interface with many Data General language-related libraries and utilities, further extending their usefulness for scientific and commercial programming. Table 10.1 displays the packages that are callable from each language, and the packages are discussed below.

Software	FORTRAN IV	FORTRAN 5	DG/L	Assembly Language
APS		x	x	
CAM	x	x		
CSP	x	x		
Dataplot	x	x		
IMSL		x		
SAM	x	x	x	x
SP Software		x		x
X.25	x	x		

Table 10.1 Software packages

Array Processor Software (APS)

The Array Processor Software is designed to simplify programming for the ECLIPSE AP/130 Array Processor and ECLIPSE computers configured with Integral Array Processors. These processors perform high-speed computations on structured data (real or complex arrays, for example), using sophisticated signal and array processing instruction built into the hardware.

APS runs on mapped RDOS and gives you access to array functions from either FORTRAN 5 or DG/L programs. For documentation, see the *Array Processor User's Manual* (DGC No. 093-000169).

Communication Access Manager (CAM)

The Communication Access Manager provides a series of concise program-level calls for the management of data transfer over Data General's Communication System (DG/CS). Both FORTRAN IV and FORTRAN 5 have access to CAM through subroutine calls. CAM is documented in the *Communications Access Manager Reference* manual (DGC No. 093-000183).

Commercial Subroutine Package (CSP)

The Commercial Subroutine Package is a library of subroutines that facilitate many commercial programming needs, such as string manipulation, editing, and double word integers. CSP is available to both FORTRAN IV and FORTRAN 5. For documentation, see the *FORTRAN Commercial Subroutine Package* (DGC No. 093-000107).

Dataplot

Dataplot is a library of subroutines that allow you to interface with an incremental plotter. You can use Dataplot from FORTRAN IV and FORTRAN 5. Documentation includes the *Dataplot User's Manual (NOVA)* (DGC No. 093-000060) and *Dataplot User's Manual (ECLIPSE)* (DGC No. 093-000156).

International Mathematics and Statistics Library (IMSL)

The International Mathematics and Statistics Library is a comprehensive set of FORTRAN subroutines covering a broad range of mathematical and statistical functions. IMSL can be used with FORTRAN 5. Contact your local customer service engineer for information on the use of IMSL.

Sensor Access Manager (SAM)

The Sensor Access Manager is a library of device handlers and subroutines that manage process I/O. SAM works in conjunction with a chassis and a group of cards collectively called DG/DAC. SAM can be used with FORTRAN IV, FORTRAN 5, or assembly language.See the *Sensor Access Manager (SAM) User's Manual* (DGC No. 093-000225) for documentation.

Satellite Processor Software Interface

The Satellite Processor Software Interface allows one or more Data General Satellite Processors (64 Kbyte ECLIPSE computers) to exchange information with a host ECLIPSE computer. SP software allows the host and the satellite processor programs to work in parallel and to transfer data to one another. The host program runs under RDOS while the satellite processors run under RTOS. The SP Software Interface is usable by FORTRAN 5, DG/L and assembly language. The interface is documented in the *Satellite Processor Interface User's Manual* (DGC No. 093-000221).

X.25

X.25 is Data General's software package for interface with a public packet switching data network, or for networking DG computers. Data General's X.25 is compliant with Recommendation X.25 of the CCITT (Comité Consultatif International Téléphonique et Télégraphique). Both FORTRAN IV and 5 can use X.25. The X.25 software is documented in the *X.25 Protocol Use. s .anual (RDOS/DOS/RTOS)* (DGC No. 093-000173).

Interactive COBOL

Data General's Interactive COBOL is an ANSI 1974 Level 1 implementation with many Level 2 features and Data General extensions. COBOL is the most widely used data processing language for business purposes, offering many advantages for implementing applications systems. COBOL syntax resembles that of everyday language, making programs easier to document and maintain. COBOL'S support for structured programming encourages the creation of programs that are easy to customize, analyze, and maintain.

Interactive COBOL includes extensions that make it useful for the creation of interactive, transaction-driven systems:

- Full screen management from within the program makes it easy to base an applications system on menus and a variety of data-entry/inquiry formats.
- Special display terminal functions such as selective field blinking and sounding a tone enhance the interaction between the program and the user.
- File/record locking and the ability to identify and distinguish terminals provide for orderly data management and system security.
- Program-defined function keys allow operators to call important procedures with a single keystroke.
- A single program controls initial access to all applications programs, protecting the system against unauthorized use and providing selective access to procedures.

The extensions are implemented as enhancements to existing programming constructs, thus making Interactive COBOL immediately accessible to COBOL programmers. For example, screen format definitions are coded as data-item descriptions in the Data Division, and interactive I/O, including function key use, is controlled by the DISPLAY and ACCEPT verbs.

Data General's Interactive COBOL is an *interpretive* language: you write and edit your source code with Interactive COBOL utilities, then use the compiler to produce code that can execute under the Interactive COBOL runtime system. The runtime system also features a debugger.

Interactive COBOL supports all the highest level ANSI COBOL file structures: indexed, relative, and sequential. Files can be converted from one type to another with the utility program REORG.

Sophisticated file access capabilities are also implemented. Indexed files allow rapid access to individual records according to the value of a record key. Each indexed file can

have up to four alternate keys in addition to the primary key, allowing access by as many as five different attributes.

Interactive COBOL programs can create and delete files dynamically, permitting database maintenance from within applications. The same program can be used to process many different files with the same structure. For example, you can specify the filename and disk at runtime; if an output file specified at runtime does not already exist, the system creates it automatically.

The runtime system for Interactive COBOL controls the execution of multiple object programs. You can execute the runtime system in RDOS foreground or background memory or both. Runtime system functions include activating the specified number of terminals, activating a LOGON menu, and interpreting input editing characters.

You can invoke Interactive COBOL utilities through the runtime system or the CLI. Some of the utilities are ANALYZE, which helps you to monitor the growth of existing indexed or relative files; CSSORT, which performs off-line sorting and merging of files; and INQUIRE, which you use to review, update, or create data files.

Utilities for program development are IC/EDIT, for writing source code and documentation; SCREEN, which helps you to design, code, and document display screen formats; the Interactive COBOL Compiler; and the Interactive COBOL Debugger.

Interactive COBOL is also available on MP/OS, AOS, and AOS/VS. Application program and data files can be transported and executed under these operating systems with the Interactive COBOL runtime system; exceptions are code that is dependent on certain operating system calls or error messages.

Interactive COBOL for RDOS is documented in *Interactive COBOL User's Guide (RDOS)* (DGC No. 069-705014), *Interactive COBOL Programmer's Reference* (DGC No. 093-705013), *Interactive COBOL Utilities (RDOS/AOS)* (DGC No. 069-705020), *IC/EDIT: Interactive COBOL Editor* (DGC No. 055-004-01), and *SCREEN: Screen Format Editor* (DGC No. 055-006-01).

The BASIC Interpreters

RDOS supports two versions of BASIC: Extended BASIC, a general-purpose BASIC offering many Data General extensions, and Business BASIC, which is tailored to the needs of commercial data processing.

Both versions of BASIC are *interpretive* languages, that is, interactive languages in which you write, edit, debug and run your programs in the environment of the interpreter. The interpreter checks your code as you enter it, signalling you immediately if it detects syntax errors, misspelled BASIC statements or commands, for instance, or missing or incorrect arguments. When you run your program, the interpreter flags structural errors, such as uninitialized variables, undefined functions, or illegally nested loops.

As interpretive languages, both of these BASIC versions are easy to learn, use and debug. In addition to programming, you can use either version as a desktop calculator to produce immediate results to arithmetic computations, outside the context of program development.

The BASIC System Generation program, BSG, permits you to build a BASIC interpreter tailored to your system and application needs.

Extended BASIC

Data General's Extended BASIC offers many enhancements to the original Dartmouth BASIC, making Extended BASIC suitable for applications ranging from scientific to commercial programming. Data General's extensions include string and matrix operations, user-controlled output formatting, and comprehensive access to data files.

Under RDOS, multiple users can simultaneously access the interpreter, allowing concurrent program development. Multiple user systems allow program swapping, and provide an accounting file capability that records each user's identification, sign-on and resource usage. Extended BASIC combines with Memory Allocation and Protection (MAP) hardware, where available, for dynamic allocation of extended memory for BASIC user programs.

Extended BASIC implements string variables and literals, string linking, and substring manipulation. Through standard statements and functions, you can determine the location of a character within a string, determine the number of characters assigned to a string variable, and convert numeric literals. You can use strings in read or conditional statements.

Matrix operations include statements for matrix manipulation, calculation, and I/O. You can dimension or redimension a matrix easily, and read or write entire matrices in a single I/O call.

Extended BASIC includes trigonometric, logarithmic and user-defined numeric functions. Extended BASIC also performs both single-precision and double-precision arithmetic on real numbers. Single-precision floating-point arithmetic does not require the optional hardware floating-point unit. Double-precision floating point numbers are carried out to 13 to 15 digits. The floating-point storage format is compatible with other Data General software, including FORTRAN. String arithmetic allows operations on string variables and literals up to 18 digits long.

You can call assembly language subroutines from an Extended BASIC program. This provision gives your BASIC programs access to routines that are most efficiently written in assembly language.

Running Extended BASIC under RDOS, you can create, access and delete sequential, random or contiguous files. You can control your output format with runtime options. Runtime options also allow you to chain programs, call subroutines, time your input, suspend execution, and specify error handling routines.

Interactive program development with the interpreter allows you to interrupt a program, print or change values of variables, print or edit program lines, and continue program execution from any line number.

With the MSG command, messages can be transmitted from user to user, user to operator, operator to user, or operator to all users. This command, for example, allows a user to instruct the operator to mount a particular tape, or allows the operator to broadcast a system shutdown message.

RDOS Extended BASIC is syntactically compatible with DOS, AOS and AOS/VS Extended BASIC, with differences occurring in the use of system calls, the CLI, and terminal handling.

Extended BASIC is documented in the *Extended BASIC User's Manual* (DGC No. 093-000065) and *Extended BASIC System Manager's Guide* (DGC No. 093-000119).

Business BASIC

Data General's Business BASIC is designed for commercial data processing, with special arithmetic, data management, and screen formatting facilities suited to business needs. The RDOS Business BASIC interpreter allows multiple users to create, test, debug, or execute programs concurrently and independently.

The Business BASIC extensions that enhance commercial applicability include integer-only arithmetic to prevent rounding errors; multiple-keyed indexed sequential file access method (ISAM); dynamic record allocation; a wide variety of PRINT USING formats; commercial string functions; extended length variable names; direct block I/O; and screen control facilities for cursor positioning and screen format generation.

Business BASIC is packaged with a library containing subroutines and utilities, all of which are written in Business BASIC. The subroutines consist of frequently needed code that you can merge with your programs. They deal primarily with file and index management. The utilities, programs that you can execute or swap to while you are within the Business BASIC environment, simplify program development, file maintenance, and system management.

The utilities packaged with Business BASIC include the CLI emulator, containing many of the commands of the RDOS CLI; the screen maintenance utility (SM), for interactive screen format construction; the EDIT utility for creating and modifying text files; and file sorting utilities. In addition, there are documentation utilities for producing printed text with control of margins, pagination, justified text, headings, and tables of contents; utilities to build, initialize and print indices.

The RDOS system manager can use Business BASIC features to control user access priority and to insulate users from interaction with the operating system. The log-on utility HELLO regulates access priority and system security. In addition, Business BASIC provides a SWAP command/statement, similar to CHAIN, that can protect the Business BASIC monitor from direct user access.

The system manager can send messages to users, interrupt selected jobs, tune system resources, examine and modify specified memory locations, or configure Business BASIC in a run-only system. The system manager can also implement RDOS system calls and assembly language subroutines in Business BASIC programs that other users can then execute.

RDOS Business BASIC also runs under DOS. RDOS Business BASIC is syntactically compatible with AOS Business BASIC, with differences occurring in the use of system calls, the CLI, and terminal handling.

Manuals on Business BASIC include the *Guide to Using Business BASIC (AOS/RDOS/DOS)* (DGC No. 069-000028), *Business BASIC Directory* (DGC No. 093-000226), and *Business BASIC System Management* (DGC No. 093-000228).

Chapter 11

RDOS Utilities

The utilities offered for use under RDOS range from the CLI, with broad functionality, to X.25, a communications software package for network interfacing. Within this range are general utilities for system administration, program development, and data backup, as well as software packages geared for types of applications, such as a database system or sensor I/O.

The RDOS utilities are described in capsule form, with a note on Data General documentation for each. Much of the material in this chapter appears elsewhere in this manual. The utilities are organized into the following categories:

- System Utilities,
- Program Development Utilities,
- Debugging Utilities,
- Backup Utilities,
- Communications Software,
- Miscellaneous Software Packages.

System Utilities

The RDOS system utilities include the Command Line Interpreter, the Batch Monitor, SYSGEN, Sort/Merge and Vertical Format Unit.

Command Line Interpreter (CLI)

The Command Line Interpreter (CLI) is the primary interface between the user and RDOS. Through the CLI you direct system activities and develop and execute your application programs. The CLI functions encompass file and directory management, I/O, system control, foreground-background program management, and running of both the system utilities and the Batch Monitor.

The CLI command syntax provides an easily-learned communications path to the system. The command structure provides mechanisms for repetition, grouping and macro expansion. Many CLI commands approximate system calls in function.

The CLI is documented in *RDOS/DOS Command Line Interpreter*, (DGC No.069-400015).

Batch Monitor

The Batch Monitor permits you to simulate a CLI session by grouping Batch commands, similar to CLI commands, in a file for subsequent processing. Batch accepts job streams from a variety of devices and sends output to specified disk files or devices. When you submit job streams to Batch for processing, the Batch Monitor runs in place of the CLI.

The Batch Monitor is documented in *RDOS/DOS Command Line Interpreter*, (DGC No. 069-400015).

SYSGEN

SYSGEN allows you to tailor RDOS operating systems to your hardware configuration and application needs. In SYSGEN's interactive dialogue, you name parameters for your I/O devices, memory management for mapped and unmapped systems, tuning, disk file structure, automatic restart, real-time clock, core dumping, and user-defined devices. SYSGEN then builds an operating system based on your specifications. You can build new or alternative operating systems as your needs evolve, to use for testing and for different applications.

SYSGEN is documented in *How to Load and Generate RDOS*, (DGC No.069-400013).

DKINIT

The stand-alone disk initialization utility, DKINIT, prepares a disk (or diskette) to run under RDOS, fitting the disk with the framework for the RDOS directory and file structure. You use DKINIT to fully initialize RDOS disks as well as to maintain already-initialized disks. DKINIT performs these functions in full initialization:

- Analyzes the surfaces of the disk for block integrity,
- Sets aside bad blocks in the bad block table to remove them from use,
- Reserves a remap area of blocks to be used in place of bad blocks,
- Writes a disk identification block on the disk, accessible to the operating system.

DKINIT is documented in *How to Load and Generate RDOS*, (DGC No. 069-400013).

Sort/Merge (RDOSSORT)

The Sort/Merge utility, RDOSSORT, facilitates the manipulation of records in data files according to user-defined parameters. Sort/Merge is disk-oriented, though it acts on both disk and magnetic tape files. Running on either a mapped or unmapped system, Sort/Merge allows arrangement of data records by as many as eight keyed fields. With this utility, you can sort records in ascending or descending ASCII sequence, or according to a collating sequence that you define. You can also delete or reformat records, or create output files with subsets of input files. Sort/Merge can produce an address file containing the relative record addresses of an input disk file. It can merge as many as six sorted files at once.

Sort/Merge is documented in *RDOS/DOS Sort/Merge and Vertical Format Unit Utilities*, (DGC No. 069-400021).

Vertical Format Unit (VFU)

The Vertical Format Unit (VFU) utility allows the user to control page formatting for data channel line printers equipped with the Direct Access Vertical Format Unit (DAVFU) option. Through the VFU utility you can create format control files (.VF) that specify tab stops, form length (number of lines per page), and line settings for channels (directives to start or end printing at chosen locations). You use VFU to load the format control file into the printer's memory; the printer format can subsequently be changed by loading another format control file. The VFU utility also allows the user to display and edit a format control file.

The VFU utility is documented in *RDOS/DOS Sort/Merge and Vertical Format Unit Utilities*, (DGC No. 069-400021).

Program Development Utilities

The program development utilities include the Superedit Text Editor and the Text Editor; the Extended Assembler and Macroassembler; and the Relocatable Loader, Library File Editor, and Overlay Loader.

The Extended Assembler, the Macroassembler, the Relocatable Loader, the Library File Editor and the Overlay Loader are documented in *RDOS/DOS Assembly Language and Program Utilities*, (DGC No. 069-400019).

Superedit Text Editor (SPEED)

The Superedit Text Editor, SPEED, is a character-oriented editor combining easy basic commands with powerful features for complex editing needs. SPEED features allow up to 36 edit buffers, multiple file I/O, storage of macros for predefined editing procedures, conditional tasking, and use of numeric variables for repetitive tasks.

SPEED is documented in *RDOS/DOS Superedit Text Editor* (DGC No. 400017).

Text Editor (EDIT)

The Text Editor (EDIT) is a character-oriented editor with easily learned commands. Input is buffered, with implicit line numbering. Output can be formatted by page and sent to multiple files. A macro facility allows predefined command sequences to be executed at will. The multiple-user version of the Text Editor is MEDIT.

EDIT and MEDIT are documented in *RDOS/DOS Text Editor* (DGC No. 069-400016).

Extended Assembler (ASM)

The Extended Assembler (ASM) translates assembly language source code into machine-readable code. Assembler pseudo-ops permit you to direct the Assembler in such operations as conditional assembly or communication between programs. Assembler output includes relocatable binary files, source program listings and error listings.

Macroassembler (MAC)

The Macroassembler (MAC) performs the functions of the Extended Assembler, in addition allowing you to include sections of code called macroinstructions in your program. Macroinstructions comprise general or frequently-needed code that you can call at any place in macroassembly, inserting specific information where needed.

Relocatable Loader (RLDR)

The Relocatable Loader (RLDR) processes assembled or compiled binary files into executable programs, translating relative memory addresses into absolute memory addresses. RLDR loads the appropriate system calls, Task Scheduler, library files, overlays, and debuggers, depending on your specifications. RLDR allows you to allot extra I/O channels and to manipulate starting locations in your program.

Library File Editor (LFE)

The Library File Editor (LFE) helps you to build and maintain library files in order to store relocatable binary files for later use. LFE organizes the binary files in logical records within a library, separating each binary file with special start and end blocks. RLDR can extract binary files from your libraries for loading into an executable program.

LFE allows you to perform the following functions:

- Create a new library from one or more binary files,
- Merge existing libraries into a new library,
- Insert binary files into a new or existing library,
- Replace logical records in a library with new binary files,
- Extract specific logical records from a library,
- Delete logical records from a library,
- Analyze a library, records within a library, or a binary file,
- List titles of records in libraries or binary files.

Overlay Loader (OVLDR)

The Overlay Loader, OVLDR, allows you to replace existing overlays in an overlay file (.OL) with one or more different overlays. For this process you specify to OVLDR the associated save file, the current overlays, and the .RB files that are to replace the current overlays. The CLI command REPLACE uses the OVLDR-produced file to replace individual overlays in the existing .OL file.

Debugging Utilities

The RDOS debugging tools are the Symbolic Debuggers, the Symbolic Editor, the Disk Editor, and the patching utilities ENPAT and PATCH. These utilities are documented in *RDOS/DOS/RTOS Debugging Utilities*, (DGC No. 069-400020).

Symbolic Debuggers (DEBUG and IDEB)

To debug your program, you load a Symbolic Debugger with your program, using RLDR. You can then control program execution, causing the program to halt in the debugger at one or more breakpoints so that you can examine the contents of memory locations and special registers (such as the accumulators), and correct the contents. Because the debugger is symbolic, you can examine memory and register contents in source language format, though you can also use octal format or other special formats.

Data General supplies two versions of the debugger with RDOS. The first, called DEBUG, interrupts during debugging. The second, called IDEB, disables interrupts during debugging. The versions of DEBUG and IDEB are tailored to NOVA or ECLIPSE, mapped or unmapped configurations.

Symbolic Editor (SEDIT)

With the Symbolic Editor, SEDIT, you can edit disk file locations symbolically in random or contiguous files. SEDIT is compatible with the RDOS Symbolic Debuggers and shares many commands with them. SEDIT searches the disk file for symbolic values, and allows you to display and modify locations in octal, instruction, system call, byte, or ASCII format. You can display whole blocks of words, and use special registers to find and modify locations. SEDIT can also access the overlay file of a program from its save file.

Disk Editor (DISKED)

The stand-alone Disk Editor, DISKED, enables you to examine, modify and rearrange disk words. With DISKED you can search for a specific word, fill different parts of the disk with a word, display any part of the disk, or dump part or all of the disk to an output device. DISKED offers seven different output formats and a choice of number bases ranging from 2 to 16.

Patching Utilities (ENPAT and PATCH)

A patch is a one-word change made in a save or overlay file. You use ENPAT to systematically encode and enter data into a patch file, and then PATCH to install the patch file. ENPAT checks syntax on all data input, while PATCH evaluates expressions. ENPAT allows you to encode a multi-word patch, and enables conditional installation of a patch based on the presence of a particular symbol in the loadmap. With ENPAT you can also enter a question for conditional installation that is later output in the PATCH dialogue. Multi-word patches can be marked as a group for conditional installation. PATCH installs the patch file created by ENPAT.

Backup Utilities

The RDOS backup utilities for duplicating disk data on other media include BURST, DDUMP and DLOAD, FDUMP and FLOAD, and OWNER. All of the backup utilities are documented in *RDOS/DOS Backup Utilities* (DGC No. 069-400022).

BURST Programs

The stand-alone BURST utility writes and reads disk images onto and from magnetic tape, and duplicates disk images onto another disk. The BURST programs are much faster than the CLI commands DUMP and LOAD because they bypass the disk file system.

DDUMP and DLOAD

DDUMP and DLOAD allow you to dump and load RDOS/DOS disk directories, or RDOS partitions, onto and from a series of diskettes.

FDUMP and FLOAD

FDUMP and FLOAD fast-copy all files in the current disk directory to magnetic tape, and vice versa. Because FDUMP and FLOAD are designed specifically for magnetic tape, these utilities are faster and use less tape than the CLI commands DUMP and LOAD.

OWNER

The stand-alone utility OWNER allows you to determine the disk block owner of a block lost during a load that used any of the disk-to-disk or disk-to-tape backup utilities.

Communication Software

Communication software includes the Communication Access Manager (CAM), X.25, RJE80 and HASP II.

Communication Access Manager (CAM)

The Communication Access Manager (CAM) provides a series of concise program-level calls for the management of data transfer over DG's Communication System (DG/CS). Both FORTRAN IV and FORTRAN 5 have access to CAM through subroutine calls. CAM is documented in *Communications Access Manager Reference Manual* (DGC No. 093-000183).

X.25

X.25 is Data General's software package for interface with a public packet switching data network, or for networking DG computers. DG's X.25 is compatible with Recommendation X.25 of the CCITT (Comite Consultatif International Telephonique et Telegraphique). Both FORTRAN IV and 5 can use X.25. The X.25 package is documented in *X.25 Protocol User's Manual (RDOS/DOS/RTOS)* (DGC No. 093-000173).

RJE80

The RJE80 software package is Data General's implementation of industry-standard remote job entry (RJE) distributed processing. RJE80 emulates an IBM 2780 or 3780 RJE station and supports the transfer of files between two computers linked by telecommunications lines. For documentation, see the *Remote Job Emulator (RJE80) User's Manual (RDOS)* (DGC No. 093-000164).

HASP II

Data General's HASP II emulates an IBM HASP II remote job entry (RJE) workstation. HASP II is an interactive program that manages the transfer of data between two computers linked by telecommunications lines. HASP II supports up to seven input and seven output streams of data. HASP II is documented in *HASP Workstation Emulator User's Manual (RDOS)* (DGC No. 093-000116).

Miscellaneous Software Packages

RDOS software packages that can be used with assembly language, FORTRAN IV, FORTRAN 5, and DG/L include Array Processor Software, Commercial Subroutine Package, Dataplot, Floating Point Interpreter, International Mathematics and Statistics Library, Sensor Access Manager, and Satellite Processor Software Interface.

Array Processor Software (APS)

The Array Processor Software is designed to simplify programming for the ECLIPSE AP/130 Array Processor and ECLIPSE computers configured with Integral Array Processors. These processors perform high-speed computations on structured data, for example, real or complex arrays. The processors use sophisticated signal and array processing instructions built into the hardware. APS runs on mapped RDOS and gives you access to array functions from either FORTRAN 5 or DG/L programs. APS documentation can be found in the *Array Processor User's Manual* (DGC No. 093-000169).

Commercial Subroutine Package (CSP)

The Commercial Subroutine Package is a library of subroutines that facilitate many commercial programming needs, such as string manipulation, editing, and double word integers. CSP is available to both FORTRAN IV and FORTRAN 5 and is documented in *FORTRAN Commercial Subroutine Package* (DGC No. 093-000107).

Dataplot

Dataplot is a library of subroutines that allow you to interface with an incremental plotter. You can use Dataplot from FORTRAN IV and FORTRAN 5. For documentation, see *Dataplot User's Manual (NOVA)* (DGC No. 093-000060) or *Dataplot User's Manual (ECLIPSE)* (DGC No. 093-000156).

Floating-Point Interpreter

The Floating-Point Interpreter allows the manipulation of floating point numbers on NOVA systems by providing floating point instructions and a set of four accumulators that simulate the action of hardware floating point. The floating point instructions are syntactically similar to machine instructions and are assembled similarly; the accumulators can be addressed and manipulated like actual hardware accumulators. An extended version of the Interpreter, in both binary and relocatable format, is available. See *Floating-Point Interpreter Manual* (DGC No. 093-000019).

International Mathematics and Statistics Library (IMSL)

The International Mathematics and Statistics Library is a comprehensive set of FORTRAN subroutines covering a broad range of mathematical and statistical functions. IMSL is usable with FORTRAN 5. Please contact your local customer service engineer for information on use of IMSL.

Sensor Access Manager (SAM)

The Sensor Access Manager is a library of device handlers and subroutines that manage process I/O. SAM works in conjunction with a chassis and a group of cards collectively called DG/DAC (Data General/Data Acquisition and Control). SAM can be used with FORTRAN IV, FORTRAN 5, or assembly language. See the *Sensor Access Manager (SAM) User's Manual* (DGC No. 093-000225) for more information.

Satellite Processor Software Interface

The Satellite Processor Software Interface allows one or more Data General Satellite Processors (64 Kbyte ECLIPSE computers) to exchange information with a host ECLIPSE computer. Connected as a data channel device to the host's I/O bus, the satellite processor (SP) can support its own terminal, disk, tape drive, communications equipment or Array Processor (AP). SP software allows the host and the satellite processor programs to work in parallel and to transfer data to one another. The host program runs under RDOS, while the satellite processors run under RTOS. The SP Software Interface is usable by FORTRAN 5, DG/L and assembly language. The interface is documented in *Satellite Processor Software Interface User's Manual* (DGC No. 093-000221).

Appendix A

CLI Command Summary

File Management Commands	Function
APPEND groupfilename filename...	Combine two or more files.
BPUNCH filename...	Punch a binary file.
BUILD outputfilename filename...	Build a file of filenames.
CCONT filename blockcount	Create a contiguous file.
CHATR filename [+][−] attribs	Change a file's attributes.
CHLAT filename [+][−] attribs	Change a file's link access attributes.
CLEAR [*filename*]	Set file use count to zero.
CRAND filename	Create a random file.
CREATE filename	Create a sequential file.
DELETE filename	Delete a file.
DUMP dumpfilename [*filename...*]	Dump a file in CLI DUMP format.
ENDLOG [*password*]	Close the LOG file.
FILCOM filename filename2	Compare the contents of two files.
FPRINT filename	Print a file in octal, or other specified format.
LINK linkname resfilename	Create a link entry to resolution filename.
LIST filename	List the statistics of a file.
LOAD dumpfilename [*filename...*]	Load DUMPed files.
LOG [*password*]	Start recording in the log file.

File Management Commands	Function
MKABS savefilename binaryname	Make an absolute file from a save file.
MKSAVE binaryname savefilename	Make a save file from an absolute file.
MOVE directory [*filename...*]	Copy a file to specified directory.
PRINT filename...	Print a file on the line printer.
PUNCH filename...	Punch an ASCII file.
RENAME oldname newname	Rename a file.
REV filename	Display the revision level of program filename.
SAVE filename	Rename a breakfile.
TYPE filename	Type a file on the terminal.
UNLINK linkname	Remove a link entry.
XFER sourcefile destinationfile	Copy the contents of one file to another file.

NOTE: *Some commands appear in several categories.*

Directory Commands	Function
CDIR directoryname	Create an RDOS subdirectory.
CPART partname blockcount	Create a secondary partition of specified number of blocks.
DELETE directoryname	Delete a directory.
DIR directoryname	Change the current directory.
DISK	Display the number of blocks used and remaining on the current partition.
DUMP outputfilename	Copy the contents of the current directory to outputfilename.
EQUIV newname oldname	Temporarily rename a disk or magnetic tape specifier.
FDUMP [MT*n*]	Fast dump the current directory to magnetic tape.
FLOAD [MT*n*]	Fast load a fast dumped (FDUMPed) file into the current directory.
GDIR	Display the current directory name.
INIT directory or tapedrive	Initialize a directory or tape drive.
LDIR	Display the last current directory name.
LOAD dumpfilename	Reload DUMPed files.
LIST [*filename*]	List file information.
MDIR	Display the master directory name.
MOVE directory [*filename...*]	Copy files to directory specified.
RELEASE directory or tapedrive	Release a directory or tape drive.

System Control Commands	Function
BOOT disk or system	Bootstrap a system from disk.
CHAIN filename	Overwrite the CLI with an executable program.
CLEAR [*filename*]	Set file or device use count to zero.
DISK	Display the number of disk blocks used and remaining.
ENDLOG [*password*]	Stop recording in the log file.
EXFG program	Execute program in the foreground.
filename [.SV]	Execute filename.
GMEM	Display background/foreground memory areas (mapped system).
GSYS	Display the current system name.
GTOD	Display the current system time.
FGND	Describe the foreground program status.
INIT directory or tapedrive	Initialize a disk directory or tape drive.
LOG [*password*]	Start recording in the log file.

System Control Commands	Function
MCABOOT	Transmit a system over an MCA line.
MESSAGE text	Display a text message.
POP	Return to the program on the next higher level.
RELEASE directory or tapedrive	Release a directory or tape drive.
SDAY mm dd yy	Set the system calendar.
SMEM background	Set the background/foreground memory areas (mapped systems only).
SPDIS devicename...	Disable spooling to devicename.
SPEBL devicename...	Enable spooling to devicename.
SPKILL devicename...	Delete data spooled to devicename.
STOD [*hh*] [*mm*] [*ss*]	Set the system clock.
TPRINT	Print the tuning file.
TUOFF	Stop recording in the tuning file.
TUON	Start recording in the tuning file.

System Utilities Commands	Function
ALGOL filename...	Compile an ALGOL source file.
ASM filename...	Assemble a source file, producing a relocatable binary (.RB) file.
BASIC	Invoke the BASIC interpreter.
BATCH [*jobfile...*]	Invoke the Batch monitor to execute Batch job stream.
BURST	Dump disk images to magnetic tape or disk.
CLG filename...	Compile, load, and execute a FORTRAN IV source file.
DDUMP	Dump partitions onto diskettes.
DEB savefilename	Debug a program.
DLOAD	Load DDUMPed partitions onto disk.
EDIT [*filename*]	Invoke the Text Editor.
ENPAT [*filename*]	Encode patch files for insertion with PATCH.
FDUMP [MT*n*]	Fast dump the current directory to magnetic tape.
FLOAD [MT*n*]	Fast load a fast-dumped (FDUMPed) file into the current directory.
FORT filename...	Compile a FORTRAN IV source program.
FORTRAN filename...	Compile a FORTRAN 5 source program.
LFE	Create or edit library file of .RB files.
MAC filename...	Assemble a source file into a relocatable binary (.RB) file with the Macroassembler.

System Utilities Commands	Function
MEDIT terminals [*clock-units*]	Invoke the Multiuser Text Editor; *clock-units* control task rescheduling.
NSPEED [*filename*]	EDIT text with the NOVA Supereditor.
OEDIT filename	EDIT disk file locations with the Octal Editor.
OVLDR	Create an overlay replacement file.
OWNER	Determine owner of disk block lost during load.
PATCH	Install patch(es) created by ENPAT.
RDOSSORT	Sort a file or merge files with the RDOS Sort/Merge program.
REPLACE savefilename	Replace overlays in an overlay file.
RLDR	Process relocatable binary files to form an executable program.
SEDIT filename	EDIT disk file locations with the Symbolic Editor.
SPEED [*filename*]	EDIT text with the ECLIPSE Supereditor.
SYSGEN	Generate a new operating system.
VFU filename	Create or load a VFU file for a data channel line printer.

Appendix B

Control Characters

CTRL-character	Function
CTRL-A	Terminate current CLI command, user program or system utility (except a text editor) from foreground or background terminal; return to the higher-level program (usually the CLI).
CTRL-C	Same as CTRL-A, but wait for task I/O to finish and create memory image breakfile BREAK.SV (background program) or FBREAK.SV (foreground program).
CTRL-F	Terminate foreground program from background terminal; release foreground devices and directories.
CTRL-L	Terminate a command line; erase CRT screen.
CTRL-Q	Resume terminal output suspended with CTRL-S.
CTRL-S	Suspend terminal output; resume output with CTRL-Q.
CTRL-Z	Close file entered at terminal with XFER command.

Appendix C

CLI Special Symbols

ASCII Character	Symbol	Meaning
CARRIAGE RETURN	⟨CR⟩	The CARRIAGE RETURN key (ASCII 015)* terminates a command line. The CLI then executes the commands on that line.
FORMFEED		The control character CTRL-L (ASCII 014) terminates a command line, and, on a CRT, erases the screen.
\	\	The backslash key (ASCII 134) deletes an entire line as typed to that point.
SPACE	□	A space (ASCII 040) separates arguments within a command line. Extra spaces have no effect.
COMMA	,	A comma (ASCII 054) separates arguments within a command line. Multiple commas indicate null arguments.
/	/	The slash key (ASCII 057) sets off a switch in a command line.
;	;	The semicolon (ASCII 073) delimits a command to CLI. Several commands separated by semicolons can be entered on one line.
↑	↑	The SHIFT-6 or SHIFT-N keys (ASCII 136), followed immediately by ⟨CR⟩, extends a command over several lines, until terminated with a ⟨CR⟩. A command delimiter (space, comma or semicolon) must precede the SHIFT-6 or SHIFT-N.
#	#	The number sign (ASCII 043) extends a command line over multiple lines in Batch processing.

ASCII Character	Symbol	Meaning
PERIOD	.	The period (ASCII 056) adds the time of day to the CLI R prompt (or removes it).
:	:	The colon (ASCII 072) separates filenames in a directory specifier, used with tape files, disks, and disk files.
*	*	The asterisk (ASCII 052) is a template representing any single character, except a period, in a filename or extension.
-	-	The dash (ASCII 055) is a template representing any number of valid characters, except a period, in a filename or extension.
( )	( )	Left and right parentheses (ASCII 050 and 051) combine commands or arguments in one command line. Multiple commands or arguments are enclosed in parentheses and separated by commas.
⟨ ⟩	⟨ ⟩	Left and right angle brackets (ASCII 074 and 076) expand multiple arguments in a command line.
@	@	The commercial at signs (ASCII 100) enclosing a filename in a command line serve to invoke the file as an indirect file.
QUOTE	"	Quotation marks (ASCII 042) delimit a literal text string, used with the MESSAGE command.
%	%	The percent signs (ASCII 045) enclose a CLI-defined variable.

*All ASCII codes are in octal form.

Appendix D

Batch Monitor Command Summary

Command	Function
!ALGOL	Compiles an ALGOL source file.
!APPEND	Appends one, two or more files to produce a single file.
!ASM	Assembles a program.
!BASIC	Executes a BASIC program.
!BPUNCH	Punches a file or files in binary on the high speed punch.
!CCONT	Creates a contiguous file.
!CHATR	Changes the attributes of an existing file.
!CHLAT	Changes link access attributes.
!COMMENT*	Outputs an operator message.
!CRAND	Adds an entry for a randomly organized file.
!CREATE	Creates a file or series of files.
!CTA*	Assigns a physical cassette unit and defines its name.
!DELETE	Deletes a file.
!DIR	Changes the current default directory.
!DISK	Lists the number of blocks used and the number of blocks still available on the default device.
!DKP*	Assigns a physical moving head disk unit and define its name.
!DUMP	Dumps files. The dump includes directory information for each file that enables later reloading.
!EOF*	Optional job stream terminator.
!EXEC*	Executes a save file.

*Does not have corresponding CLI command.

Command	Function
!FILCOM	Compares two files.
!FORT	Compiles and assembles a FORTRAN IV source file.
!FORTRAN	Compiles and assembles a FORTRAN 5 source file.
!FPRINT	Prints a disk file in byte, decimal, hexadecimal, or octal format.
!GDIR	Prints the name of the current default directory on SYSOUT.
!GMEM	Gets the current memory allocations in a mapped system.
!GSYS	Gets the name of the operating system under which Batch is currently executing.
!GTOD	Prints the current time and date on SYSOUT.
!JOB*	Defines the beginning of a job.
!LFE	Updates a library file.
!LINK	Creates a link.
!LIST	Lists names and other optional information of files in a directory.
!LOAD	Reloads dumped files.
!MAC	Performs a macro assembly.
!MDIR	Gets the name of the master directory.
!MKABS	Makes an absolute file from a save file.
!MKSAVE	Makes a core image file from an absolute binary file.
!MOVE	Moves files from one directory to another.
!MTA*	Assigns a physical magnetic tape unit and defines its name.

Command	Function
!OVLDR	Creates an overlay replacement file.
!PAUSE	Outputs an operator message and pause.
!POP	Returns to the next higher level program in this program environment.
!PRINT	Prints a file or files on SYSOUT.
!PUNCH	Copies ASCII files on the high speed punch.
!RDOSSORT	Invokes RDOS Sort/Merge.
!RELEASE	Prevents further access to a logical device and frees its associated physical device.
!RENAME	Changes the name of a file.
!REPLACE	Performs an overlay replacement.
!REV	Displays the revision level of a save file.
!RLDR	Loads a core image from a series of relocatable binary files.
!SAVE	Renames TMP.SV.
!TPRINT	Prints the tuning file.
!TUOFF	Stops recording in the tuning file.
!TUON	Initiates recording in the tuning file.
!UNLINK	Deletes a link entry.
!XFER	Transfers contents of a file to another file.

Appendix E

System Call Summary

Call	Function
.APPEND	Opens a file for appending.
.BOOT	Bootstraps a new system.
.BREAK	Interrupts the current program; saves the current state of memory in save file format.
.CCONT	Creates a contiguously organized file with all data words zeroed.
.CDIR	Creates a subdirectory.
.CHATR	Changes file attributes.
.CHLAT	Changes link access attributes.
.CHSTS	Gets the status of the file currently open on a specified channel.
.CLOSE	Closes a file.
.CONN	Creates a contiguously organized file with no zeroing of data words.
.CPART	Creates a secondary partition.
.CRAND	Creates a random file.
.CREAT	Creates a sequential file.
.DDIS	Disables user access to a device in a mapped system.
.DEBL	Enables user access to a system (mapped) device.
.DELAY	Delays the execution of a task.
.DELET	Deletes a file.
.DIR	Changes the currrent directory.
.DUCLK	Defines a user clock.
.EOPEN	Opens a file for reading and writing by one user only.
.EQIV	Assigns a temporary name to a device.

Call	Function
.ERDB	Reads one or more disk blocks into extended memory (mapped).
.ERTN	On an error, returns from program and describes error (if to CLI).
.EWRB	Writes one or more 256-word blocks from extended memory to disk (mapped).
.EXBG	Checkpoints a background program (mapped).
.EXEC	Swaps or chains in a new program.
.EXFG	Executes a program in the foreground.
.FGND	Checks whether there is a foreground program running.
.GCHAR	Gets character from the console.
.GCHN	Gets the number of a free channel.
.GCIN	Gets the operator input console name.
.GCOUT	Gets the operator output console name.
.GDAY	Gets today's date.
.GDIR	Gets the current directory name.
.GHRZ	Examines the real-time clock.
.GMCA	Gets the current MCA unit number.
.GPOS	Gets the current file pointer.
.GSYS	Gets the name of the current operating system.
.GTATR	Gets file attributes.
.GTOD	Gets the time of day.
.ICMN	Defines a program communications area.
.IDEF	Identifies a user device.
.INIT	Initializes a device or a directory.

Call	Function
.INTAD	Defines a program interrupt task.
.IRMV	Removes a user device.
.LINK	Creates a link entry.
.MAPDF	Defines a window map (mapped).
.MDIR	Gets the logical name of the master device.
.MEM	Determines available memory.
.MEMI	Changes NMAX.
.MTDIO	Performs free format I/O on tape or cassette.
.MTOPD	Opens a mag tape or cassette for free format I/O.
.ODIS	Disables keyboard interrupts for this console.
.OEBL	Enables keyboard interrupts for this console.
.OPEN	Opens a file for reading and/or writing by one or more users.
.OVLOD	Loads a user overlay into memory.
.OVOPN	Opens a user overlay file.
.OVRP	Replaces an overlay file.
.PCHAR	Writes a character to the console.
.RDB	Reads one or more disk blocks.
.RDCMN	Reads a message from the communications area of the other program.
.RDL	Reads a line.
.RDOPR	Reads an operator message.
.RDR	Reads a random record.
.RDS	Reads sequential bytes.
.RDSW	Reads the console switches.
.RENAM	Renames a file.
.RESET	Closes all files.
.RLSE	Releases a directory or device.

Call	Function
.ROPEN	Opens a file for reading only by one or more users.
.RSTAT	Gets a resolution file's statistics.
.RTN	Returns from a program to a higher-level program.
.RUCLK	Removes a user clock.
.SDAY	Sets today's date.
.SPDA	Disables spooling.
.SPEA	Enables spooling.
.SPKL	Deletes the current spool file.
.SPOS	Sets the current file pointer.
.STAT	Gets a file's statistics.
.STMAP	Sets the data channel map for a user device (mapped).
.STOD	Sets the time of day.
.TUOFF	Turns off the tuning report function.
.ULNK	Deletes a link entry.
.UPDAT	Updates the current file size.
.VMEM	Determines the number of memory blocks.
.WRB	Writes one or more 256-word blocks to disk.
.WRCMN	Writes a message to communications area of the other program.
.WREBL	Removes the write protection of a memory area.
.WRL	Writes a line.
.WROPR	Writes an operator message.
.WRPR	Protects a memory area (mapped).
.WRR	Writes a random record.
.WRS	Writes sequential bytes.

Appendix F

Task Call Summary

Call	Function
.ABORT	Terminates a task immediately.
.AKILL	Kills all tasks of a given priority.
.ARDY	Readies all tasks of a given priority.
.ASUSP	Suspends all tasks of a given priority.
.DQTSK	Dequeues a previously-queued task.
.DRSCH	Disables the rescheduling of the task environment.
.ERSCH	Re-enables the rescheduling of the task environment.
.IDST	Gets a task's status.
.IOPC	Initializes the Operator Communications Package (OPCOM).
.IXMT	Transmits a message from a user interrupt.
.KILAD	Defines a kill-processing address.
.KILL	Kills the calling task.
.LEFD	Disables LEF mode.
.LEFE	Enables LEF mode.
.LEFS	Gets the LEF mode status.
.OVEX	Releases an overlay and returns to the caller.
.OVKIL	Kills an overlayed task and releases the overlay.
.OVREL	Releases an overlay node.
.PRI	Changes the calling task's priority.
.QTSK	Queues a core-resident or overlay task.

Call	Function
.REC	Receives a message from a task.
.REMAP	Triggers the MMPU for a window remap.
.SMSK	Modifies the current interrupt mask.
.SUSP	Suspends the calling task.
.TASK	Initiates a task.
.TIDK	Kills a task by ID number.
.TIDP	Changes the priority of a task by ID number.
.TIDR	Readies a task by ID number.
.TIDS	Suspends a task by ID number.
.TOVLD	Loads a user overlay in a multitask environment.
.TRDOP	Reads an operator message.
.TWROP	Writes an operator message.
.UCEX	Returns from a user clock routine.
.UIEX	Returns from a user interrupt routine.
.UPEX	Returns from a user power fail service routine.
.XMT	Transmits a message to another task.
.XMTW	Transmits a message to another task and waits for its receipt.

Appendix G

Programming Language Examples

Assembly Language

This assembly language source code demonstrates free-format I/O from magnetic tape to disk.

```
; UNBUFFERED I/O EXAMPLE - DISK AND FREE-FORMAT MAG TAPE I/O
; THIS IS THE ASSEMBLER IMPLEMENTATION
; IT WILL COPY THE CONTENTS OF THE DISK FILE "FOO" TO "MT0:0"
;
                    .TITL EXAMPL
                    .NREL
                    .TXTM 1                     ;PACK BYTES LEFT TO RIGHT
BPFOO:              .+1*2                       ;BYTE-POINTER TO "FOO"
                    .TXT /FOO/
BPMTO:              .+1*2                       ;BYTE-POINTER TO "MT0:0"
                    .TXT /MT0:0/
START:              LDA 0,BPFOO
                    SUB 1,1
                    .SYST
                    .OPEN 0                     ;OPEN DISK FILE
                    JSR ERR
                    LDA 0,BPMT0
                    SUB 1,1
                    .SYST
                    .MTOPD 1                    ;OPEN TAPE FOR FREE-FORMAT
                    JSR ERR
                    LDA 0,BUFAD                 ;BUFFER ADDRESS
                    LDA 1,RELBLK                ;START WITH REL. BLK 0
                    SUBZL 2,2
                    MOVS 2,2                    ;1 BLK AT A TIME
LOOP:               .SYST
                    .RDB 0                      ;READ THE FILE
                    JSR ERR
                    LDA 1,CMND1                 ;WRITE 400 WORDS
                    .SYST
                    .MTDIO 1
                    JSR ERR

                    SUBZL 3,3                   ;1 IN AC3
                    LDA 1,EOFFLAG
                    SUB 1,3,SNR                 ;IS EOF FLAG SET?
                    JMP END                     ;YES, GO WRITE EOF'S
                    ISZ RELBLK                  ;NO, GET NEXT BLOCK
                    JMP LOOP-3
ERR:                LDA 3,EOF
                    SUB 2,3,SZR                 ;WAS ERROR EOF?
                    JMP BAD                     ;NO - OR-OR!
                    ISZ EOFFLAG                 ;SET EOF FLAG
                    JMP LOOP+3                  ;WRITE LAST BLOCK
```

```
END:        LDA 1,CMND2          ;WRITE EOF
            .SYST
            .MTOIO 1
            JMP .+1
            LDA 1,CMND2
            .SYST
            .MTDIO 1
            JMP .+1
            .SYST
            .RTN
BAD:        .SYST
            .ERTN
EOF:        6
EOFFLAG:    0
RELBLK:     0
CMND1:      050400
CMND2:      060000
BUFA0:      .+1
            .BLK 400
            .END START
```

FORTRAN IV

This FORTRAN IV source code demonstrates multitasking, with tasks performing line printer and terminal I/O at various intervals.

```
C       MAIN TASK
        CHANTASK 8,8
        EXTERNAL T1,T2,T3,T4,T5
        WRITE (12,5)
5       FORMAT (1H1, "MAIN TASK STARTED",/)
        TYPE "ACTIVATING TASKS"
        CALL ITASK (T1,1,1, IER)
        CALL ITASK (T2,2,2, IER)
        CALL ITASK (T3,3,3 IER)
        CALL ITASK (T5,5,5, IER)
        IF (IER.EQ.1) GO TO 10
        WRITE (12) "MAIN TASK DYING"
        CALL KILL
10      TYPE "ERROR =" IER
        CALL EXIT
        END

C       5 SECOND TASK
        TASK T1
        COMMON/M/ITHREE,IFOUR,IFIVE,IARRAY(10)
5       CALL WAIT (5,2, IER)
        IF (IER.EQ1 ) GO TO 10
        TYPE "ERROR IN TASK 1 = ", IER
        WRITE (12,15)
15      FORMAT (1H0, "5 SECOND TASK")
        GO TO 5
        END

C       15 SECOND TASK
        TASK T2
        COMMON/M/ITHREE,IFOUR,IFIVE,IARRAY(10)
5       CALL WAIT (15,2, IER)
        IF (IER.EQ.1) GO TO 10
        TYPE "ERROR IN TASK 2 = ", IER 12
        WRITE (12) "15 SECOND TASK"
        GO TO 5
        END

C       THIS TASK WILL ACCEPT INPUT FROM $TTI
        TASK T3
        COMMON/M/ITHREE,IFOUR,IFIVE,IARRAY(10)
        ITHREE = 0
        I FOUR = 0
        MESSAGE = 1
5       ACCEPT "ENTER 10 INTEGERS",IARRAY
        CALL XMT (IFOUR, MESSAGE,$10)
        CALL REC (ITHREE,MD)
        GO TO 5
        END
```

```
C       THIS TASK SORTS THE ARRAY
        TASK T4
        COMMON /M/ITHREE,IFOUR,IFIVE,IARRAY(10)
        INTEGER TEMP
        MESSAGE = 1
5       CALL REC(IFOUR, MES)
        DO 6 I=1, 9
        K = K+1
        DO 7 J=K, 10
        TEMP = IARRAY(I)
        IF (IARRAY(I).LE.IARRAY(J)) GO TO 7
        IARRAY(I) =IARRAY(J)
        IARRAY(J) = TEMP
7       CONTINUE
6       CONTINUE
        CALL XMT(IFIVE, MESSAGE,$10)
        GO TO 5
10      TYPE "ERROR IN SORT"
        GO TO 5

C       THIS TASK WILL PRINT OUT THE ARRAY
        TASK T5
        COMMON /M/ITHREE,IFOUR,IFIVE,IARRAY(10)
        MESSAGE = 1
5       CALL REC (IFIVE, MES)
        WRITE (12,100) IARRAY
100     FORMAT (1H0, 10(I7,1,1X))
        CALL XMT(ITHREE, MESSAGE, $10)
        GO TO 5
10      TYPE "ERROR IN PRINTOUT"
        GO TO 5
        END
```

FORTRAN 5

This FORTRAN 5 source code shows how to set up overlays.

```
C       TO SET UP OVERLAYS IN FORTRAN 5, MAKE
C       SURE THAT THE OVERLAY NAME IS DECLARED
C       EXTERNAL IN EVERY PROGRAM UNIT THAT REFERENCES IT,
C       AND THAT AT LEAST ONE .RB FILE FROM EACH OVERLAY
C       DECLARES THE OVERLAY NAME IN AN OVERLAY STATEMENT.
C
        EXTERNAL A1,B2
        TYPE "IN ROOT CODE"
C
C       OPEN A CHANNEL TO THE OVERLAY FILE
C
        CALL OVOPN(0,"MAIN.OL",IER)
        IF(IER.NE.1) STOP "OVOPN ERROR"
C
C       LOAD FIRST OVERLAY
C
        CALL OVLOD(0,A1,-1,IER)
        IF(IER.NE.1) STOP "OVLOD ERROR, A1"
        CALL SUBA1
C
C       LOAD SECOND OVERLAY. IT CAN BE REFERENCED BY OVERLAY
C       NAME B2 OR C3 (PROVIDED THEY'RE DECLARED EXTERNAL).
C
        CALL OVLOD(0,B2,-1,IER)
        IF(IER.NE.1) STOP "OVLOD ERROR, B2"
        CALL SUBB2
        CALL SUBC3
        TYPE "BACK IN ROOT"
        STOP
        END

C       THESE ARE THE SUBROUTINES

        OVERLAY A1
        SUBROUTINE SUBA1
        TYPE "IN SUBROUTINE A1"
        RETURN
        END

        OVERLAY B2
        SUBROUTINE SUBB2
        TYPE "IN SUBROUTINE B2"
        RETURN
        END

        OVERLAY C3
        SUBROUTINE SUBC3
        TYPE "IN SUBROUTINE C3"
        RETURN
        END
```

DG/L

This fragment of DG/L source code demonstrates the initial declarations section, which specifies variables and procedures for the program. Declarations are characteristic of DG/L as a structured programming language.

```
1   /* This is a program to calculate relativistic effects for
2   various velocities of your spacecraft. This program requires
3   no external procedures other than built-in DG/L runtimes for
4   communication with the terminal.*/
5
6   BEGIN
7
8   /* You specify global declarations, valid over the entire program,
9   as the first part of this outermost block.*/
10
11  /* You can substitute C for this value throughout the program
12  by declaring this literal.*/
13
14  LITERAL C (299792.458);
15
16  /* You set up the real variables you need for calculation.*/
17
18  REAL (4) VELOCITY, FRACTION_OF_C, LORENTZ_FITZGERALD_CONTRACTION, TIME,
19  LENGTH, MASS;
20
21  /* You use this string to store replies to queries from the
22  program.*/
23
24  STRING YES;
25
26  /* You declare an internal procedure to carry out calculations.*/
27
28  REAL (4) PROCEDURE RELATIVITY;
29
30  /* The procedure contains a block with its own declaration.*/
31
32  BEGIN
33
34
35  /* You declare a Boolean to control loop termination.*/
36
37  BOOLEAN DONE;
38
39  /* This ends the declaration part of the block and begins
40  the executable statements of the block. The DO statement is
41  controlled by a trailing UNTIL, and contains a compound
42  statement.*/
```

Interactive COBOL

This fragment of Interactive COBOL source code illustrates a screen section in a typical COBOL program.

```
1   SCREEN SECTION.
2
3   01 MASTER-MENU.
4     03 BLANK SCREEN.
5     03 LINE 1 COL 20 VALUE
6     "REGIONAL HARDWARE DISTRIBUTION CORPORATION".
7     03 LINE 3 COL 20 VALUE
8     "**********************************************".
9     03 LINE 7 COL 26 VALUE
10    "CUSTOMER RECORDS MAINTENANCE".
11    03 LINE 12 COL 26 VALUE
12    "The functions available are:".
13    03 LINE PLUS 2 COL 30 VALUE "1) Add a record".
14    03 LINE PLUS 1 COL 30 VALUE "2) Change a record".
15    03 LINE PLUS 1 COL 30 VALUE "3) Examine a record".
16    03 LINE PLUS 1 COL 30 VALUE "4) Delete a record".
17    03 LINE PLUS 2 COL 25 VALUE
18    "Please type function number".
19    03 COL PLUS 3 PIC 9 TO CHOICE.
20
21  01 FILENAME-SCREEN.
22    03 BLANK SCREEN.
23    03 LINE 1 COL 20 VALUE
24    "REGIONAL HARDWARE DISTRIBUTION CORPORATION".
25    03 LINE 3 COL 20 VALUE
26    "**********************************************".
27    03 LINE 14 COL 26 "What is the name of your file?".
28    03 LINE PLUS 2 COL 36 PIC X(10) TO I-FILE.
29
30  01 SECURITY-SCREEN SECURE.
31    03 BLANK SCREEN.
32    03 LINE 1 COL 20 VALUE
33    "REGIONAL HARDWARE DISTRIBUTION CORPORATION".
34    03 LINE 3 COL 20 VALUE
35    "**********************************************".
36    03 LINE 12 COL 23 VALUE "Please enter your authorization code".
37    03 LINE PLUS 3 COL 39 PIC X TO I-RANK AUTO.
38    03 PIC XX TO I-AREA.
39
40  01 BADCODE-SCREEN.
41    03 LINE 22 BLANK LINE BELL VALUE
42    "INCORRECT CODE".
43    03 LINE 24 BLANK LINE VALUE
44    "Hit 'ESC' to stop -- Hit 'CR' to try again".
45    03 COL PLUS 2 PIC X TO CHAR.
46
47  01 NOT-AUTHORIZED-SCREEN.
48    03 LINE 22 BLANK LINE BELL VALUE
49    "YOU ARE NOT AUTHORIZED TO PERFORM THAT FUNCTION".
50    03 LINE 24 BLANK LINE VALUE
51    "Hit 'ESC' to stop Hit 'CR' to try again".
52    03 COL PLUS 2 PIC X TO CHAR.
53
54  01 BAD-CHOICE.
55    03 LINE 22 BLANK LINE BELL VALUE
56    "CHOICE MUST BE A NUMBER FROM 1 TO 4".
57    03 LINE 24 BLANK LINE VALUE
58    "Hit 'ESC' to stop -- Hit 'CR' to try again".
59    03 COL PLUS 2 PIC X TO CHAR.
```

Extended BASIC

This example of Extended BASIC code demonstrates I/O and calculation.

```
0010  REM PROGRAM MORTGAGE.BA, COMPUTES MORTGAGE PAYMENTS, HAS TAX SUBROUTINE.
0020  PRINT "(12) I CALCULATE MORTGAGE PAYMENTS, INTEREST, AND TAXES."
0030  PRINT "TYPE AMOUNT OF PRINCIPAL, INTEREST RATE IN WHOLE NUMBERS,"
0040  PRINT "MORTGAGE LIFE IN YEARS, AND ANNUAL PROPERTY TAX BILL FOR HOUSE."
0050  PRINT "SEPARATE ENTRIES WITH A COMMA; FOR EXAMPLE 40000,10.5,25,2000."
0060  PRINT
0070  PRINT " AMOUNT? RATE? YEARS? TAXES?"
0080  INPUT " ? ",A,R1,Y,T
0090  REM GET MONTHLY RATE R (R1/12), TURN TO FRACTION AS R1 WAS WHOLE NUMBER.
0100  LET R=R1/1200
0110  REM GET NUMBER OF MONTHS M FOR LOAN.
0120  LET M=12*Y
0130  REM COMPUTE MONTHLY PAYMENT.
0140  LET P=A*R*(1+R)  M/((1+R)  M-1)
0150  REM DEFINE FORMAT F$ THAT ROUNDS NUMBERS TO NEAREST WHOLE CENT.
0160  LET F$="-------.##"
0180  REM PRINT TOTALS AND GIVE OPTION FOR TAX SUBROUTINE.
0190  PRINT "MONTHLY PAYMENT:  TAXES:  HIDEOUS TOTAL:"
0200  PRINT USING F$,P,"",T/12," ",P+T/12
0210  PRINT
0220  PRINT "WANT TO COMPUTE THE TRUE COST AFTER U.S.TAX DEDUCTIONS ON THE"
0230  PRINT "INTEREST AND TAXES? YOU MUST ITEMIZE TO QUALIFY."
0240  INPUT "ANSWER Y (YES) OR N (NO).  ",Q$
0250  REM $ SPECIFIES STRING INPUT (E.G."Y") INSTEAD OF NUMERIC.
0260  IF Q$="Y" THEN GOSUB 1000
0270  PRINT
0280  INPUT "TYPE Y (YES) TO RUN PROGRAM AGAIN, ANYTHING ELSE TO STOP. ",Q$
0290  IF Q$="Y" THEN GOTO 0060
0300  STOP
1000  REM TAX DEDUCTION COMPUTATION SUBROUTINE.
1010  INPUT "WHAT IS YOUR TAX BRACKET, IN WHOLE NUMBERS?  ",B1
1020  LET B=B1/100
1030  PRINT
1040  PRINT "SHOULD I LIST PAYMENTS FOR THE FIRST TWO YEARS? I HAVE"
1050  INPUT "TO FIGURE THE INTEREST ANYWAY. ANSWER Y (YES) OR N (NO).  ",Q$
1060  REM SET UP VARIABLES A1 (PRINCIPAL PD PER MO.) AND I1 (FOR TOTAL INTEREST)."
1070  LET A1=A
1080  LET I1=0
1090  IF Q$<>"Y" THEN GOTO 1110
1100  PRINT "  MONTH  PRIN.  INT. INT. TOTAL"
1110  REM FOR-NEXT LOOP COMPUTES (OPTIONALLY LISTS) FIGURES BY MOS. & TOTALS.
1120  FOR J=1 TO 24
1130  LET P1=A1*R/((R+1)  M-1)
1140  LET I1=I1+(P-P1)
1150  LET A1=A1-P1
1155  LET M=M-1
1160  IF Q$<>"Y" THEN GOTO 1180
1170  PRINT USING F$,J,P1,P-P1,I1
1180  NEXT J
1190  PRINT
```

```
1200  REM GET DEDUCTIONS D FOR 1 YEAR, T(AXES) + I1/2 (HALF OF 2 YRS. INTEREST).
1210  LET D=T+I1/2
1220  PRINT "ANNUAL MORTGAGE-RELATED DEDUCTIONS ARE:"
1230  PRINT USING F$,D
1240  PRINT "BUT I MUST SUBTRACT THE $3200 STANDARD (0 BRACKET) DEDUCTION"
1250  PRINT "BUILT INTO THE TAX TABLES. THE TRUE MONTHLY COST IS:"
1260  LET D1=D-3200
1270  REM GET REAL MO. COST. (TOTAL MO. PAY = P+T12) - ((BRKT * ADJ. DEDS)/12).
1280  LET C=(P+T/12)-(B*D1/12)
1290  PRINT USING F$,C
1300  PRINT     "****SUMMARY****"
1310  PRINT "  LIFE:  AMOUNT:  RATE:  CASH PAY:  BRKT:  TRUE COST:"
1320  PRINT USING F$,Y,A,R1,P+T/12,B1,C
1330  RETURN
```

Business BASIC

This Business BASIC example illustrates I/O and database management.

```
: THE FOLLOWING PROGRAM PERFORMS A DATABASE BUILD FROM THE
: TERMINAL. NOTE THAT THIS PROGRAM IS ASSUMED TO BE RUNNING
: IN A SINGLE-USER ENVIRONMENT. SO, NO LOCKING OF THE
: DATA FILE OR INDEX FILE IS PERFORMED.
:
0030 ENTER "POSFL.SL"
0050 ENTER "GETREC.SL"
0070 DIM REC$[50],C1[1,3],PCODE$[12],DESC$[21],X$[512]
0075 DIM B$[544],KEY$[8],D$[18],INAME$[10]
:
: SETUP STRING TO HOLD NAME OF INDEX AND ALSO
: SET UP STRING FOR OPEN PROGRAM
:
0170 LET INAME$="INVINX.AK",FILL$(0)
0190 LET X$="INVMST.AK,5,INVINX.AK,5",FILL$(0)
:
: WRITE STRING TO COMMON AND OPEN FILES
:
0210 BLOCK WRITE X$
0230 SWAP "OPEN"
0250 BLOCK READ X$
:
: CONVERT STRING RETURNED FROM OPEN TO C1 ARRAY
:
0270 LET K=1
0275 FOR I=0 TO 1
0290 FOR J=0 TO 3
0310 LET C1[I,J]=ASC(X$[K,K+3])
0330 LET K=K+4
0350 NEXT J
0355 NEXT I
:
: VERIFY THAT OPEN WAS SUCCESSFUL
:
0370 IF C1[0,0])=0 THEN GOTO 0415
0390 PRINT "OPEN ERROR"
0410 STOP
:
: SET LOCK FLAG TO 1 TO INDICATE NO-LOCKING OF INDEX
: (THIS MUST BE SET TO 0 IN A MULTI-USER ENVIRONMENT)
:
0415 LET LFLAG=1
:
: SET UP D$ STRING FOR KEY ROUTINES. NOTE THAT LOGICAL FILE 1
: IS BEING DESCRIBED BY THIS STRING.
:
0416 LET D$=CHR$(C1[1,0],2),CHR$(C1[1,1]4),CHR$(LFLAG,2),INAME$,FILL$
:
: INPUT DATA FROM TERMINAL
:
0430 PRINT
0450 INPUT "PRODUCT ID CODE:",PCODE$," PRODUCT DESCRIPTION:",DESC$
0470 INPUT "UNIT COST:",COST," AVERAGE COST:",AVCST;
0490 INPUT " UNIT SALES YTD:",YTDS%
0510 INPUT "DATE OF LAST SALE:",DATLS," DATE OF LAST PURCHASE:",DATLP
0530 INPUT "QUANTITY ON HAND:",QTYOH%,"MINIMUM QUANTITY ON HAND:",MINQ
0550 INPUT "QUANTITY RECEIVED YTD:",RECYTD%
```

```
:
: NOW, SET UP RECORD AND KEY STRINGS
:
0590 LET REC$=CHR$(1,2)
0610 LET REC$[3]=CRM$(PCODE$),FILL$(0)
0615 LET KEY$=CRM$(PCODE$),FILL$(0)
0630 LET REC$[11]=CRM$(DESC$),FILL$(0)
0650 LET REC$[25]=CHR$(COST,4),CHR$(AVCST,4),CHR$(YTDS%,2),CHR$(DATLS),
0670 LET REC$[0]=CHR$(DATLP,3),CHR$(QTYOH%,2),CHR$(MINQTY%,2)
0690 LET REC$[0]=CHR$(RECYTD%,2),FILL$(0)
:
: NOW, FIND AVAILABLE RECORD VIA GETREC (GOSUB 8400). NOTE THAT
: LOGICAL FILE 0 IS BEING REFERENCED.
:
0730 LET F%=0
0750 GOSUB 8400
0770 IF R1()-1 THEN GOTO 0830
0790 PRINT "ERROR ON GETREC"
0810 STOP
:
: GETREC SUCCESSFUL, SO POSFL TO THAT RECORD (GOSUB 9610)
: AND WRITE THE RECORD
:
0830 GOSUB 9610
0850 WRITE FILE[C%],REC$
:
: NOW, ADD KEY TO INDEX
:
0855 LET RECNO=R1
0856 KADD D$,B$,KEY$,RECNO
0857 IF RECNO()0 THEN GOTO 0870 : *****SEE IF MORE DATA TO INPUT*****
0858 PRINT "KADD ERROR"
0859 STOP
:
: NOW, SEE IF USER HAS MORE DATA TO INPUT
:
0890 INPUT "MORE DATA (0=YES,1=NO)?",ANS
0910 IF ANS=0 THEN GOTO 0430
0920 CLOSE
0930 STOP
```

Index

B

C

D

E

F

G

N

O

T

U

V

W

X

Z

Data General

Data General Corporation, Westboro, Massachusetts 01580

069-400011-00