

User's Guide

ACUCOBOL-GT[®]

Version 8.1.3

Micro Focus

9920 Pacific Heights Blvd
San Diego, CA 92121
858.795.1900

© Copyright Micro Focs (IP) LTD. 1998-2010. All rights reserved.

Acucorp, ACUCOBOL-GT, Acu4GL, AcuBench, AcuConnect, AcuServer, AcuSQL, AcuXDBC, **extend**, and “The new face of COBOL” are registered trademarks or registered service marks of Micro Focus. “COBOL Virtual Machine” is a trademark of Micro Focus. Acu4GL is protected by U.S. patent 5,640,550, and AcuXDBC is protected by U.S. patent 5,826,076.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries. UNIX is a registered trademark of the Open Group in the United States and other countries. Solaris is a trademark of Sun Microsystems, Inc., in the United States and other countries. Other brand and product names are trademarks or registered trademarks of their respective holders.

E-01-UG-100501-ACUCOBOL-GT-8.1.3

Contents

Chapter 1: Introduction

1.1 ACUCOBOL-GT Documentation	1-2
1.2 Product Overview	1-2
1.2.1 Portability and Compatibility	1-4
1.2.2 Native Instructions.....	1-4
1.2.3 The ACUCOBOL-GT Runtime.....	1-5
1.2.3.1 Windows console runtime	1-6
1.2.4 Runtime Configuration	1-7
1.2.5 Graphical Technology	1-8
1.2.6 File System Flexibility.....	1-9
1.2.7 Complementary Technologies	1-10
1.3 Document Overview	1-12
1.3.1 User's Guide	1-12
1.3.2 User Interface Programming.....	1-13
1.3.3 Reference Manual	1-14
1.3.4 Appendices	1-15
1.3.5 Getting Started	1-16
1.3.6 Transitioning to ACUCOBOL-GT	1-16
1.3.7 A Guide to Interoperating with ACUCOBOL-GT	1-16
1.3.8 A Programmer's Guide to the Internet	1-17
1.3.9 Related Documents	1-17
1.4 Supported Hardware	1-17
1.4.1 Native Code Supported Processors	1-18
1.5 Environment Variables	1-18
1.6 Notation	1-20
1.7 How to Get Help	1-20
1.7.1 Handling Compilation Problems	1-21
1.7.2 Handling Program Execution Problems	1-22

Chapter 2: Compiler and Runtime

2.1 Introduction.....	2-2
2.2 Using the Compiler	2-3
2.2.1 Standard Options	2-4
2.2.2 Native Code Options.....	2-5
2.2.3 Listing Options	2-9

2.2.4	Internal Tables Options.....	2-13
2.2.5	Compatibility Options.....	2-15
2.2.6	Interoperability Options.....	2-21
2.2.7	File Options.....	2-23
2.2.8	Source Options.....	2-30
2.2.9	Reserved Word Options.....	2-32
2.2.10	Data Storage Options.....	2-34
2.2.10.1	Truncation Options.....	2-46
2.2.11	Video Options.....	2-48
2.2.12	Warning and Error Options.....	2-51
2.2.13	Debugging Options.....	2-52
2.2.14	Mapping Options.....	2-54
2.2.15	Conditional Compilation Options.....	2-57
2.2.16	Miscellaneous Options.....	2-58
2.2.17	Upper and Lower Case.....	2-68
2.2.18	File Name Handling.....	2-68
2.2.18.1	Remote file name handling.....	2-69
2.2.19	Compiler Command-Line Examples.....	2-70
2.2.20	CBLFLAGS Environment Variable.....	2-71
2.2.21	Help, Version Information, and Communication With C Programs.....	2-72
2.2.22	The “>>IMP” Directive.....	2-73
2.3	Using the Runtime System.....	2-75
2.3.1	Runtime Options.....	2-77
2.4	Compatibility Modes.....	2-92
2.5	Source Formats.....	2-93
2.6	COPY Libraries.....	2-95
2.6.1	Resource Files.....	2-97
2.6.1.1	General Rules for Resources.....	2-98
2.7	Source Code Control.....	2-99
2.8	Runtime Configuration.....	2-100
2.8.1	File Name Assignments.....	2-101
2.8.2	Code and Data File Search Paths.....	2-103
2.8.3	File Status Codes.....	2-104
2.8.4	Terminal Handling Options.....	2-105
2.8.5	File Handling Options.....	2-105
2.8.5.1	Sort files.....	2-106
2.8.5.2	Carriage control.....	2-106
2.8.5.3	Device locking.....	2-107
2.9	File Name Interpretation.....	2-107
2.9.1	File Names Starting With a Hyphen.....	2-110

2.9.2 File Name Examples.....	2-111
2.9.2.1 Example 1: Default name handling.....	2-111
2.9.2.2 Example 2: Accessing printers.....	2-113
2.9.3 Assigning Files to Local Printers.....	2-114
2.10 Calling Subprograms.....	2-115
2.10.1 CALL.....	2-115
2.10.2 CANCEL.....	2-118
2.10.3 CHAIN.....	2-119
2.10.4 Alternate ENTRY Points.....	2-119
2.11 Reducing the Size of the Runtime.....	2-120
2.12 acushare Utility Program.....	2-121
2.12.1 Using Shared Memory.....	2-122
2.12.1.1 Indicating programs to share.....	2-123
2.12.2 Using acushare.....	2-124
2.12.2.1 acushare -start.....	2-124
2.12.2.2 acushare -kill.....	2-125
2.12.2.3 acushare -clean.....	2-125
2.12.2.4 acushare -version.....	2-126
2.12.2.5 acushare (with no options).....	2-126
2.12.3 acushare errors.....	2-127
2.13 General Preprocessor Interface.....	2-129
2.13.1 Use of Preprocessors.....	2-130
2.13.1.1 Calling a preprocessor.....	2-130
2.13.1.2 Calling two or more preprocessors.....	2-132
2.13.1.3 Compiler options forwarded to preprocessors.....	2-133
2.13.1.4 Calling a preprocessor without the compiler.....	2-134
2.13.2 AcuSQL Pre-compiler.....	2-135
2.13.2.1 Compatibility with ACUCOBOL-GT general preprocessor interface.....	2-135
2.13.2.2 Calling the AcuSQL pre-compiler.....	2-136
2.13.3 Writing a Preprocessor.....	2-136
2.13.3.1 Command-line options.....	2-136
2.13.3.2 Line and file directives.....	2-138
2.13.3.3 Error messages.....	2-141

Chapter 3: Debugger and Utilities

3.1 Runtime Debugger.....	3-2
3.1.1 Entering the Debugger.....	3-5
3.1.2 Cursor and Mouse Handling in Source-level Debugging.....	3-7
3.1.3 Debugger Commands.....	3-8
3.1.3.1 Source-level commands.....	3-10
3.1.3.2 Other commands.....	3-11

3.1.3.3	Multithreading Issues.....	3-15
3.1.3.4	Getting help.....	3-16
3.1.3.5	File menu.....	3-17
3.1.3.6	View menu.....	3-20
3.1.3.7	Run menu.....	3-22
3.1.3.8	Source menu.....	3-25
3.1.3.9	Data menu.....	3-28
3.1.3.10	Breakpoints menu.....	3-35
3.1.3.11	Selection menu.....	3-41
3.1.3.12	Help menu.....	3-45
3.1.4	File Tracing.....	3-47
3.1.5	Screen Tracing.....	3-49
3.1.6	Macro Debugger.....	3-50
3.1.7	Specifying Addresses.....	3-51
3.1.7.1	Variables.....	3-51
3.1.7.2	Program addresses.....	3-53
3.1.8	Debugger Restrictions.....	3-53
3.1.9	Using the Abend Diagnostic Report (ADR).....	3-54
3.1.9.1	Generating a report.....	3-56
3.1.9.2	ADR restrictions.....	3-58
3.2	Object File Utility — cblutil.....	3-58
3.2.1	Object Libraries.....	3-59
3.2.2	Creating Object Libraries.....	3-60
3.2.2.1	Creating remote object libraries.....	3-62
3.2.3	Getting Object Information.....	3-63
3.2.4	Generating Native Code.....	3-64
3.3	Vision File Utility — vutil.....	3-66
3.3.1	Examining File Information.....	3-67
3.3.2	Testing File Integrity.....	3-69
3.3.3	Rebuilding Files.....	3-71
3.3.4	Resetting User Counts.....	3-78
3.3.5	Resetting Internal Revision Number.....	3-78
3.3.6	Extracting Records From a File.....	3-79
3.3.7	Recovering Deleted Records.....	3-80
3.3.8	Creating Empty Files.....	3-81
3.3.8.1	Responding to vutil generated prompts.....	3-82
3.3.8.2	Specifying file attributes in advance.....	3-84
3.3.9	Unloading to Binary and Line Sequential Format.....	3-87
3.3.10	Loading a File.....	3-89
3.3.11	File Size Summary Report.....	3-92
3.3.12	Converting RM/COBOL-85 Indexed Files.....	3-92

3.3.13	Converting C-ISAM Files	3-92
3.3.14	Converting Micro Focus Files	3-94
3.3.15	Changing Record Size	3-96
3.3.16	Setting the Comment Field	3-97
3.3.17	Miscellaneous Commands	3-97
3.3.18	Default Settings of vutil	3-98
3.4	File Transfer Utility — vio	3-98
3.4.1	vio Options	3-100
3.4.2	Windows Considerations	3-104
3.4.3	vio Examples	3-105
3.4.4	Known Limitations	3-106
3.5	Indexed File Record Editor (alfred)	3-107
3.6	logutil	3-107
3.6.1	Syntax and Options	3-107
3.6.2	logutil Report Headings	3-110
3.7	The Profiler	3-111
3.7.1	Using the Profiler	3-112
3.7.2	Configuring the Profiling Tools	3-113
3.7.3	Understanding the Report	3-115
3.7.4	Understanding the XML Data File	3-117
3.8	External Sort Utility — AcuSort	3-121
3.8.1	AcuSort Command Format	3-121
3.8.2	AcuSort Instructions	3-122
3.8.2.1	CHAR-ASCII and SIGN-ASCII	3-122
3.8.2.2	CHAR-EBCDIC and SIGN-EBCDIC instructions	3-123
3.8.2.3	SORT/MERGE instructions	3-123
3.8.2.4	USE/GIVE instructions	3-125
3.8.2.5	INCLUDE/OMIT instructions	3-128
3.8.3	Code Sample	3-131
3.8.4	AcuSort Environment Variables	3-133
3.9	Remote Preprocessing Utility — Boomerang	3-135
3.9.1	License Requirements and Installation	3-135
3.9.2	Server Setup and Configuration	3-136
3.9.2.1	Step 1: Creating an Alias File	3-136
	Pro*COBOL Alias Example	3-140
	CICS Alias Example	3-141
	UniKix Alias Example	3-142
	DB2 Alias Example	3-143
3.9.2.2	Step 2: Creating a Configuration File	3-145
3.9.2.3	Step 3: Creating an Access File	3-145
3.9.2.4	Step 4: Starting the Server	3-146

3.9.3 Server commands.....	3-146
3.9.4 Client-side Operation – Remote Precompiling.....	3-147
3.9.5 Client Commands.....	3-148
3.9.6 Working with INCLUDE files.....	3-150

Chapter 4: Terminal Manager

4.1 How the Terminal Manager Works.....	4-2
4.1.1 Terminal Manager Functions.....	4-3
4.1.2 Alternate Terminal Manager (ATM).....	4-4
4.2 Getting Your Terminals Ready.....	4-5
4.2.1 Step One: Terminal Identification.....	4-5
4.2.2 Step Two: Terminal Definition.....	4-7
4.2.2.1 Windows special considerations.....	4-7
4.2.3 Step Three: Configuration Variables.....	4-8
4.3 The Keyboard Interface.....	4-9
4.3.1 Key Mapping.....	4-10
4.3.1.1 Key interpretation.....	4-10
4.3.1.2 Key translation.....	4-11
4.3.1.3 Keyboard configuration.....	4-12
4.3.2 Redefining the Keyboard.....	4-16
4.3.2.1 The KEYBOARD variable.....	4-16
4.3.2.2 The KEYSTROKE variable.....	4-19
4.3.2.3 Table of keys.....	4-31
4.3.2.4 Additional Windows keys.....	4-35
4.3.2.5 Special keys.....	4-37
4.3.2.6 Default keyboard.....	4-39
4.3.2.7 Modification examples.....	4-41
4.4 The Display Interface.....	4-42
4.4.1 Adding Color.....	4-43
4.4.2 The SCREEN Option.....	4-45
4.4.2.1 SCREEN examples.....	4-57
4.4.3 Additional Configuration Variables.....	4-57
4.4.4 Double-Byte Character Handling.....	4-60
4.5 Restricted Attribute Handling.....	4-61
4.5.1 Restricted Video Modes.....	4-62
4.5.1.1 Restrictions.....	4-64
4.6 The Terminal Database File.....	4-65
4.6.1 Required Functions.....	4-70
4.6.2 Additional Screen Functions.....	4-71
4.6.3 Video Attributes.....	4-73

4.6.4 Color	4-74
4.6.4.1 One-color terminals	4-75
4.6.5 Function Keys and Other Keys	4-75
4.6.5.1 User-defined keys	4-76
4.6.6 Line Drawing	4-76
4.6.6.1 Multi-character sequences for graphics	4-77
4.6.7 Graphical Window and Control Emulation	4-78
4.6.8 Mouse Support for X Terminals	4-82
4.6.9 Initialization	4-83
4.6.10 Print Functions	4-83
4.6.11 Continued Entries	4-84

Chapter 5: File Processing

5.1 Transaction Management	5-2
5.1.1 Overview of Transaction Management	5-2
5.1.1.1 Transaction logging	5-3
5.1.1.2 File types	5-4
5.1.1.3 Features	5-4
5.1.2 The Transaction Logging Process	5-4
5.1.3 Transaction Management Verbs	5-6
5.1.4 Extended Locking Rules	5-7
5.1.4.1 Special handling of implicit transactions	5-9
5.1.5 Logging and Rollback of File Update Operations	5-9
5.1.6 Multiple Log Files	5-10
5.1.7 Configuration Variables	5-11
5.1.8 Transaction Error Handling	5-11
5.1.9 Compiler File Options	5-13
5.1.10 Recovery	5-14
5.1.10.1 Transaction logging and recovery with AcuServer	5-15
5.2 AcuServer	5-16
5.2.1 System Requirements	5-17
5.2.2 Remote Name Notation	5-18
5.3 XFD Files	5-19
5.3.1 Defaults Used in XFD Files	5-23
5.3.1.1 KEY IS phrase	5-23
5.3.1.2 RENAMES clause	5-23
5.3.1.3 REDEFINES clause	5-24
5.3.1.4 Multiple record definitions	5-24
5.3.1.5 Group items	5-25
5.3.1.6 FILLER data items	5-25

5.3.1.7 OCCURS clauses.....	5-25
5.3.1.8 Summary of dictionary fields	5-26
5.3.1.9 Identical field names.....	5-26
5.3.1.10 Long field names	5-27
5.3.1.11 Naming the XFD.....	5-27
5.3.1.12 Examples of XFD names	5-28
5.3.2 Using Directives	5-29
5.3.2.1 Important for Acu4GL and AcuXML sites	5-30
5.3.3 Directive Syntax	5-30
5.3.3.1 ALPHA directive.....	5-31
5.3.3.2 BINARY directive	5-32
5.3.3.3 COBOL-TRIGGER directive	5-33
5.3.3.4 COMMENT directive.....	5-35
5.3.3.5 DATE directive.....	5-35
5.3.3.6 FILE directive.....	5-39
5.3.3.7 NAME directive.....	5-40
5.3.3.8 NUMERIC directive.....	5-43
5.3.3.9 SECONDARY-TABLE directive.....	5-44
5.3.3.10 SUBTABLE directive (AcuXDBC use only).....	5-45
5.3.3.11 USE GROUP directive	5-46
5.3.3.12 VAR-LENGTH directive.....	5-47
5.3.3.13 WHEN directive	5-48
5.3.3.14 XSL directive.....	5-54
5.3.4 XFD Format.....	5-54
5.3.4.1 Identification section	5-55
5.3.4.2 Key section	5-56
5.3.4.3 Condition section.....	5-58
5.3.4.4 Field section.....	5-59
5.4 International Character Handling.....	5-62
5.4.1 Files Required for Translation	5-64

Chapter 6: Programmer's Guide

6.1 Handling Files.....	6-2
6.1.1 Sequential Files.....	6-2
6.1.2 Relative Files	6-4
6.1.3 Indexed Files - Vision.....	6-5
6.1.3.1 Segment naming of Vision 4 and 5 files.....	6-7
6.1.3.2 Method one: The format method	6-8
6.1.3.3 Method two: The default method	6-9
6.1.3.4 Overriding individual segment names.....	6-10
6.1.3.5 Selecting the Vision version	6-10
6.1.3.6 Keys.....	6-11

6.1.3.7 Other Vision features.....	6-13
6.1.4 Record Locking	6-15
6.1.5 Device Locking Under UNIX.....	6-16
6.1.6 Indexed File Considerations	6-17
6.1.6.1 Compression	6-17
6.1.6.2 Mass update	6-19
6.1.6.3 Bulk addition mode for Vision.....	6-20
6.1.7 Performance Considerations.....	6-32
6.1.8 Limits on Open Files	6-33
6.2 Terminal I/O	6-34
6.2.1 Performance Considerations.....	6-34
6.2.2 Terminal Manager Restrictions	6-35
6.3 Memory Management.....	6-37
6.3.1 External Data Items	6-41
6.4 Memory Testing and Error Handling.....	6-41
6.4.1 Memory Access Violations.....	6-41
6.4.2 Logging Errors to the Runtime's Error File	6-42
6.4.3 Runtime Memory Tracking and Testing.....	6-43
6.4.3.1 Memory handling descriptions.....	6-43
6.4.3.2 Memory tracking	6-44
6.4.3.3 Memory bounds checking	6-44
6.5 Screen Section.....	6-45
6.5.1 Advantages	6-46
6.5.2 Structure.....	6-46
6.5.3 Syntax	6-47
6.5.4 Comparison to Field-level	6-50
6.5.5 Using Screen Section Embedded Procedures.....	6-51
6.6 Data Validation	6-54
6.7 Exiting From ACUCOBOL-GT Programs	6-55
6.8 Multiple Execution Threads.....	6-56
6.8.1 Thread Fundamentals.....	6-57
6.8.1.1 LAST THREAD.....	6-58
6.8.2 Data Sharing Among Threads	6-59
6.8.2.1 LOCK THREAD and UNLOCK THREAD	6-60
6.8.3 Thread Communication	6-61
6.8.3.1 SEND and RECEIVE	6-61
6.8.4 Thread Priorities	6-64
6.8.5 Threading Rules That Affect Windows and ACCEPT Statements	6-64
6.8.6 Thread Pausing	6-67
6.8.7 Multithreading and Multiprocessor Systems	6-68
6.8.8 Thread Interaction With Run Units	6-69

6.9 Working with External Sort Modules (UNIX)	6-70
6.9.1 Before Using an External Sort Module.....	6-70
6.9.2 Linking in a Third-Party Sort Module	6-70

Index

1

Introduction

Key Topics

ACUCOBOL-GT Documentation	1-2
Product Overview	1-2
Document Overview	1-12
Supported Hardware	1-17
Environment Variables	1-18
Notation	1-20
How to Get Help	1-20

1.1 ACUCOBOL-GT Documentation

This is Book 1 of a four-book set that describes the features of ACUCOBOL-GT®.

The other books in this set include:

- Book 2, *ACUCOBOL-GT User Interface Programming*
- Book 3, *ACUCOBOL-GT Reference Manual*
- Book 4, *ACUCOBOL-GT Appendices*

This set is augmented by four additional volumes:

- *Getting Started*
- *Transitioning to ACUCOBOL-GT*
- *A Guide to Interoperating with ACUCOBOL-GT*
- *A Programmer's Guide to the Internet*

Please see **section 1.3, “Document Overview,”** for a brief introduction to all of these volumes.

1.2 Product Overview

ACUCOBOL-GT is part of the *extend*® family of Micro Focus solutions. ACUCOBOL-GT is an ANSI® 1985 COBOL compiler that also includes components of the ANSI X3.23a-1989 supplement. It is designed to provide a powerful development environment for a wide range of computers.

Fast compile speed, clear error messages, and a multi-window source level debugger work together to provide a high performance, easy-to-use COBOL development platform.

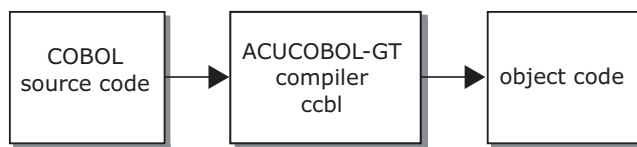
Portable object code, a generic interface to a variety of file systems, and a device-independent terminal interface help to simplify the distribution of applications developed with ACUCOBOL-GT.

In addition to portable object code, ACUCOBOL-GT can generate and execute object files that contain native instructions for specific types of processors. This enables you to optimize the use of CPU resources on the host machine while maintaining full portability within the same family of processors.

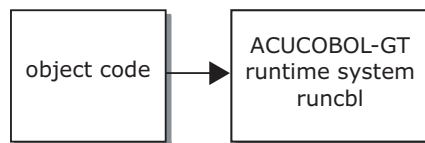
Installation instructions for all computer types are provided in the *Getting Started* book. The compiler, the runtime, and some utility programs have different names, depending on the host system. For simplicity, we refer to the compiler as **ccbl**, the runtime as **runcbl**, and the utilities by their base names (**vutil**, **vio**, **logutil**, and **cblutil**).

Unless otherwise indicated, the references to “Windows” in this manual denote the following versions of the Windows operating systems: Windows XP, Windows Vista, Windows 7, Windows 2003, Windows 2007, Windows 2008 R2. In those instances where it is necessary to make a distinction among the individual versions of those operating systems, we refer to them by their specific version numbers (“WindowsXP,” “Windows Vista,” etc.).

ACUCOBOL-GT is a single-pass compiler that produces an object file.



The object file is ready for immediate execution by the ACUCOBOL-GT runtime system. No linking step is required.



1.2.1 Portability and Compatibility

One of the primary focuses of the product is program portability. ACUCOBOL-GT offers source code compatibility modes for five popular compilers:

- Ryan McFarland's RM/COBOL™
- ICOBOL (originally Data General Interactive COBOL)
- VAX™ COBOL
- IBM® COBOL
- HP COBOL

Although not provided as a “mode” that is turned on with a compiler switch, there is extensive built-in compatibility for MF COBOL. These facilities allow programs to be moved to or from these environments with minimum effort.

ACUCOBOL-GT also offers fully portable object code. Any program compiled on one machine using ACUCOBOL-GT will run unmodified on any other machine that supports ACUCOBOL-GT--recompiling is not required. This allows the developer access to a diverse user base with a minimum investment in hardware.

1.2.2 Native Instructions

ACUCOBOL-GT can also generate and execute object files that contain native instructions for selected families of processors. This results in generally faster code at the cost of reduced portability and larger object files.

Generally speaking, a native-code module will be a much more efficient user of the CPU. However, a native-code module will typically be several times larger than a portable object code module. You must weigh the benefits of better CPU utilization versus the cost of more memory used. For systems where memory is tight, one solution is to use native code only for those modules that are CPU intensive while leaving the rest of the system in portable object code.

When you compile using native instructions, the resulting object file may be run only on a machine containing the appropriate processor. Note, however, that within this class of machines, the object file is still portable. For example, you could run an Intel object file on a Windows machine or on a Linux machine. There is no need (or benefit) to compile directly on the target machine.

Please refer to **section 1.4** of this chapter for the complete list of platforms supported by ACUCOBOL-GT and to **section 1.4.1** for the complete list of processors supported by its native-code functionality.

Object files that contain native-code instructions are similar to normal COBOL object files. The only difference is the internal instruction set used. Native-code object files are run in the same fashion as normal (portable) objects: by using the runtime. The runtime acts as the memory manager and support libraries for the native-code module. You may mix native and non-native modules in a single run freely.

1.2.3 The ACUCOBOL-GT Runtime

After a program is successfully compiled it is ready for immediate execution with the ACUCOBOL-GT runtime. There is no link step. Detailed instructions on the use of the runtime are provided in Chapter 2, **section 2.3, “Using the Runtime System.”**

On UNIX, Linux, OpenVMS, and MPE/iX systems, the runtime executable is named “runcbl” or “runcbl.exe”. On some UNIX systems the runtime is provided as a shared object library named “libruncbl.so” or “libruncbl.a”.

Several distinct runtimes are available for Windows systems. Each is licensed separately.

- The standard Windows runtime is named “wrun32.exe”. It is used with all standard deployments.
- Thin client deployments use a special runtime named “acuthin.exe”. **acuthin** is used in conjunction with AcuConnect and a standard runtime, both of which are installed on the application host. For more information about thin client technology, see Chapter 1, section 1.3.2, “Thin Client,” in the *AcuConnect User’s Guide*.

- To support applications originally developed for Extended DOS, as well as other character-based applications, we offer a Windows console *console* runtime, named “crun32.exe”. The console runtime uses the Windows Console API and runs in a virtual DOS window. For more information, see **section 1.2.3.1**, below.
- The Alternate Terminal Manager (ATM) runtime, named “run32.exe”, allows you to use a 32-bit Windows server in much the same way that some UNIX servers are used. With the ATM runtime, the user can telnet to the Windows server (with a third-party telnet service) to execute character-based ACUCOBOL-GT programs in the telnet window. The ATM is described in more detail in Chapter 4, **section 4.1.2, “Alternate Terminal Manager (ATM).”**
- To support Windows-based deployment of applications to be accessed via the World Wide Web, we offer a special Web runtime and CGI runtime. For more information on these options, see the book titled, *A Programmer’s Guide to the Internet*.

1.2.3.1 Windows console runtime

The Windows console runtime (crun32.exe) is a 32-bit Windows runtime that provides support for applications originally developed for Extended DOS and other character-based systems. The console runtime uses the Windows Console API and runs in a virtual DOS window. The Windows console runtime is sold and licensed separately.

The console runtime includes support for:

- calling Windows DLLs
- using Acu4GL modules without having to relink the runtime

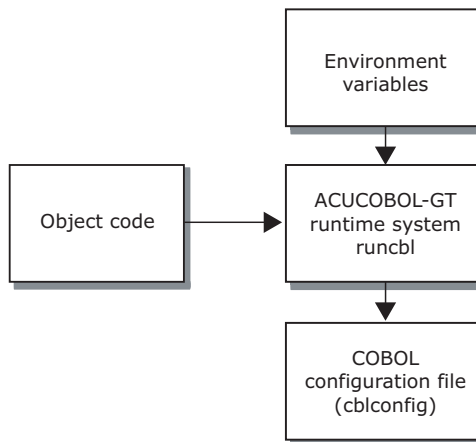
Although the console runtime is designed to run ACUCOBOL-GT applications developed for Extended DOS, your applications may require some modifications. For example, the console runtime supports printing capabilities based on the Windows model. Programs that rely on DOS printing functions must be changed. You should examine your applications for the use of DOS functions that are not supported in the Windows environment.

Two runtime configuration variables allow you to tune the execution environment: **DOS_SYS_EMULATE** and **DOS_BOX_CHARS**. For details, see their respective entries in Book 4, Appendix H.

Note: Should you need to relink the console runtime, refer to the instructions in section 6.3.6 of *A Guide to Interoperating with ACUCOBOL-GT*.

1.2.4 Runtime Configuration

Users of ACUCOBOL-GT can modify many aspects of the runtime environment on a site-by-site or user-by-user basis without recompiling. This is accomplished in a text file known as a COBOL configuration file, and by environment variables:



For example, the location of data files, names of devices, color, text of error messages, file buffering, and screen editing functionality can all be maintained outside of the compiled programs. For maximum flexibility, we strongly encourage you to review Appendix H in Book 4, *Appendices*, for configuration variables that may be useful to your site.

If machine-specific behavior is required, a program can directly inquire what type of machine it is running on. All of these features work together to allow the developer to support a wide range of machines with a minimum amount of resources.

1.2.5 Graphical Technology

ACUCOBOL-GT includes broad programming and runtime support for adding a native *Graphical User Interface* (GUI). In addition, many graphical elements, such as floating windows, labels, entry fields, push buttons, radio buttons, check boxes, list boxes, combo boxes, and others, are emulated with character-based components when run on text-mode systems. Some graphical elements are not emulated in text-mode environments and are simply ignored by the runtime (see Book 2, *User Interface Programming*, Chapter 1).

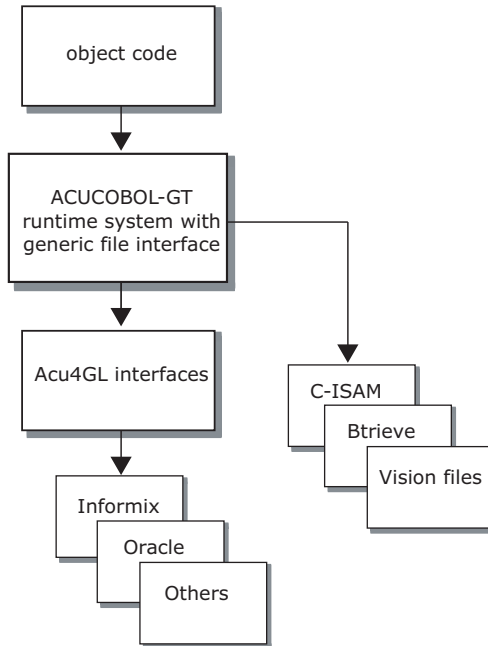
ACUCOBOL-GT GUI syntax extensions make it easy to implement specialized user interfaces that are well suited to both graphical and character-based systems, and which remain fully portable. For a discussion of user interface development approaches, see section 1.5, “Creating Portable User Interfaces,” in Book 2, *User Interface Programming*.

The graphical capabilities supported in ACUCOBOL-GT include:

- syntax extensions for creating floating windows, toolbars, and controls (such as buttons, entry fields, and labels)
- support for creating and managing menu bars with pull-down submenus
- configuration variables for customizing windows, importing icons, and mapping colors
- library support for host specific features such as file *open* and *save-as* dialog boxes, message boxes, font selections, print spoolers, and context-sensitive help

1.2.6 File System Flexibility

ACUCOBOL-GT also includes a generic architecture for connecting to file systems. This means that developers can choose from a variety of file systems and database management systems, as shown in the following diagram:



On all platforms except VMS and OpenVMS, ACUCOBOL-GT is shipped with the powerful, performance optimized Vision indexed file system. On VMS and Open VMS systems, ACUCOBOL-GT uses the native RMS file system. On the HP e3000 platform, in addition to Vision, ACUCOBOL-GT interfaces with MPE/KSAM files.

extend's interfaces to other file systems are licensed separately.

1.2.7 Complementary Technologies

Acu4GL

ACUCOBOL-GT uses Acu4GL® libraries to access information stored in relational database management systems (RDBMSs). *Data dictionaries* generated by the compiler guide the libraries in mapping the field names and data types that are passed between COBOL and the database engine.

The essence of Acu4GL libraries is that *standard COBOL I/O statements are used to access databases*.

Acu4GL dynamically generates industry-standard SQL from the COBOL I/O statements. As the ACUCOBOL-GT runtime module is executing your COBOL application, Acu4GL is running “behind the scenes” to match up the requirements and rules of both COBOL and the RDBMS to accomplish the task set by your application.

AcuBench

AcuBench® is an integrated development environment for COBOL. AcuBench extends and enhances the ACUCOBOL-GT compiler and runtime system with a powerful suite of GUI-based development tools for COBOL. With AcuBench you can develop and maintain your COBOL applications in an integrated, developer friendly Microsoft Windows environment and deploy your applications on any of the more than 600 platforms supported by Micro Focus.

AcuSQL

AcuSQL® is an add-on tool that supports embedded SQL (ESQL) COBOL programs. It gives COBOL applications access to Microsoft SQL, IBM DB2, MySQL and ISO/ANSI SQL92 compliant data sources.

AcuXDBC

AcuXDBC™ is a data management system, designed to integrate ACUCOBOL-GT data files into a relational database-like environment. AcuXDBC enables you to apply SQL and relational database concepts to

your COBOL data files resulting in data that is accessed and managed in much the same way as many of today's popular relational database management systems.

AcuXDBC is the next generation of AcuODBC and is engineered to provide broader flexibility in the way your COBOL data is accessed and maintained. Like previous versions of AcuODBC, AcuXDBC lets you retrieve and update ACUCOBOL-GT's Vision indexed files, relative files, and sequential files from Windows-based applications including Microsoft Word, Excel, and Access. Business Intelligence tools such as Crystal Reports[®] Professional, and custom applications developed in ODBC supported environments such as Visual Basic[®] are supported as well. With the enterprise edition, new functionality lets you retrieve data through Java applications that utilize JDBC standards. Direct SQL access to your ACUCOBOL-GT data is now available in both the Windows and UNIX environments.

AcuConnect

AcuConnect[®] is a client/server technology that is an integral part of *extend's* distributed computing solution. AcuConnect lets you implement a client/server system in which the client piece can be as "thin" or as "fat" as you need.

AcuConnect has two deployment environments. With AcuConnect's distributed processing deployment, users can distribute application logic between client and server machines in a way that best suits their needs. AcuConnect users can also take advantage of Micro Focus's Thin Client technology, which lets you run the user interface (UI) portion of your application on a graphical display host while the rest of the application and data reside on the server.

AcuServer

AcuServer[®] is an add-on module that provides remote file access services to ACUCOBOL-GT applications. AcuServer is available for applications running on most UNIX, Linux, and Windows TCP/IP based networks and executing with ACUCOBOL-GT runtime Version 5.0 or later.

With AcuServer, your applications gain:

- the ability to create and store data files on any UNIX or Windows NT/Windows 2000 server equipped with AcuServer
- full function remote access from UNIX, Linux, and Windows clients to all Vision, relative, sequential, and object files stored on an AcuServer server
- full record locking support of all Vision and relative files
- transparent access of remote and local files

AcuServer is described briefly in Chapter 5 of this book. A separate *AcuServer User's Guide* is included with the server software and provides additional details.

1.3 Document Overview

ACUCOBOL-GT's features and how to use them are documented in a four-volume set of books. These books include: Book 1, *User's Guide*, Book 2, *User Interface Programming*, Book 3, *Reference Manual*, and Book 4, *Appendices*. This set is supplemented by four essential volumes: *Getting Started*, *Transitioning to ACUCOBOL-GT*, *A Guide to Interoperating with ACUCOBOL-GT*, and *A Programmer's Guide to the Internet*. These books are briefly outlined in the following sections.

This *User's Guide* describes how to compile and run programs with ACUCOBOL-GT and includes programming suggestions for both new and experienced programmers. It also includes details about how to use the debugger and file utilities. Please note that this book does not teach the COBOL programming language. A companion book, the *ACUCOBOL-GT Reference Manual*, describes every verb included in the language.

1.3.1 User's Guide

The *User's Guide* is Book 1 of the ACUCOBOL-GT documentation set.

- Chapter 1:** “Introduction” gives an overview of this *User’s Guide* with information on supported hardware, environment variables, and command notation.
- Chapter 2:** “Compiler and Runtime” describes how to use the ACUCOBOL-GT compiler and its runtime system.
- Chapter 3:** “Debugger and Utilities” discusses the source-level debugger and the file system maintenance utilities.
- Chapter 4:** “Terminal Manager” gives a detailed description of the operational aspects of the ACUCOBOL-GT screen manager, along with instructions for creating new terminal definitions.
- Chapter 5:** “File Processing” describes the transaction management and client/server features of the ACUCOBOL-GT language.
- Chapter 6:** “Programmer’s Guide” gives suggestions for taking advantage of the power and flexibility of ACUCOBOL-GT.

1.3.2 User Interface Programming

User Interface Programming is Book 2 of the ACUCOBOL-GT documentation set.

- Chapter 1:** “Introduction” provides a general introduction to floating windows and graphical controls, as well as detailed discussions on interface programming issues and approaches, and basic windowing concepts.
- Chapter 2:** “Floating Windows” provides an introduction to the properties of moveable, floating windows.
- Chapter 3:** “Graphical Controls” provides an introduction to the properties of graphical controls, as well as specific discussions regarding the use of bitmap buttons and paged list boxes.

- Chapter 4:** “Supporting Concepts and Related Issues” discusses a variety of issues related to the use of floating windows and graphical controls.
- Chapter 5:** “Control Types Reference” describes the components of a control, in general, and the special properties of each control type.
- Chapter 6:** “Events Reference” describes every event that the ACUCOBOL-GT runtime can return to the program.
- Chapter 7:** “Using the Mouse” describes how to use the automatic and programmable mouse functions available in ACUCOBOL-GT.
- Chapter 8:** “Menu Bars and Pop-Up Menus” describes the process of creating, integrating, and managing program menu bars and pop-up menus.
- Chapter 9:** “Color Mapping” describes how to configure the program and environment to take advantage of color display devices.
- Chapter 10:** “Help Automation” describes how to use the syntax extensions and runtime functions that support context-sensitive help.
- Chapter 11:** “Tips and Hints” answers commonly asked questions having to do with interface programming and the use of floating windows and controls.
- Chapter 12:** “UI Terminology” provides a glossary of common user interface terminology.

1.3.3 Reference Manual

The *Reference Manual* is Book 3 of the ACUCOBOL-GT documentation set.

- Chapter 1:** “Introduction” gives an overview of the *Reference Manual* and includes information on the conventions used.
- Chapter 2:** “Program Structure” describes the overall structure of a COBOL program and provides basic information about the format of user-defined words, literals, and source code.
- Chapter 3:** “Identification Division” gives a detailed description of the COBOL Identification Division.
- Chapter 4:** “Environment Division” describes the COBOL Environment Division.
- Chapter 5:** “Data Division” describes the COBOL Data Division.
- Chapter 6:** “Procedure Division” describes the COBOL Procedure Division. This also contains a complete discussion of all the verbs used by ACUCOBOL-GT.

1.3.4 Appendices

Appendices is Book 4 of the ACUCOBOL-GT documentation set.

- Appendix A:** “Specifications” describes how ACUCOBOL-GT compares with the ANSI standard and also describes various internal limits set by the compiler.
- Appendix B:** “Reserved Words” provides a complete list of words reserved in ACUCOBOL-GT and notes which dialect is the source of each word.
- Appendix C:** “Changes Affecting Earlier Versions” describes features of ACUCOBOL-GT that have changed since earlier versions, and how to accommodate these differences.
- Appendix D:** “Compiler Error Messages” lists and describes the error messages that the compiler can generate.

- Appendix E:** “File Status Codes” is a table of the various file status and transaction status values and their meanings.
- Appendix F:** “Intrinsic Functions” provides detailed information about the latest ANSI intrinsic function module standards for COBOL.
- Appendix G:** “Reserved For Future Use” is a placeholder that serves to preserve the historic placement of Appendices H and I.
- Appendix H:** “Configuration Variables” summarizes the entries that are available for use in the ACUCOBOL-GT runtime configuration file.
- Appendix I:** “ACUCOBOL-GT Library Routines” describes callable routines that are built into the ACUCOBOL-GT runtime system. These routines may be called directly by your programs.

1.3.5 Getting Started

Getting Started contains information about how to install, configure, and rapidly become productive with ACUCOBOL-GT.

1.3.6 Transitioning to ACUCOBOL-GT

Transitioning to ACUCOBOL-GT provides information on how to convert from several popular COBOL dialects to ACUCOBOL-GT. It includes chapters on RM COBOL, ICOBOL, HP COBOL, and IBM DOS/VS COBOL.

1.3.7 A Guide to Interoperating with ACUCOBOL-GT

A Guide to Interoperating with ACUCOBOL-GT describes the capabilities and methods that enable your ACUCOBOL-GT applications to interoperate with other languages and other technologies. Topics include working with

Java, working with Windows, working with C and C++, deploying on the Web, accessing mobile devices, working with OLTP systems, and working with non-COBOL data.

1.3.8 A Programmer's Guide to the Internet

A Programmer's Guide to the Internet is designed to teach ACUCOBOL-GT developers how to deploy applications on the Internet. It includes a description of alternative approaches to Internet deployment and specific instructions on implementing each approach.

1.3.9 Related Documents

Product documentation in the form of user's guides and technology supplements are included on the ACUCOBOL-GT product media and can be installed when the ACUCOBOL-GT software is installed, or installed at a later date, or accessed directly from the product media when the media is a CD-ROM. The set of documents includes information about: Acu4GL, AcuBench, AcuConnect, AcuXDBC, AcuServer, and AcuSQL. Product documentation is also available for viewing and downloading from Micro Focus's website, www.microfocus.com.

1.4 Supported Hardware

ACUCOBOL-GT is available on a wide range of machines. In this manual, these machines are distinguished by their operating system. These are the main classes of currently supported operating environments:

- Windows:** Windows Vista, Windows XP, Windows NT 4.0 or later, Windows 2000, Windows 2003; and the following 64-bit versions: Windows Server 2003 and 2008 x64, Vista x64.
- UNIX/Linux:** Virtually any UNIX[®] or Linux operating system.
- VMS:** The VMS[™] and OpenVMS operating system.

MPE/iX: The HP e3000 operating system.

Differences among the operating environments are detailed in the appropriate sections of the manual. See, also, *A Guide to Interoperating with ACUCOBOL-GT*.

1.4.1 Native Code Supported Processors

The native code generation functionality of ACUCOBOL-GT currently extends to the following popular families of processors:

Intel:	486, Pentium®, Pentium II, Pentium III (and compatible); protected (32-bit) mode only.
PA-RISC:	32-bit and 64-bit PA-RISC processors under HP-UX and MPE/iX.
PowerPC:	32-bit and 64-bit IBM pSeries™ running under AIX®
SPARC:	v7, v8, v9 (and compatible); 32-bit, and SPARC v9 64-bit mode.

Generated for one of these families, the native code object file is completely portable among all the supported members of that family.

1.5 Environment Variables

This manual often refers to the host machine's *environment variables*. These are values maintained by the host operating system that can be changed by the user. Exactly how an environment variable is set differs among various operating systems:

1. With Windows NT/2000/2003/XP, environment variables are set using the “System” applet in the Control Panel. Windows must be restarted for the new values to take effect.
2. With Windows 98/ME, environment variables can be defined in the “autoexec.bat” file (with the same syntax that’s used with MS-DOS). Any changes made to the “autoexec.bat” file require restarting

Windows to take effect. Environment variables can be defined temporarily by booting to MS-DOS mode, defining variables with the SET command and then starting Windows. Variables defined in this way persist until the system is rebooted.

3. On Windows systems where the console runtime (**crun32**) is used, environment variables are inherited from the Windows environment. Temporary environment variables can be defined in the console window (DOS-box) with the SET command. For example, to set the environment variable “SORT_DIR” to “C:\TEMP”, you would use the following line:

```
SET SORT_DIR=C:\TEMP\
```

Variables defined in this way persist until the DOS-box is closed. Forward slashes (/) may be used in place of backslashes (\). Upper-case and lower-case letters are interchangeable.

4. On UNIX and Linux systems, the environment is controlled in one of two fashions, depending on which command shell you are using. If you are using the Bourne shell (“sh”) or the Korn shell (“ksh”), then you set a shell variable to the desired value and then export that variable. For example:

```
SORT_DIR=/tmp/; export SORT_DIR
```

If you are using the C-shell instead, then you use the **setenv** command. For example:

```
setenv SORT_DIR /tmp/
```

Upper-case and lower-case environment variables are distinct.

If you are using a different shell, see the documentation for that shell, or ask your system administrator.

5. On VAX/VMS systems, you set a *symbol* to the desired value. For example:

```
SORT_DIR == "$DISK1:[TEMP]"
```

Unlike UNIX, upper-case and lower-case variable names are treated the same on VMS.

Environment variables for handling temporary files

Temporary files that are not placed in a specific directory via a configuration variable, such as `SORT_DIR`, can be directed to a specific directory by defining the `A_TMPDIR` or `TMPDIR` environment variables. The runtime gives preference to `A_TMPDIR`. This allows users to specify an alternative temporary directory exclusively for ACUCOBOL-GT temporary files. On UNIX systems, `TMPDIR` is the traditional variable for specifying a location for temporary files. On Windows, the runtime system also checks the `TEMP` and `TMP` environment variables.

1.6 Notation

When commands are described in this manual, the following conventions are used:

Plain text - any text appearing in the normal text font refers to a literal value that must be used exactly as shown.

Italics - any text appearing in *italics* refers to a generic item. These are explained in the text that follows the command.

Brackets - any text appearing in brackets ([]) refers to an optional item. The text following the command describes the action taken if these optional items are used.

For example, in this command:

```
vutil -info [-x] [files]
```

the words “vutil” and “-info” must appear exactly as shown. The “-x” and “*files*” items are optional. If they are used, the “-x” must be typed literally while the “*files*” would be described in the text explaining the command.

1.7 How to Get Help

For the latest information on contacting customer care support services go to:

<http://www.microfocus.com/about/contact>

For worldwide technical support information, please visit:

<http://supportline.microfocus.com/xmlloader.asp?type=home>

1.7.1 Handling Compilation Problems

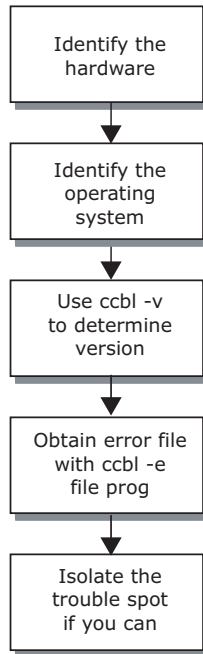
When you compile, direct the error messages to a file, and save this file to examine. The compile command for capturing error output in a file is:

```
ccbl -e filename program-name
```

From the information in this file, you will probably be able to determine the cause of the problem and make the necessary changes. If not, we would be pleased to assist you. Here's an overview of the information we need to know, most of which you can simply fax to us:

- We need to know your hardware and operating system.
- It's also helpful for us to know which version of the ACUCOBOL-GT compiler you are using. In case you've encountered a compiler anomaly, a later version of the compiler may solve your problem. To determine the version you are running, type: `ccbl -v`
- We need a copy of the compile-time error file. If any message appears on your screen, we need the exact text of the message.

- Finally, if you can isolate the section of the code that is causing the problem, please fax us a printed copy of this code.



1.7.2 Handling Program Execution Problems

The runtime system contains a built-in debugger that allows you to view the source code while you are debugging. To prepare to use source debugging, you must compile with one of the compiler options that include the source code in the compiled object file, such as “-Ga”.

For example:

```
ccbl -Ga -o myprog myprog.cbl
```

Where:

-Ga	includes the source code in the compiled object file and enables other helpful debugging features (see section 2.2.13, “Debugging Options,” for detailed information on those features).
-o	allows you to specify the name of the output file for the object code
<i>myprog</i>	is the user-specified name of the output file
<i>myprog.cbl</i>	is the name of the file containing the COBOL source code

See Chapter 2, sections [section 2.2.13](#) and [section 2.2.16](#) in this book for complete listings of compiler options available for use with the debugger. See [section 2.2](#) for syntax rules governing the specifying of compiler options on the command line.

Using the debugger

Then, when you execute the program, specify “-d” to turn on the debugger. For example:

```
runcbl -dle errfile myprog
```

Where:

-d	turns on the debugger
-l	causes the contents of the runtime configuration file to be included in the error output
-e	causes the error output to be placed in the file named immediately after the option
<i>errfile</i>	is the user-specified name of the error file (Be sure to give a file name if you use the “-e” option.)
<i>myprog</i>	is the name of your object file

Use “s” to step through the program one step at a time, or use the other options described in [section 3.1.3](#) in this book.

Using file trace

Another tool that is helpful in dealing with program execution problems is the file trace feature of the debugger. It enables you to save information about all file OPENS, READS, and WRITES. File trace can be used even if the program was compiled without the debugging option. File trace information is saved in the error file. (The file trace feature is described in detail in [section 3.1.4](#).)

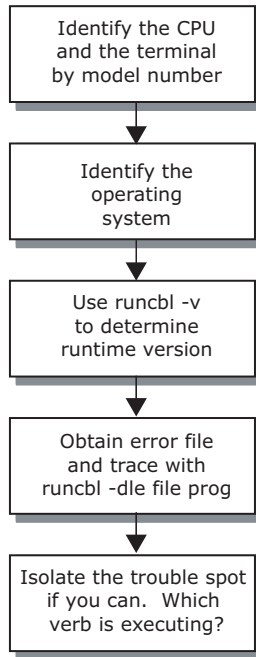
Using screen trace

Another tool that is helpful in dealing with program execution problems is the screen trace feature of the debugger. The screen trace feature enables you to save information about DISPLAYS of screen section items and CREATEs, DISPLAYs, MODIFYs and INQUIREs of ActiveX objects. Screen trace can be used even if the program was compiled without the debugging option. The screen trace information is saved in the error file. (The screen trace feature is described in greater detail in [section 3.1.5](#).)

Working with your Technical Support Representative

We would be happy to assist you with the resolution of runtime problems. Questions that require extended research are entered into our Technical Support database and are assigned a tracking number. We use this tracking number when we phone to give you updates on our investigation.

Here's the information we'll need to start our research:



2

Compiler and Runtime

Key Topics

Introduction	
Using the Compiler	2-3
Using the Runtime System	2-75
Compatibility Modes	2-92
Source Formats	2-93
COPY Libraries	2-95
Source Code Control	2-99
Runtime Configuration	2-100
File Name Interpretation	2-107
Calling Subprograms	2-115
Reducing the Size of the Runtime	2-120
acushare Utility Program	2-121
General Preprocessor Interface	2-129

2.1 Introduction

ACUCOBOL-GT is a single-pass ANSI X3.23-1985 COBOL compiler (**ccbl**). Errors encountered during compilation are displayed on the screen (or written to an error file if one is requested) referencing the line where the error was detected by source file and line number. A source listing, a symbol table listing, and general information can be created on request.

A successful compile produces an object-code file. This file is ready to be run by the ACUCOBOL-GT runtime system, **runcbl**. No linking is needed to run the program. Any programs that are called during execution are loaded dynamically at runtime by **runcbl**. Subprograms written in C may be linked into the runtime system directly, or loaded dynamically if packaged in a Windows DLL or UNIX/Linux shared object library, and then called by a COBOL program using the CALL verb.

Note: The compiler, the runtime, and some utility programs have different names, depending on the host system. For simplicity in this manual, we refer to the compiler as **cbl**, the runtime as **runcbl**, and the utilities by their base names (**vutil**, **vio**, **logutil**, and **cblutil**).

runcbl has a symbolic source-level debugger built into it. You can run any program under the debugger by specifying the debugger option (“-d”) at runtime. The debugger runs in its own window that overlays the running program. This prevents the debugger’s output from interfering with the program’s output.

2.2 Using the Compiler

The compiler is called **ccbl**. It can be run in one of three different modes. These modes are:

Compile	Compiles COBOL programs
Help	Prints a command-line summary
Version	Prints the version and copyright

The compile mode is the default mode. The other modes are activated by command-line options. The compile mode is discussed here; other modes are described in separate sections in this chapter.

To compile a program, enter the following:

```
ccbl [options] program
```

No options are required. When no options are specified, the compiler prints an error listing on the user's terminal and names the object file "*source-name.acu*". The compiler runs in VAX COBOL compatibility mode by default.

Many options are available. These are all indicated by an initial hyphen (minus sign). These options are divided into different groups. Each group (except the first) has a distinguishing letter. For example, the options that control listings all start with the letter "L". Any set of options from the same group may be specified together. For example, the "-Lo" option and the "-Ls" option may be grouped together as "-Los". See below for examples. Command-line options that begin with a hyphen are not case sensitive. For example, "-Lo" and "-lo" are equivalent.

Options from the same group that require arguments may also be specified together, as long as the required arguments immediately follow the combined options, in the same order as the options. For example,

```
-o file1 -e file2
```

may also be specified as

```
-oe file1 file2
```

The complete list of options is given in the following sections.

The options used to compile a COBOL program are automatically embedded in the header of the resulting object file. Use the “-x” option of the “cblutil -info” command to output these options.

The compiler also supports several *compiler directives*. Compiler directives are directly embedded in the source code and cause the compiler to behave as specified by the directive. For more information, see **section 2.2.15, “Conditional Compilation Options.”** Support for compiler directives is likely to expand in future versions.

2.2.1 Standard Options

- e This option must be followed by a file name (as the next separate argument). When specified, this option causes the error listing to be written to the specified file instead of the screen. This file is removed if no errors are found. The *acurfap* syntax can be used to specify a file located on a system being served by AcuServer or AcuConnect. See **section 2.2.18.1, “Remote file name handling.”**
- o This option must be followed by a file name (as the next separate argument) which becomes the name of the object file instead of “*source-name.acu*”. This file is removed if the compiler detects errors in the source.

This option supports the use of *acurfap* syntax to create remote files and libraries. For detailed rules regarding the file name specifications for remote objects, see **section 2.2.18.1, “Remote file name handling,”** and **section 3.2.2.1, “Creating remote object libraries,”** in Chapter 3.

- v This option has multiple applications:

If it is the first and only option on the command line, then the compiler runs in “Version” mode (see [section 2.2.21, “Help, Version Information, and Communication With C Programs”](#)). Using “-v” you can display version information, the copyright notice, and other information.

Otherwise, if it is used in combination with other options, it causes the compiler to be verbose about its progress.

Because “-v” is the lead-in sequence for the video options (see [section 2.2.11, “Video Options”](#)), this option should be specified by itself.
- w Causes warning messages to be suppressed (a warning condition is never a fatal compilation error). Suppressing warning messages can be helpful when you are converting programs from another COBOL dialect that uses slightly different syntaxes. Note that “-w” negates the effect of “-Wa” and “-WI”, which generate additional warning messages.
- x This causes the CBLFLAGS environment variable to be ignored. This is described in detail in [section 2.2.20, “CBLFLAGS Environment Variable.”](#)

2.2.2 Native Code Options

ACUCOBOL-GT supports several options that cause it to generate and execute object files that contain native instructions for select families of processors. These options may *not* be combined with other options into a single option. For example, it is an error to specify “-nv”; you must specify “-n -v” instead.

The compiler uses **cbutil** to produce the native-code object file. **cbutil** is installed in the same directory as the compiler by default. The compiler searches the execution path if it cannot find **cbutil** in its own directory.

You run a native-code object module just like a portable-code object module. You may mix native-code objects and portable-code objects in the same run. Native objects may be placed in libraries just like portable objects.

You can use the debugger with a native-code module in the same fashion as with a portable-code module. The only restriction is that you may not begin program execution at an arbitrary point in a native-code module (the “@!” command). Also, to prepare a native-code object for use with the debugger, you must compile the program with one of the debugging options, such as “-Ga” or “-Gd”. A complete list of debugging options is located in **section 2.2.13, “Debugging Options.”**

Example:

To compile and run the “tour.cbl” program as a portable-code object, use the following commands:

```
ccbl tour.cbl
runcbl tour
```

To compile and run the same program as a native-code object, use the following:

```
ccbl -n tour.cbl
runcbl tour
```

These are the ACUCOBOL-GT compiler native code options:

-n This option causes the compiler to produce native code for the host machine (instead of processor-independent portable code). If native code generation is not supported for the host machine, the compiler generates an error.

The word-size of the native code generated with the “-n” option matches the internal word-size of the compiler, and not that of the host machine’s processor. It does this because the word-size of the native code must match the internal word-size of the runtime to execute correctly. Therefore, a 32-bit compiler, even when running on a 64-bit machine, will produce 32-bit native code.

If running a program compiled with “-n” on a Windows machine that has Data Execution Protection (DEP) enabled for all processes, you need to set the runtime configuration variable

USE_EXECUTABLE_MEMORY to “TRUE”.

--noinlineCall This option turns off a feature that enhances the performance of called subroutines. By default, this feature is turned on, but in version 8.0 had to be specified by the “--inlineCall” command. It is not recommended that you turn off this feature. It is useful if you have interest in testing the potential performance difference.

--intel
or
--ia-32 This option causes the compiler to produce 32-bit native code for Intel-class processors (486, Pentium, Pentium II, Pentium III, or compatible processors). You can use this option from any host machine. This is the same as specifying “-n” when the host machine is an Intel-class machine.

--pa_risc or --pa	This option causes the compiler to produce 32-bit code for PA-RISC Version 1.0 processors running the HP-UX or MPE/iX operating systems. Note that the runtime must be a 32-bit runtime, to match the code contained in the object file. The 32-bit runtime runs without modification on PA-RISC 2.0 platforms as well, providing a portable solution. You can use this option from any host machine. This is the same as specifying “-n” with a 32-bit compiler on an HP-UX or MPE/iX machine.
--pa_risc_2.0 or --pa2	This option causes the compiler to produce 64-bit code for PA-RISC Version 2.0 processors running the HP-UX operating system. Note that the runtime must be a 64-bit runtime, to match the code contained in the object file. You can use this option from any host machine. This is the same as specifying “-n” with a 64-bit compiler on an HP-UX machine.
--power	Produces code that is compatible with POWER and POWER2 processors, as well as PowerPC and later POWER series processors. This option allows you to use a wide range of machines, but it may affect performance.
--powerpc or --ppc	This option causes the compiler to produce 32-bit PowerPC native code for pSeries processors running AIX. You can use this option from any host machine. This is the same as specifying “-n” with a 32-bit compiler on a pSeries machine. Note that you can compile native code only for machines with a POWER3 or later chip, not with POWER2 or earlier.
--powerpc_64 or --ppc64	This option causes the compiler to produce 64-bit PowerPC native code for pSeries processors running AIX. You can use this option from any host machine. This is the same as specifying “-n” with a 64-bit compiler on a pSeries machine.

- sparc** This option causes the compiler to produce 32-bit native code for SPARC (v7 - v9) processors. You can use this option from any host machine. This is the same as specifying “-n” with a 32-bit compiler on a SPARC-based host machine.
- sparc_v9** This option causes the compiler to produce 64-bit native code for SPARC version 9 processors. You can use this option from any host machine. This is the same as specifying “-n” with a 64-bit compiler on a SPARC v9-based host machine.

2.2.3 Listing Options

By default, ACUCOBOL-GT does not generate a listing. The options described below cause a listing to be created and control its contents. In addition to the items specified for these options, the listing file contains all the options given to the compiler. For example, a listing file could begin with

```
iobench.cbl          Sat Sep 18 06:55:54 2005  ACUCOBOL-GT v7.3.0 Page: 0001
..\bin\ccbl32 -Lfo @.lst iobench.cbl
CBLFLAGS: -v -Za
```

Note that CBLFLAGS (or any other variable) is not included in the listing unless it is set

- La** This option causes the compiler to create and display separate tallies for “warnings” and “cautions.” Without this switch the compiler combines the tallies and reports the sum under the category of “warnings.”
- Lc** This option creates a cross-reference table at the end of the listing. The cross-reference first lists all Working-Storage data items in alphabetical order, then all section names and paragraph names. Included for each item are: the line on which the item is declared, the name of the item, and the lines on which it is referenced. A small example is shown below.

CROSS-REFERENCE

Line	Name	Referenced on line(s)
------	------	-----------------------

32	ACOMMA		
30	DISPLAY-FILE-STATUS	54*	62
36	ERROR-WINDOW	59*	65*
26	FILE-STATUS	44	83
35	FLD1	44*	
34	FLD1-LITERAL	43	
27	FULL-FILE-STATUS	53	54*
42	MAIN-SCREEN	Unreferenced	
31	PRIMARY-FILE-STATUS	Owning group referenced	
28	PRIMARY-FILE-STATUS	Owning group referenced	
23	RST-REC	Owning group referenced	
33	SECONDARY-FILE-STATUS	Owning group referenced	
29	SECONDARY-FILE-STATUS	Owning group referenced	
21	TEST-RECORD75	77	79
37	WHITE-ON-BLUE	57	

testit.cbl Thu Sep 04 10:05:42 1998 ACUCOBOL-GT v4.0 Page:0002

CROSS-REFERENCE SECTIONS/PARAGRAPHS

Line	Name	Referenced on line(s)
69	MAIN-LOGIC	
51	TESTFIL-ERR-HANDLING	
49	TESTFIL-ERROR-HANDLING	50

If an item is not referenced directly, the listing indicates if the item's "owning group" is referenced, or if a "subordinate item" is referenced, or both ("owning group & subordinate item").

If the compiler detects that a Working Storage data item may be removed without affecting program functionality, it prints the word "Unreferenced" to the right of the name of the data item. This happens only if the data item is *not* a group item with subordinates that are referenced, and the data item is *not* subordinate to a group item that is referenced.

A line number followed by an asterisk indicates that the contents of the data item were modified at that line. Parameters passed in a CALL statement BY REFERENCE are always marked as modified in the listing, because it is possible that they were changed. An item that is not directly referenced but belongs to a referenced group, or is subordinate to an item that is referenced, is so indicated. When a full listing is requested with a cross-reference ("-Lfc"), line numbers are placed in columns 3 through 9 of the listing.

The first three columns of the listing indicate the copy files. Each line of a copy file is indicated, with the nest level number followed by a greater than symbol (>). The nest level number is "1" for a copy file included directly in the main source file, "2" for a copy file that is included in a level "1" copy file, "3" for a copy file included in a level "2" copy file, and so forth. An example of a copy file listing is shown under the "-Lw" option.

- Lf** Creates a full listing of the source program including the text of the COPY libraries.
- Li** Creates summary information about the program compiled. This is automatically set by any of the "-L" options.
- Ll** Sets the page length of the listing. This option must be followed by the number of lines per page (as the next separate argument). Specifying a negative number or zero ("0") for the argument produces a continuous listing without page headers or form feeds (a blank line separates different sections of the listing).

- Lo** This must be followed (as the next separate argument) by the name of the file to hold the listing. If this option is not specified, the listing is written to the standard output. This may be redirected to a file or a printer using the normal operating system commands. The *acurfap* syntax can be used to specify a file located on a system being served by AcuServer or AcuConnect. See **section 2.2.18.1, “Remote file name handling.”**
- Lp** This option directs the compiler to produce a “preprocessed” output file that can later be compiled to produce the same object code as the original source. The output includes the content of all COPY files and the results of all COPY REPLACING and REPLACE logic. The output does *not* include any comments or formatting from the original source. The output is sent to the standard output stream unless “-Lo” is used to specify a file. “-Lp” overrides all other listing options except “-Lo”.

This option is especially useful if your COPY files contain statements that other preprocessors don’t allow in COPY files, such as embedded SQL or CICS statements. To compile such a program, you first create the preprocessed file and then compile it. For example:

```
ccbl -Lpo finalsrc.cbl -sp copybooks origsrc.cbl
ccbl -Ga -o myprogram.acu finalsrc.cbl
```

Note: If the preprocessed code is subsequently compiled to include debugging symbols, the code displayed in the debugger is, of course, the preprocessed code, which does not include comments or formatting.

- Ls** Creates a symbol table at the end of the listing.
- Lw** Specifies a wide listing format. This option implies a full listing (thus “-Lw” is equivalent to “-Lfw”). The listing uses 101 characters to display copy book indicators, line numbers, relative line numbers, program addresses, sequence numbers, and the rest of each source line.

For multiple-line sentences, the address of the first verb of each line is shown.

The first three columns indicate copy files. Each line of a copy file is indicated, with the nest level number followed by a greater than symbol (>). The nest level number is “1” for a copy file included directly in the main source file, “2” for a copy file that is included in a level “1” copy file, “3” for a copy file included in a level “2” copy file, and so forth. Following is a portion of a listing that shows a copy file:

```

      50                main-logic.
      51      000002      copy "copyfile".
1> 52                *
1> 53                display "COPY file".
1> 54                *
      55      00000E      display window
      56                  size 40 lines 10
      57                  line 10 col 10 boxed
      58                  pop-up area is error-window.
```

- Lx** Creates extended statistics at the end of the listing.

2.2.4 Internal Tables Options

The ACUCOBOL-GT compiler makes use of several internal tables for storing information about the COBOL program it is compiling. Most of these tables are dynamically resized as needed during compilation. A few of them are allocated a fixed amount of space. The amount of space allocated to the *fixed size* tables is enough for the vast majority of programs, but occasionally a program will require more space than is allocated by default. When this occurs, the compilation fails and a “*** %s overflow ***” error message is output. (See [Appendix D, “List of Errors.”](#) for a detailed explanation of the

error.) In such cases, the user can increase the amount of space allocated to that table by including the appropriate “-T” compilation switch on the compiler command line.

The switches are:

- Td ##### Identifier and statement table--sets the maximum number of items in each statement. The default value is 4096.
- Te ### Subscript statement table--sets the maximum size for OCCURS statements. The default value is 256.

Note: The compiler error message always suggests that you double the size of the default value. However, doubling the size is not always the best approach. Ideally, the size of the table should not be substantially larger than what is needed to compile the program.

2.2.5 Compatibility Options

ACUCOBOL-GT has several options that aid in converting programs written for other COBOL environments. See **Section 2.4, “Compatibility Modes,”** for more information. By default, ACUCOBOL-GT runs in the VAX COBOL compatibility mode.

- Ca** This option causes simple ACCEPT and DISPLAY statements to be treated in accordance with ANSI semantics. Specifying this option is the same as specifying FROM CONSOLE for all simple ACCEPT statements and UPON CONSOLE for all simple DISPLAY statements. You can control this behavior for individual ACCEPT or DISPLAY statements by specifying an explicit FROM/UPON phrase. For more information, see the discussion in Book 2, **Chapter 2, section 2.3.1, “ANSI ACCEPT and DISPLAY Verbs”**

Prior to Version 2.1, “-Ca” was a synonym for the compile-time option “-Va”. If you have an obsolete reference to “-Ca” in any of your compile scripts, or in your CBLFLAGS environment setting, make sure that you change this to “-Va”.

- Cb** This option causes all sequential files that are not explicitly specified as LINE or BINARY sequential to be treated as BINARY sequential. Normally the file type is determined by a long sequence of rules (see Book 3, *Reference Manual*, **Section 5.1.7, “File Types”**). Files specified as print files are treated as LINE sequential even when this option is selected.

- Ce** This option allows you to specify a default source name extension that is used for both the main source file and its COPY files. When it is specified, any source or COPY file name that does not explicitly specify an extension has the default extension appended to it. When the option is specified in RM/COBOL compatibility mode, the default extension is “.CBL”. Otherwise, the default is “.COB”.

You may specify an alternate default by naming it after an “=” sign in the “-Ce” option. For example, the option “-Ce=CPY” would cause the default extension to be “.CPY”.

ACUCOBOL-GT automatically adjusts the extension for upper or lower case to match the case of the name it is being appended to, then searches for the COPY file. If this first search fails, ACUCOBOL-GT matches the case of the extension specified in the “-Ce” compiler option to perform a second search for the COPY file.
- Cf** This option forces all indexed, relative, and binary sequential files to be given fixed-length records. Normally the record type is determined by a sequence of rules (see **Section 5.1.7, “File Types”** in Book 3, *Reference Manual*). This flag is supplied to enhance compatibility with pre-85 compilers and earlier versions of ACUCOBOL-GT.
- Cg** Turns off the conditional compiling features. In version 8.1.0 (ECN 3607) conditional compilation was added to ACUCOBOL-GT. Conditional compiling statements begin with the \$ sign. In prior versions, lines of code that began with the \$ sign were treated as comments. Unfortunately, some COBOL programs use a type of conditional compilation, which now fail to compile. Turning off conditional compiling should rectify these situations. See **Section 2.2.15, “Conditional Compilation Options”** for information on conditional compiling.
- Ci** Causes the compiler to be compatible with ICOBOL for certain COBOL constructs. References in this manual to VAX COBOL compatibility also apply to ICOBOL compatibility mode unless otherwise stated.

-Ck Causes the compiler to generate key numbers for indexed file alternate keys in a different order. Key numbers are used internally by the compiler and runtime to identify a key. Normally, you do not need to know the key number for a particular key, or care about the order in which they are generated. By default, the compiler sorts the alternate keys in record order before assigning key numbers. This ensures that the keys are given the same number regardless of the ordering in the file's SELECT. Specifying "-Ck" causes the compiler to assign key numbers in the SELECT order.

For most applications, the order in which key numbers are assigned is irrelevant. This option provides compatibility with some other COBOL compilers. This could be helpful when you are sharing data files between ACUCOBOL-GT based programs and programs compiled with these other systems.

-Cm This option causes the compiler to be compatible with IBM/COBOL for the ASSIGN phrase of a SELECT statement. The syntax for the IBM/COBOL ASSIGN phrase is:

```
ASSIGN TO <filename-1> [<filename-2>  
<filename-3> ...]
```

where <filename-1>, <filename-2>, and <filename-3> are file identifiers.

Compiling with the "-Cm" option allows the IBM/COBOL ASSIGN phrase to ignore <filename-2> and <filename-3>. If <filename-1> contains any "-" characters, only the characters following the last "-" will constitute the file name assignment for the particular SELECT command. For example:

```
SELECT TESTFILE ASSIGN TO SYS000-AS-TEST1
```

results in a file called "TEST1".

All of the existing ACUCOBOL-GT SELECT phrases and options can be used with the IBM/COBOL ASSIGN phrase.

-Cp This sets the compiler to its HP COBOL compatibility mode. In this mode, it accepts features of HP COBOL that are not otherwise accepted. See Chapter 4 in *Transitioning to ACUCOBOL-GT* for more information.

- Cr This option sets the compiler to its RM COBOL compatibility mode. In this mode, it accepts features of RM COBOL that are not otherwise accepted. See Chapter 2 in *Transitioning to ACUCOBOL-GT* for more information.
- Cv This option sets the compiler to its IBM DOS/VS compatibility mode. In this mode, it accepts features of IBM DOS/VS COBOL that are not otherwise accepted. See Chapter 5 in *Transitioning to ACUCOBOL-GT* for more information.

Since there are slight differences between IBM COBOL versions, “-Cv” also takes the following optional arguments:

“-Cv=OSVS” specifies OSVS compatibility.

“-Cv=VSC2” specifies VSC2 compatibility.

“-Cv” by itself defaults to OSVS mode. The two modes are very similar, except that in VSC2 compatibility mode, the following words are not reserved:

CURRENT-DATE

EXAMINE

TIME-OF-DAY

TRANSFORM

Note that CURRENT-DATE is a valid function in any compatibility mode.

- C3 Causes ACUCOBOL-GT to abide by the rules used by Version 1.3. Later versions changed the way ACUCOBOL-GT treats a few special cases. If you use this flag, ACUCOBOL-GT will treat these cases in the same way it did under Version 1.3. This is useful if you are compiling programs written for Version 1.3 and you do not want to modify the programs. Appendix C in Book 4, *Appendices*, contains details of which features are affected by this option.

Specifying this option does not prohibit the use of later features, but merely ensures that the changed features behave in the original fashion. Compare with the “-Z3” option below.

- C4 Similar to -C3, this option causes the compiler to generate code according to the rules used by Version 1.4.

- C5 This option causes the compiler to maintain source compatibility with the Version 1.5 compiler.
- C20 Causes the compiler to maintain source compatibility with the Version 2.0 compiler. Chart verbs are allowed.
- C21 Causes the compiler to maintain source compatibility with the Version 2.1 compiler
- C22 Causes the compiler to maintain source compatibility with the Version 2.2 compiler.
- C23 Causes the compiler to maintain source compatibility with the Version 2.3 compiler
- C24 Causes the compiler to maintain source compatibility with the Version 2.4 compiler
- C30 Causes the compiler to maintain source compatibility with the Version 3.0 compiler
- C31 Causes the compiler to maintain source compatibility with the Version 3.1 compiler
- C32 Causes the compiler to maintain source compatibility with the Version 3.2 compiler
- C40 Causes the compiler to maintain source compatibility with the Version 4.0 compiler
- C41 Causes the compiler to maintain source compatibility with the Version 4.1 compiler
- C42 Causes the compiler to maintain source compatibility with the Version 4.2 compiler
- C43 Causes the compiler to maintain source compatibility with the Version 4.3 compiler
- C50 Causes the compiler to maintain source compatibility with the Version 5.0 compiler
- C51 Causes the compiler to maintain source compatibility with the Version 5.1 compiler
- C52 Causes the compiler to maintain source compatibility with the Version 5.2 compiler

- C60 Causes the compiler to maintain source compatibility with the Version 6.0 compiler
- C61 Causes the compiler to maintain source compatibility with the Version 6.1 compiler
- C62 Causes the compiler to maintain source compatibility with the Version 6.2 compiler
- C70 Causes the compiler to maintain source compatibility with the Version 7.0 compiler
- C71 Causes the compiler to maintain source compatibility with the Version 7.1 compiler
- C72 Causes the compiler to maintain source compatibility with the Version 7.2 compiler
- C73 Causes the compiler to maintain source compatibility with the Version 7.3 compiler
- C80 Causes the compiler to maintain source compatibility with the Version 8.0 compiler
- C81 Causes the compiler to maintain source compatibility with the Version 8.1 compiler

2.2.6 Interoperability Options

- javaclass** Generates a “.java” file in addition to a “.acu” file. The “.java” file has the same prefix as the “.acu” file and is placed in the same directory. This “.java” file is a Java class that calls the COBOL program being compiled. Refer to *A Guide to Interoperating with ACUCOBOL-GT*, section 2.2 for more information on calling COBOL from Java.
- javamain** Generates a “.java” file in addition to a “.acu” file. The “.java” file has the same prefix as the “.acu” file and is placed in the same directory. This “.java” file is a Java class with a main method added. This class calls the COBOL program that is being compiled. Refer to *A Guide to Interoperating with ACUCOBOL-GT*, section 2.2 for more information on calling COBOL from Java.

--netexe

Generates a .NET executable file for command-line execution. The name of the executable is the name of the program followed by “.exe.” All valid ACUCOBOL-GT command-line options can be specified with the executable, as well as any of the following Linkage Section parameters:

-int:

-string:

-uint:

-short:

-ushort:

-float:

-double:

-long:

-ulong:

-byte:

See *A Guide to Interoperating with ACUCOBOL-GT*, section 5.4 for more details.

Note: On Windows 7, compiling with this option may fail due to a missing **al.exe**, which is not included as a utility with Windows 7. To fix this, download and install the Microsoft .Net framework SDK and then try compiling with this option.

--netdll

Generates a .NET dynamic link library (DLL) that gives .NET assemblies—both executables and DLLs—a programmatic interface to your COBOL program. All COBOL entry points are exposed as .NET methods along with ACUCOBOL-GT runtime properties and methods. This allows .NET programmers to set ACUCOBOL-GT command options and call runtime interfaces from their .NET assembly. Refer to *A Guide to Interoperating with ACUCOBOL-GT*, section 5.4 for more details.

Note: On Windows 7, compiling with this option may fail due to a missing **al.exe**, which is not included as a utility with Windows 7. To fix this, download and install the Microsoft .Net framework SDK and then try compiling with this option.

2.2.7 File Options

Some file options (-Fa, -Fe, -Fx, and -F4) are used to generate XFD files (data dictionaries) that are used with Acu4GL, AcuXML, the **alfred** record editor, and with international character mapping for AcuXDBC and AcuServer. Other file options (-Fl, -Fs, and -Ft) can simplify the addition of transaction management facilities to existing programs that use the Vision file system.

For more details regarding transaction management, see **section 5.1, “Transaction Management.”** For more details about international character mapping, see **section 5.4, “International Character Handling.”**

-Fa This option tells the compiler to build data dictionaries (XFD files) for every indexed, relative, and sequential data file in the FDs of the program. It is the only option that builds XFDs for relative and sequential files. This option is also used for AcuXML file translation and for international character mapping. See also the “-Fo” compile-time option, which specifies the directory in which the data dictionaries are placed.

The “-Fa” option generates the most current format for the XFD files (Version 6). (XFD versioning is distinct from Vision versioning.) Any version of Vision will work with any version of XFD, as long as the XFD version can store all the necessary precision of the Vision file. (For example, if you are using Vision Version 5 and you want to use a larger record size than what can be stored in 5 digits, then you will need to use Version 5 XFDs.)

An attempt to use the “-Fa” option with a backwards-compatibility switch results in a warning and the XFDs for RELATIVE and SEQUENTIAL files are not created. An attempt to use this option for records containing duplicate element names also results in a warning; however, if you are creating XFDs for AcuXML, you may disregard this warning. XFDs will be generated, and XML supports duplicate names.

-Fc This option causes the field names in generated XFD files to match exactly the source of the COBOL program that generated them.

- Fe** Causes XFD files to be generated in standard flat text format rather than XML format. Acu4GL, AcuXDBC, and **alfred** can all read XFD files in XML format. This option must be used in conjunction with the “-Fx” or “-Fa” options. The **CSXML** library routine can be used to parse the XML files if desired. This option will not work in combination with “-F4” or “-F3”. Version 4 and 3 XFD files cannot be generated in XML format.
- Fl** Enables single locking rules rather than multiple locking rules as the lock mode default. Normally, “WITH ROLLBACK” causes multiple locking rules to be in effect for a file. When “-Fl” is used, the “WITH ROLLBACK” clause does not affect whether single or multiple record locking rules are followed. Single locking becomes the default. You may enable multiple locking either by specifying “WITH LOCK ON MULTIPLE RECORDS” in a file’s SELECT statement or by using “APPLY LOCK-HOLDING ON *file*” in the I-O CONTROL paragraph.
- Fm** Causes LOCK MODE IS MANUAL to be applied when the LOCK clause is omitted from the SELECT statement in the FILE-CONTROL paragraph.

-Fn Specifies file locking as the default behavior for files that do not have locking or sharing already specified or implied from within the program. This option can improve batch processing times and make file access more restrictive for programs that do not specify any form of file locking.

This option implies a lock on any OPEN statement contained in the program. The implied lock is based on the OPEN format as follows:

OPEN	Implied Lock
OPEN INPUT	ALLOWING READERS
OPEN OUTPUT	ALLOWING NO OTHERS
OPEN EXTEND	ALLOWING NO OTHERS
OPEN I-O	ALLOWING NO OTHERS

If the following file locking or sharing methods are applied to a file then the “-Fn” has no effect:

- a. SELECT contains a LOCK phrase. The specified or implied behavior of the LOCK phrase is used.
- b. The OPEN statement contains the EXCLUSIVE, LOCK, MASS-UPDATE, BULK-ADDITION, or ALLOWING phrases. The sharing attributes that these phrases set are used.

See Book 3, ACUCOBOL-GT Reference Manual, **OPEN Statement** for details on applying file locks and sharing.

- Fo** This option must be followed (as the next separate argument) by the directory that will hold the data dictionary files generated by the compiler when you use the “-Fx” option.
- Type a space after the option and then give the name of the chosen directory. If this option is not used, the data dictionaries are placed into the current directory.
- For example, to cause the dictionaries to be stored in the directory “/usr/inventory/dictionaries” you would enter:
- ```
-Fo /usr/inventory/dictionaries
```
- The *acurfap* syntax can be used to specify a location on a system being served by AcuServer or AcuConnect. For information about *acurfap* syntax, see [section 2.2.18.1, “Remote file name handling.”](#)
- Fp** Causes all files, except sort files, to be treated as if the OPTIONAL phrase is specified in the SELECT statement.
- Fs** Causes an implied START TRANSACTION verb before the first OPEN, CLOSE, WRITE, REWRITE, or DELETE and after each COMMIT or ROLLBACK. In effect, every file operation is part of a transaction. If this option is enabled, and the compiler encounters a START TRANSACTION verb, it reports a warning and does not generate any code for the START TRANSACTION. The “-Fs” option provides an alternate way to program transactions and is often useful when you are converting from other COBOL or SQL implementations. Transaction management is discussed in detail in [Section 5.1](#).

- Ft** Causes implied transactions for every OPEN, CLOSE, WRITE, REWRITE, or DELETE that is not part of an explicit transaction. Single file operations that are not part of a transaction are preceded by an implied START TRANSACTION and followed by an implied COMMIT. This option makes converting existing applications to a transaction system easier. Note that unlike most COMMITs, which unlock all of the file's currently locked records, the implied COMMIT does not unlock any records. See **Section 5.1.4, "Extended Locking Rules"** for details regarding file locking and the handling of implied transactions.
- Fx** This option directs the compiler to build Version 6 data dictionaries (XFD files) for every indexed data file in the FDs of the program. If you need an older version of XFD files, specify the "-F3", "-F4" or "-F5" option instead of "-Fx". If you use relative, sequential, or XML data files, use the "-Fa" option instead. Use the "-Fo" option to specify the directory in which the data dictionaries should be placed.

---

**Note:** XFDs are required if you plan to use any Acu4GL interface, AcuXDBC, or AcuXML; these interfaces cannot operate without data dictionaries. XFDs are also required for international character mapping with AcuServer and they provide useful information to the **alfred** record editor. XFD files have three different formats: Version 4, Version 5 and Version 6. Database technologies in the **extend6** Interoperability Series, including AcuXML, require Version 5 XFDs. Version 4 XFDs are required for previous versions of Acu4GL, **alfred**, and AcuODBC (Version 5.x and earlier).

---

- F4** This option tells the compiler to build Version 4 data dictionaries (XFD files) for every indexed data file in the FDs of the program. This older version of the XFD files is compatible with Acu4GL, AcuODBC, and **alfred** Version 5.x and earlier. To build Version 4 XFDs for every indexed, relative, and sequential data file in your FDs, combine “-F4” with “-Fa”, as in “ccbl -F4 -Fa”.
- Fxa** Supported but obsolete. It produces the same result as “-Fa”.
- fileAssign=** This option allows you to specify how to assign a filename when neither DYNAMIC or EXTERNAL is part of the ASSIGN clause of the SELECT statement. The “=” must be followed by the keyword DYNAMIC or EXTERNAL. For example, “--fileAssign=external” causes the compiler to treat SELECT statements that do not specify DYNAMIC or EXTERNAL in the ASSIGN clause, as if EXTERNAL were specified.
- fileIdSize=#** Specifies the size, in bytes, of file handles passed to COBOL programs. “#” is replaced by the value 2, 4, or 8 to specify the number of bytes in the passed integer. A four byte native integer is passed by default. A two byte native integer is used in HP COBOL compatibility mode (“-Cp”).

## 2.2.8 Source Options

The source options modify the way the compiler treats the physical source files.

- S#    Specifying a digit with “-S” causes ACUCOBOL-GT to use alternate tab stops for its source files. By default, ACUCOBOL-GT sets tabs every eight columns. When this option is used, tabs will be set every “#” columns apart where “#” is the number specified. For example, “-S4” will set tab stops at every fourth column. Tab stops always start in column 1.
  
- Sa    This causes the compiler to assume that the input source is in the standard ANSI source format. For details on source formats, see **section 2.5, “Source Formats.”** This option can be specified in a compiler directive. See **section 2.2.15, “Conditional Compilation Options.”**
  
- Sc    This option must be followed by one of the following encoding scheme identifiers: BIG5, DBC, EUC, GB, KSC, or SJC. When used, this option causes the compiler to assume that the input source is encoded using the specified encoding scheme. This allows the compiler to recognize double-byte characters used in string literals and comments. The table below shows the encoding scheme identifiers, the code system to which each setting refers, and some examples of operating systems to which the particular encoding scheme applies:

| Setting | Code System                                                                 | Op. System Example         |
|---------|-----------------------------------------------------------------------------|----------------------------|
| BIG5    | Big Five (Taiwan)                                                           | Chinese DOS, Windows       |
| DBC     | ACUCOBOL-GT<br>Generic<br>Double-byte<br>Coding Scheme                      | other double-byte machines |
| EUC     | Extended UNIX                                                               | Most UNIX machines         |
| GB      | Code of Chinese<br>Graphic Character<br>Set (People’s<br>Republic of China) | Chinese DOS, Windows       |

| Setting | Code System                                   | Op. System Example                 |
|---------|-----------------------------------------------|------------------------------------|
| KSC     | Korean Character Standard                     | Korean DOS                         |
| SJC     | Shift JIS Code (Japanese Industrial Standard) | DOS/V, Windows, some UNIX machines |

- Sd**     Setting this option causes debugging lines marked with “D” in the indicator area to be treated as normal source lines instead of comment lines. This is equivalent to supplying the phrase “WITH DEBUGGING MODE” in the SOURCE-COMPUTER paragraph.
  
- Si**     This option causes the compiler to include source lines according to a pattern in the Identification Area of the source line. The next (separate) command-line argument is treated as this pattern. (The pattern is case sensitive; enclose it in double quotes on systems such as VMS where you need to preserve its case.) For details on source code control and the use of this option, see **section 2.7, “Source Code Control.”**
  
- Sp**     With this option you can specify a series of directories to be searched when the compiler is looking for COPY libraries. This option is followed (as the next separate argument) by the set of directories to search. See **section 2.6, “COPY Libraries.”**  
  
The *acurfap* syntax can be used to specify a location on a system being served by AcuServer or AcuConnect. For information about the *acurfap* syntax, see **section 2.2.18.1, “Remote file name handling.”**
  
- Sr**     This option causes ACUCOBOL-GT to use RM/COBOL-style tab stops. These tabs are set four columns apart starting in column 8 and ending at column 72.

- St This forces the compiler to use the terminal source format. For details, see **section 2.5, “Source Formats.”** This option can be specified in a compiler directive. See **section 2.2.15, “Conditional Compilation Options.”**
- Sx This is similar to the “-Si” option above, except that the next argument is treated as a pattern to use to *exclude* source lines. For details on source code control and the use of this option, see **section 2.7, “Source Code Control.”**

## 2.2.9 Reserved Word Options

ACUCOBOL-GT provides the ability to suppress certain sets of reserved words. When suppressed, these words are treated as user-defined words instead. This ability is provided to aid in converting programs written for other COBOL environments. Reserved words that have been added to the 1974 standard are divided into the groups identified below. By specifying the corresponding option, you cause that group of words to be treated as user-defined words.

**Section B.2** in Book 4, *Appendices*, includes a list of reserved words and shows which group each word belongs to. Some words belong to more than one group. Every group a word belongs to must be suppressed for the word to be treated as a user-defined word.

- R2 Suppresses words reserved by the 2002 COBOL standard.
- R8 Suppresses words reserved by the 1985 COBOL standard.
- Ra Suppresses words reserved by ACUCOBOL-GT that are required for its unique extensions.

- Rc** Allows you to change a reserved word. This option must be followed by two separate arguments. The first is the reserved word you want to change. The second is the word that you want to use instead. For example, “-Rc TITLE NAME” will allow you to use “TITLE” as a user-defined word and will cause the word “NAME” to be treated as the reserved word “TITLE”. You may not specify a word that is already reserved as the new reserved word. This option may be repeated to transform multiple reserved words.
- Ri** Suppresses words reserved by COBOL that are not in the 1985 standard.
- Rn** Allows you to make a reserved word a synonym for another reserved word. This option must be followed by two separate arguments. The first is the reserved word for which you want a synonym. The second is the word that functions as the synonym. For example: “-Rn COMP COMP-5” causes “COMP-5” to be treated the same as the reserved word “COMP”. This option may be repeated to make multiple synonyms.
- Rr** Suppresses words reserved by RM/COBOL that are not in the 1985 standard.
- Rs** Suppresses words added to ACUCOBOL-GT in order to support the Screen Section.
- Rv** Suppresses words reserved by VAX COBOL that are not in the 1985 standard.

- Rw** This option allows you to suppress a particular reserved word. The option must be followed (as the next separate argument) by the reserved word you want to suppress. This option may be repeated to suppress multiple reserved words.

This option also allows you to suppress some non-reserved words, such as control names (e.g., “entry-field”, “label”) or property names (e.g., “max-text”, “bitmap-number”). If you tell the compiler to suppress a non-reserved word, it issues the following warning:

```
Unknown reserved word: non-reserved word
```

- Rx** This option tells the compiler to ignore a particular word. The option must be followed (as the next separate argument) by the word you want the compiler to ignore. This option may be repeated to ignore multiple words.

## 2.2.10 Data Storage Options

For more information about the format of a data item in memory, see **Section 5.7.1.8** in Book 3, *Reference Manual*.

- Da** Allows you to specify the data alignment modulus for level 01 and level 77 data items. Normally, level 01 and level 77 data items are aligned on a 4-byte boundary (modulus 4). This is optimal for 32-bit architectures. You can specify an alternate alignment boundary by following this option with the desired modulus. This should be specified as a single digit that immediately follows the “-Da” as part of the same argument. For example, “-Da8” specifies that data should be aligned on eight-byte boundaries, which can provide improved performance on a 64-bit machine.
- Db** Causes COMPUTATIONAL data items to be treated as if they were declared as BINARY data items. This is the default when you are using VAX COBOL compatibility mode.



- De** Causes the compiler to generate MOVE code for all LINKAGE data items that works regardless of the alignment of the data item. The move is, however, less efficient. Please note that this option should be used only if the linkage is given values using the SET ADDRESS OF verb.
- Dca** This selects the ACUCOBOL-GT storage convention. It is the default setting. This convention is also compatible with data produced by RM/COBOL (*not* RM/COBOL-85) and previous versions of ACUCOBOL-GT. It also produces slightly faster code.
- Dcb** This selects the MBP COBOL sign storage convention. See Book 3, [section 5.7.1.8](#) for the specifics of sign storage.
- Note that the MBP COBOL sign storage convention for USAGE DISPLAY directly conflicts with that used by IBM COBOL and some other COBOLs. As a result, signed USAGE DISPLAY items in the MBP format are correctly understood only when the program is compiled with “-Dcb”. This is unlike the other sign conventions in which the runtime can usually extract the correct value even when a mismatched sign convention is specified at compile time.
- Also note that MBP COBOL does not have the COMP-2 storage type. The convention that ACUCOBOL-GT implements (Positive: x'0C'; Negative: x'0D') was chosen because MBP COBOL most closely matches the sign storage of other COBOLs that use that convention.
- Dci** This selects the IBM storage convention. It is compatible with IBM COBOL, as well as with several others including RM/COBOL-85. It is also compatible with the X/Open COBOL standard.
- Dcm** This selects the Micro Focus storage convention. It is compatible with Micro Focus COBOL when the Micro Focus “ASCII” sign-storage option is used (this is the Micro Focus default).

- Dcn** Causes a different numeric format to be used. The format is the same as the one used when the “-Dci” option is used, except that positive COMP-3 items use “x0B” as the positive sign value instead of “x0C”. This option is compatible with NCR COBOL.
- Dcr** This selects the Realia sign storage convention. Sign information for S9(*n*) variables is stored using the conventions for Realia COBOL, and their conversion to binary decimal is the same as that performed by the Realia compiler.
- Dcv** This creates numeric sign formats that are compatible with VAX COBOL. These are identical to the IBM formats, except that unsigned COMP-3 fields place “x0C” in the sign position, instead of “x0F”.

The ANSI definition of COBOL does not state how signs should be stored in numeric fields (except for the case of SIGN IS SEPARATE). As a result, different COBOL vendors use different conventions. By using the options “-Dca”, “-Dci”, “-Dcm”, “-Dcn”, or “-Dcv”, you may select alternate sign-storage conventions. Doing so is useful in the following cases:

**If you need to export data** to another COBOL system and need to match its sign-storage convention.

**If you are importing data** from another COBOL system, and that data contains key fields with signed data. Keys are treated alphanumerically, so if you use the incorrect sign-storage convention, ACUCOBOL-GT will not find a matching key when it is doing a READ.

The storage-convention affects how data appears in USAGE DISPLAY, COMP-2 and COMP-3 data types. For additional information and tables, see Book 3, *Reference Manual*, **section 5.7.1.8**, “USAGE clause.”

**-Dd31** This option supports data items with up to 31-digits or 16 bytes. When this option is in effect, you may use as many as 31 “X” or “9” symbols in a PIC, instead of the usual 18. The maximum number of bytes in a COMP-X or COMP-N data item, whose picture contains only “X” symbols, is 16, instead of the usual 8. Intermediate results are calculated to 33 digits instead of the usual 20.

**-Df** This option changes the way the compiler treats data items declared as COMP-1 and COMP-2.

Some compilers use COMP-1 and COMP-2 to specify single and double precision floating point data items. However, ACUCOBOL-GT assigns a different meaning to COMP-1 and COMP-2 and uses FLOAT and DOUBLE to specify floating point data items.

When the “-Df” option is used, the compiler treats data items declared as COMP-1 as if they were declared FLOAT and data items declared as COMP-2 as if they were declared DOUBLE. With the “-Df” option, you have the following correspondence:

```
COMP-1 FLOAT single precision
COMP-2 DOUBLE double precision
```

The “-Df” option makes it easier to compile code originally written for another compiler--one that used COMP-1 and COMP-2 to specify floating point data items. The “-Df” option lets you compile such code without having to change COMP-1 and COMP-2 to FLOAT and DOUBLE.

**-Di** This option causes the compiler to initialize Working-Storage. Normally, the compiler will initialize all data items to spaces or the value specified with the “-Dv” option, except for those items given a VALUE clause. If this option is specified, data items are initialized according to their type:

**Alphabetic, alphanumeric, alphanumeric edited, and numeric edited items** are initialized to spaces.

**Numeric items** are initialized to zero.

**Pointer items** are initialized to null.

**Index items** are initialized to the value 1.

Automatic initialization applies only to Working-Storage and does not apply to any item that (a) is given a VALUE clause, (b) is EXTERNAL, or (c) is subordinate to a REDEFINES phrase.

**-DI** Allows you to limit the maximum alignment modulus that will be used for SYNCHRONIZED data items. Normally, a synchronized data item is aligned on a 2-, 4-, or 8-byte boundary depending on its type. This option allows you to specify an upper bound to the modulus used. This is specified as a single digit that immediately follows the “-DI” as part of the same argument. For example, “-DI4” specifies that the maximum synchronization boundary is a 4-byte boundary. If you want to make programs that are compliant with the 88/ Open COBOL specification, you should specify “-DI4”. This option can be specified in a compiler directive. See **section 2.2.15, “Conditional Compilation Options.”**

**-Dm** Causes any data item whose underlying type is binary to be stored in the minimum number of bytes needed to hold it. Normally, binary types are stored in two, four, or eight bytes. This option allows storage in any number of bytes ranging from one to eight. The exact number of bytes used for a particular data item is described under the USAGE clause in the ACUCOBOL-GT *Reference Manual*.

- Dq** Causes the QUOTE literal to be treated as an apostrophe, or single quotation mark, rather than as a double quotation mark (“”). One exception to this is the HP e3000 TRANSFORM verb, in which QUOTE is always treated as a double quotation mark.
- Ds** This causes USAGE DISPLAY numeric items with no SIGN clause to be treated as if they were described with the SIGN IS TRAILING SEPARATE clause. Several versions of RM/COBOL behave this way (all versions before 2.0, and some versions afterward).
- Dv** Allows you to specify the default byte (initial value) used to initialize any data item not otherwise initialized when the program is loaded. The option must be followed by an equals sign (“=”) and the decimal value of the byte to use (for all current platforms, this is the ASCII value of the desired character). For example, to fill memory with the NULL character, use “-Dv=0”. To fill memory with the ASCII space character, use “-Dv=32”.

The default value is the space character.

- Dw** Lets you modify the definition of certain data types. These data types are generally dependent on the host machine's native word size. You may use this option to:
1. Maintain compatibility with COBOL source code written specifically for 32-bit machines (for example, to set the size of USAGE POINTER data items to 4 bytes).
  2. Make it easier to match a C structure for a particular machine.
  3. Optimize your data storage for a particular class of machines.

“-Dw” selects the maximum word size of the set of machines that you expect to run on. You follow this option with the maximum word size you desire, expressed as the number of bits per word. Currently, the legal forms are “-Dw32” and “-Dw64”.

This option determines the *maximum* word size, not the exact word size.

The “-Dw” option affects the size of the following data types:

SIGNED-SHORT  
SIGNED-INT  
SIGNED-LONG  
POINTER  
UNSIGNED-SHORT  
UNSIGNED-INT  
UNSIGNED-LONG

It also affects the size of the RETURN-CODE special register.

Selecting a maximum word size does not inhibit the portability of your code. Instead, it limits the size of certain data items. If you attempt to use a data item that is too small for a particular machine, you may lose precision. For example, USAGE POINTER data items are stored in 4 bytes if you use “-Dw32”. If you attempt to run a program that stores an address in 4 bytes on a 64-bit machine, you may lose some of the address. While the program will technically run, the results may not be useful.

For maximum portability, you should use “-Dw64”. This will allow your code to run on all machines that run ACUCOBOL-GT. However, if your program will call Windows DLLs, you should use “-Dw32”. For strict compatibility with ACUCOBOL-85 Version 2.2 or earlier, you should use “-Dw32”. For programs prior to Version 2.3, the only real effect of shifting from “-Dw32” to “-Dw64” is that USAGE POINTER data types expand from 4 bytes to 8 bytes. If your program does not depend on the size of POINTER data items, then you should be able to use “-Dw64” with no harm.

If you use any of the “-C” compile options that establish source compatibility with ACUCOBOL-85 Version 2.2 or earlier (e.g. “-C21”), then the default setting is “-Dw32”. If you do not use these options, then the default setting is “-Dw64”. You may override the default by using the “-Dw” option. For example, to compile for source compatibility with Version 2.1, but to set the target architecture to 64 bits, you would use “-C21 -Dw64”. In order to use “-Dw64”, you must use a Version 2.3 or later runtime.

See the related information on USAGE types in Book 3, *Reference Manual*, section 5.7.1.8. See also the information on RETURN-CODE and support for 64-bit architectures in section C.4 in Book 4, *Appendices*.

- Dy** Specifies that all data items whose underlying representation is binary should be treated as if they were described as SYNCHRONIZED. This option is *not* recommended unless you have a particular need for it. ACUCOBOL-GT is optimized for non-synchronized handling of binary data, so synchronization will usually not have beneficial results. Note, however, that you must specify this option if you want to make programs that are compliant with the 88/Open COBOL specification.
- Dz** This option causes the compiler to modify its size checking rules for numeric items. Instead of computing size error by examining the number of 9's in an item's picture, the compiler computes size error by examining the actual storage for that item. For example, normally a PIC 99 BINARY data item cannot hold a number larger than 99, although the storage for the item can hold a value up to 255. When "-Dz" is used, ACUCOBOL-GT will not cause a size error until a value greater than 255 is moved to this item. This option also affects truncation in MOVE statements and implied moves.
- There are four truncation options in all. See the chart in **Section 2.2.10.1** for description and comparison information on these options.
- D1** Causes any data item whose underlying type is binary to be stored in one byte if that data item has only one or two digits. Normally, such a data item would be stored in two bytes.
- D2** Causes COMPUTATIONAL data items to be treated as if they were declared as COMPUTATIONAL-2. This is the default when you are using RM/COBOL compatibility mode.
- D5** Causes data items declared as BINARY to be treated as if they were declared as COMPUTATIONAL-5. This causes the values to be stored in the host machine's native byte-ordering instead of the machine-independent byte-ordering normally used. This option is usually advised when converting Micro Focus applications on UNIX/Linux hosts. This option should be used with caution, however, because it can lead to programs that are not portable.



- D6** Causes unsigned data items declared as PACKED-DECIMAL to be treated as if they were declared as COMPUTATIONAL-6. This saves one-half of a byte because the compiler will not generate any storage for the sign.
- D7** Allows you to match one of the binary storage conventions used by Micro Focus COBOL. That convention is identical to the ACUCOBOL-GT “-Dm” convention, except that a PIC 9(7) data item (unsigned) is stored in 3 bytes instead of 4 and a PIC 9(12) data item (unsigned) is stored in 5 bytes instead of 6. When you use this option, the size of a binary item is determined as follows (the value in the table is the number of bytes occupied by the data item):

| Number of 9's in PIC | Signed Storage | Unsigned Storage |
|----------------------|----------------|------------------|
| 1 - 2                | 1              | 1                |
| 3 - 4                | 2              | 2                |
| 5 - 6                | 3              | 3                |
| 7                    | 4              | 3                |
| 8 - 9                | 4              | 4                |
| 10 - 11              | 5              | 5                |
| 12                   | 6              | 5                |
| 13 - 14              | 6              | 6                |
| 15 - 16              | 7              | 7                |
| 17 - 18              | 8              | 8                |

**--FpRounding**

Simulates the behavior of other COBOL systems with regard to implied rounding when floating point is used in a math statement.

This case-insensitive option is followed by an equals sign (“=”) and one of the following:

**OSVS** means that any math statement that contains a floating-point data item as a sending item has “ROUNDED” implied for every receiving item.

**VSC2** means that any math statement that contains a floating-point data item as either a sending item or a receiving item has “ROUNDED” implied for every receiving item.

If “--FpRounding” is not specified, rounding is not automatically implied.

For example, consider the following program fragment:

```
77 INT-1 PIC 99.
77 FLOAT-1 USAGE FLOAT.
MOVE 1.6 TO FLOAT-1
COMPUTE INT-1 = 3 * FLOAT-1.
```

Without this compiler option, the value moved to INT-1 is “4” (4.8 truncated). If either “--FpRounding=OSVS” or “--FpRounding=VSC2” is specified, the value moved would be “5” instead (4.8 rounded up).

Alternatively, the following example:

```
77 INT-1 PIC 99.
77 DEC-1 PIC 99V99.
77 FLOAT-1 USAGE FLOAT.
MOVE 1.6 TO DEC-1
COMPUTE FLOAT-1, INT-1 = 3 * DEC-1
```

would move “5” to INT-1 only if “--FpRounding=VSC2” is used (because the only floating item is a receiving item).

**--lastWSDataSeg=#** “#” is an integer between “1” and “32”, inclusive.

This option sets the data segment number that will be the last one used by data items contained in Working-Storage. Note that this option applies only to Version 7.2 and earlier.

The compiler allocates up to 32 data segments per program, each of which can be up to 64 KB in size. Data outside of Working-Storage must fit within these segments. Data contained in Working-Storage need not, but it can be slightly more efficient to place Working-Storage data within these segments. The compiler normally places Working-Storage data within these segments until it places some in the segment identified by “--lastWSDataSeg”. After that point, the compiler places all Working-Storage data into a separate address space (identified by addresses larger than x“4000000” in the symbol table listing). This reserves space for other data items that follow Working-Storage, such as the Screen Section or literals found in the Procedure Division.

The default setting is “24”. This reserves 8 segments (512 KB) for use by the Screen Section and literals. This is normally much more than needed. If it is not enough, then setting “--lastWSDataSeg” to a lower value will reserve more space.

When compiling for Version 6.0 or earlier object format, the default setting is “32”. This causes all of Working-Storage to be allocated in data segments. This sets an absolute limit of about two MB on the amount of “small” data that a program can allocate. The term “small” in this sense means data items individually smaller than 64 KB.

We recommend leaving this option at its default value unless you receive the error message “*Program exceeds 32 segments*”. Should this occur, try setting “--lastWSDataSeg” to a value smaller than the default of “24”.

- |                     |                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>--noAlignLit</b> | Inhibits use of the default algorithm for aligning literals in memory. Use the “-Da” option (described above) to specify an alignment modulus other than the default of “4.”                                                                                                                                                                                                                                                               |
| <b>--truncANSI</b>  | There are four truncation options in all. See the chart in <b>Section 2.2.10.1</b> for description and comparison information on these options.                                                                                                                                                                                                                                                                                            |
| <b>--noTrunc</b>    | There are four truncation options in all, see the chart in <b>Section 2.2.10.1</b> , for description and comparison information on these options.                                                                                                                                                                                                                                                                                          |
| <b>--fastRefMod</b> | Directs the compiler to use an optimized method for handling code that performs reference modification. The result is faster runtime execution. Note that this option may not work with certain programs, resulting in unexpected results or compiler failure. If using this option, be sure to verify runtime results. When this option is not specified, the compiler uses the standard method for reference modification code handling. |

### 2.2.10.1 Truncation Options

In addition to the compiler’s default truncation behavior, the compiler also supports the ability to use a data item’s storage capacity instead of its PICTURE phrase to determine the largest numeric value it could hold. This is the “-Dz” compile option. However, other COBOL systems have similar concepts that differ in some details from “-Dz”. To give you the greatest amount of flexibility and compatibility with other COBOLs, there are four

truncation options for numeric values. Only one of these options may be used for any individual compile. The following chart describes and compares these options.

| Option                 | Behavior                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>&lt;default&gt;</b> | Full ANSI COBOL rules are in place. Each numeric data item stores values up to its PICTURE in size. A small number of USAGE types provide exceptions (such as COMP-X). These are documented under the USAGE phrase. Values larger than allowed by the PICTURE are truncated using the standard size rules when the data item is the target of a MOVE statement. The results of an arithmetic overflow (without the SIZE phrase) are undefined. |
| <b>--truncANSI</b>     | Similar to the default, but COMP-5 is added to the list of data types that ignore their PICTURE when determining the largest value they can hold. However, COMP-5 items do use their PICTURE when moving a value to a nonnumeric data item. The name of this option is similar to the name used by some other COBOL systems that behave this way.                                                                                              |
| <b>--noTrunc</b>       | All binary data types ignore their PICTURE when determining the largest value they can hold. However, the PICTURE is used when moving data from a binary number to a nonnumeric data item. The name of this option is similar to the name used by some other COBOL systems that behave this way.                                                                                                                                               |
| <b>-Dz</b>             | All binary and packed-decimal data types ignore their picture when determining the largest value they can hold. The PICTURE is not used when moving to a nonnumeric destination (the largest possible value determines the number of digits moved instead).                                                                                                                                                                                    |

## 2.2.11 Video Options

- Va** This option must be used with either the “-Vh” or “-Vl” option. When used, the default intensity specified by the “-Vh” or “-Vl” option will be used only for ACCEPT statements. DISPLAY statements will use the *opposite* intensity. Thus “-Vha” will cause the default intensity for ACCEPT statements to be high while the default for DISPLAY statements will be low. When Screen Section items are used, then input and update fields will use the ACCEPT intensity, while output and literal fields will use the DISPLAY intensity.
- Vb** Causes the phrase “BLANK LINE”, when used in a Screen Section entry, to be treated as if it were written “BLANK EOL”. Most other COBOL compilers work this way, but the X/Open COBOL standard requires the syntax supported by ACUCOBOL-GT.
- Vc** Causes any ACCEPT statement that contains a numeric or numeric edited receiving field to be treated as if the CONVERT phrase were also specified. This is the default for RM/COBOL-85 but not for RM/COBOL version 2 or VAX COBOL. For information about the CONVERT phrase, see the **CONVERT Phrase** in Book 3, *Reference Manual*.
- Vd** Causes non-USAGE DISPLAY numeric items to be converted to USAGE DISPLAY before the screen display occurs. This option is used in conjunction with the “-Ca” switch for Micro Focus and HP COBOL compatibility.

**-Ve** Alters the rules that determine which conditions cause an ON EXCEPTION phrase to receive control in a Format 1 ACCEPT statement. Normally, entering an exception key will cause the ON EXCEPTION phrase to receive control. If you specify this option, then you must follow the “-Ve” with a digit (as part of the same argument) taken from the following list:

‘1’ Exception keys cause the ON EXCEPTION phrase to execute (default handling).

‘2’ Conversion error causes the ON EXCEPTION phrase to execute. When a numeric item is being entered and the CONVERT phrase is specified, then entering an illegal number will cause the exception.

‘3’ Combines the effects of both ‘1’ and ‘2’.

By default, the runtime system handles conversion errors by displaying an error message and forcing the user to re-enter the data. If you specify “-Ve2” or “-Ve3”, then conversion errors are returned to your program instead. For strict compatibility with VAX COBOL, you should specify “-Ve2”, while for strict compatibility with RM/COBOL, you should specify “-Ve3”. The default setting is recommended, however, because it provides a meaning for ON EXCEPTION that is more consistent with the rest of the ACUCOBOL-GT language.

**-Vh** Forces the default video intensity for ACCEPT and DISPLAY statements to be set to *high* intensity. The default for ACUCOBOL-GT is to use the normal operating intensity for the terminal being used.

**-Vi** Changes the behavior of a Screen Section ERASE (and BLANK) phrase with respect to color handling. If the screen item containing the ERASE phrase has either BACKGROUND-COLOR or COLOR specified, then the “-Vi” option causes the erased area to be painted with that color instead of the subwindow’s background color.

Keep in mind that background color can be specified using a background color number with the COLOR phrase. For a table of foreground and background color numerical values, see the entry for **FOREGROUND-COLOR and BACKGROUND-COLOR Phrases** in section 6.4.9 of Book 3.

If the screen item has no background color specification, or if the background color portion of a COLOR phrase is zero, then the subwindow’s background color is used.

The default behavior is to use the current subwindow’s background color when executing an ERASE phrase.

**-Vl** Forces the default video intensity for ACCEPT and DISPLAY statement to be set to *low* intensity.

**-Vq** Causes ACUCOBOL-GT to produce quiet programs by inhibiting the default bell produced by ACCEPT statements compiled in RM/COBOL compatibility mode. However, this does not override explicit BELL or BEEP phrases.

**-Vu** Allows you to imply the UPDATE phrase for all Format 1 ACCEPT statements that do not have an explicit UPDATE or DEFAULT phrase specified for them.

**-Vx** Allows exception keys to be entered by the user for any ACCEPT statement. Normally, ACUCOBOL-GT inhibits the use of exception keys on ACCEPT statements that do not have a CONTROL KEY clause or an ON EXCEPTION clause.



---

## 2.2.12 Warning and Error Options

---

**Note:** The “-w” is an existing compiler option to turn warnings off. The “-w” option will negate the effects of “-Wa” and -Wl”. To use the previous “-w” option, it must be specified alone (as the “-v” option is).

---

**-a** This flag is now obsolete and should not be used.

**-Wl** Generates the following 01-level item warning:

```
USING parameter <name> is not an 01-level item
```

The ANSI COBOL standard requires that parameters passed to subprograms be 01-level items. ACUCOBOL-GT does not restrict them as such; however, there are valid reasons for restricting their use. For example, starting in version 7.0.0, the compiler can generate better code for certain moves and comparisons, based on the alignment of the underlying data types. When those data types are in LINKAGE, the alignment rules that the compiler assumes may not be valid. Making all of the passed parameters be 01-level items ensures that the compiler’s assumptions about alignment of the data items are valid. Note that the compiler can generate incorrect code when the assumptions are invalid. In such situations, it is possible to get a MAV at runtime.

**-Wa** Generates the following alignment warning:

```
USING parameter <name> not aligned and may
cause problems in the called subprogram
```

To be less restrictive, the compiler also includes an alignment warning. This is generated whenever a passed parameter is a group or is binary, and whose alignment is not an even multiple of the alignment specified by the **“-Da#” option**.

**-Wr** Display warning messages for 01 level redefined zones if they are smaller than the redefiner.

- Qm** This option specifies the number of errors the compiler reports before it exits. The option must be followed by a positive numeric argument, which is the maximum number of errors the compiler reports before it exits. The default value is “100”.
- Qp** This option allows the compiler to skip entire sections of code when it finds an error. In particular, when this option is used, the compiler skips to the next period (“.”) when an error is detected.

### 2.2.13 Debugging Options

All the information obtained when you use the Debugging Options listed below is coded and stored in the COBOL object file. The runtime retrieves, decodes, and sends it to the debuggers (runtime and AcuBench integrated) when requested.

- Ga** This is a shortcut for turning ON all the debugging options.
- Gd** This option includes the source code in the compiled object file. It also generates a single-byte “no operation” (NOP) instruction for each CONTINUE statement to make it easier to break at such statements when debugging.

See also **section 3.1, “Runtime Debugger,”** for additional details about this option.

- GI** This option includes line numbers in the object file. It does not include source code in the object file. Only a mapping between addresses and line numbers is included. This allows the runtime to output line number information along with an address when a program terminates abnormally. This information also enables AcuBench to find and indicate the line of source on which the debugger is currently stopped.

If line number mapping is needed in an object to be placed in production, “-GI” is the best switch to use because it results in an object file that is much smaller than those produced with “-Ga”.

- Gs** This option includes extra symbol information. It does not include source code in the compiled object. This information is accessible only from AcuBench. The extra symbol information can be used by AcuBench to get all the “children” and “siblings” of any particular data item. It allows AcuBench to traverse the symbol table of the COBOL program and to have a tree-type control for viewing group data items.
- Gy** This option includes minimal symbol information. It does not include source code in the compiled object. Instead, minimal information about names and locations is stored in the object. Using this switch, you can accept and display variables from within the debuggers.
- Gz** This option restricts a program from stopping in the debugger in nearly all cases. The only time the runtime will stop and enter the debugger when executing a COBOL program compiled with -Gz is when the runtime encounters a format 2 STOP statement (STOP literal), which is meant to break into the debugger.

This option is useful if you distribute COBOL objects and allow your users to create their own COBOL programs, but you don't want them to be able to debug your programs.

This option is not compatible with any other debugger option. If other debug options are used on the command line, the last debugger option will take precedence. For example, specifying “-Ga -Gz” will result in an object that will never stop in the debugger, while specifying “-Gz -Ga” will result in an object with full debugging symbols.

This option is available only if targeting an 8.1 runtime or later. Note that specifying “-Znn” for any nn <= 80 will turn off a special flag that is required in order for the -Gz option to work.

---

**Note:** To debug a program in AcuBench, you need only the “-Gl” and “-Gs” options. This combination allows AcuBench to display the correct line of the correct source file when the debugger is stopped at a breakpoint and to display any data items that you want to see. It does not include the source code in the compiled object, so the object can be safely distributed without fear of unauthorized access to source code.

To perform source-level debugging with the runtime debugger, you need only the “-Gd” and “-Gy” options.

The “-Ga” option is helpful when you expect to use both debuggers.

---

The section on the runtime debugger (**section 3.1, “Runtime Debugger”**) describes the differences between symbolic debugging and source-level debugging.

## 2.2.14 Mapping Options

Each mapping option causes the compiler to produce a simple report about the program that is being compiled. Mapping options that produce reports prevent the compiler from generating an object file or XFD file. In this situation, errors in the source program are usually not reported. Only one report may be produced at a time, so several of the mapping options are mutually exclusive.

Specification of any mapping option causes the compiler to ignore all text positioned between EXEC and END-EXEC keywords.

Mapping options are frequently used internally by AcuBench to implement some of the editor features.

- Mc** Lists all of the special style and property names associated with graphical controls. You can combine this with the “-Mr” option to get a listing of both reserved words and property names. You might use this to produce word lists for the AcuBench Code Editor, to specify the words to be colored as “reserved” words.
- Ml** Causes the “-Mp” (paragraphs) and “-Mv” (variables) reports to be printed in lowercase. The default is uppercase. See also the “-Zw” compiler option ([section 2.2.16, “Miscellaneous Options”](#)).
- Mm** Causes the “-Mp” (paragraphs) and “-Mv” (variables) reports to be printed in mixed upper and lowercase. In this mode, the beginning of each word is capitalized, and the rest of the word is made lowercase. See also the “-Zw” compiler option ([section 2.2.16, “Miscellaneous Options”](#)).
- Mo** Names the output file for a “map” report. This option must be followed (as the next argument) with the name of a file. If this option is not used, then “map” reports are sent to the compiler’s standard output.
- Mp** Causes the compiler to produce a report on the Procedure Division sections and paragraphs contained in the program. If you select this option, the program is not compiled into an object file.
- Mr** Causes the compiler to produce a report of the reserved words known to the compiler. The exact list depends on the other compile time options. You can use this option to produce a word list that can be used by the workbench’s editor to color reserved words. The workbench comes with the full reserved word set pre-programmed. If you modify the set of reserved words with any of the “-R” compiler options, you can use this option to produce a custom list that matches the reserved words you use. Selecting this option prevents the compiler from producing an object file.

- Ms** Causes the compiler to produce a report of the COBOL source files used in the program. This includes the program itself and any COPY libraries. This list can be useful for generating dependencies if you use a “make” utility, and can help you determine which files to send to Technical Support staff when you need help resolving compiler issues. This option prevents the compiler from producing an object file.
- Mss** Similar to -Ms, this option lists all the COBOL source files needed to compile a program. If you specify -Mss instead of -Ms you will get an enhanced listing with the following format:
- ```
path base-filename "C" or "R" source-file  
line-number
```
- where:
- path*** is the pathname processed by the compiler. This path may be relative to the current directory. For Win32 compilers, path is the full pathname leading up to the COPY file.
- base-filename*** is the base file name of the COPY library.
- “C” is listed if the file is a standard COPY library or the original source file.
- “R” is listed if the file corresponds to a COPY RESOURCE statement.
- source-file*** is the name of the file containing the COPY statement
- line-number*** is the line number of the COPY statement. The compiler lists a “0” if the file is the original source file.
- AcuBench uses this format when you import programs to the workbench.
- Mu** Causes the “-Mp” (paragraphs) and “-Mv” (variables) reports to be printed in uppercase. This is the default. See also the “-Zw” compiler option (**section 2.2.16, “Miscellaneous Options”**).
- Mv** Produces a report of the data items (variables) declared in the program. This option prevents the compiler from producing an object file.

2.2.15 Conditional Compilation Options

The compiler supports conditional compilation through the use of special constructs in the COBOL source file and by accepting command-line arguments that turn on compiler directives and set constants to values. (Refer to the *ACUCOBOL-GT Reference Manual*, **Section 2.5, “Conditional Compilation”** for a description of the COBOL constructs).

The `-/` (a forward slash) compiler option is used to turn on directives and specify constants.

The two forms to this option are:

1. Directive setting mode

```
-/[NO] directive-name
```

This option is equivalent to having the following line in your COBOL source from the very beginning of the compilation:

```
$SET [NO] directive-name
```

By adding this compile switch, you turn on the directive (or off, if the directive is preceded by “NO”). This directive can now be tested with a Format 3 \$IF statement, which can signal the compiler to either compile or skip the next lines of code.

Refer to the *ACUCOBOL-GT Reference Manual*, **Section 2.5, “Conditional Compilation”** for details on the \$IF and \$SET statements.

If compiling for HP compatibility (`-Cp`) the `-$IF` and `-$SET` directives are supported. See Section 4.4.5 of the *Transitioning to ACUCOBOL-GT* guide for details.

2. Value setting mode

```
-/CONSTANT name=value
```

this option is equivalent to having a level-78 item defined in your COBOL source. If "value" is composed solely of numeric digits, the constant is considered numeric; otherwise, it is considered a string literal. "name" can now be tested with a Format 1 \$IF statement.

Note: The `-Cg` compiler option will turn off the conditional compiling features. Conditional compiling statements begin with the \$ sign. In prior versions, lines of code that began with the \$ sign were treated as comments. Unfortunately, some COBOL programs use a type of conditional compilation, which now fail to compile. Turning off conditional compiling should rectify these situations.

2.2.16 Miscellaneous Options

- | | |
|-------------------------|---|
| --acceptrefresh | This option takes the most recent value of what was entered on the screen and then uses that value in subsequent ACCEPT statements. If you enter something in a screen section that goes to a variable, then MOVE something to that variable, the --acceptrefresh option will allow the subsequently MOVED value to be the basis of what is in the next ACCEPT statement. |
| --arithmeticVSC2 | Causes truncation of intermediate results in arithmetic statements to follow the rules of VS COBOL II and COBOL/370. |
| --binaryMath | Causes the compiler to use binary math operations to handle arithmetic as long as the target runtime supports these operations; this can be abbreviated "--bin". |

- brand *serial#*** Causes the compiler to embed the specified value into the object file. *Serial#* is a separate argument of up to 20 characters. When you specify this option and a value, the COBOL object is associated with the value of *serial#* in such a way that it will execute only with a runtime that has the corresponding serial number. Attempts to execute the program with a runtime that has a different serial number (either as the initial program, or as a called subprogram) result in the runtime displaying the message “Incorrect serial number” and returning a call error of 28. To display a runtime’s serial number, execute the runtime with the “-v” option.
- decimalMath** Causes the compiler to use decimal math operations to handle arithmetic, overriding other compiler flags; can be abbreviated “--dec”. This is the default behavior. You may wish to experiment with the --binaryMath option, which may improve performance; however, you should test precision and accuracy.
- defines** This option must be followed by the name of a file as the next separate argument. This option causes the compiler to find all level 78s in a COPY file and create equivalent C-language “#define” statements in the specified file. For more information, see **section 2.2.21, “Help, Version Information, and Communication With C Programs.”**

--newARC

With "--newARC", the syntax rules for Abbreviated Combined Relation conditions become relaxed to allow for parenthesis to appear immediately after the relation.

For example:

```
IF A = (B OR C OR D)
```

would now be accepted by the compiler.

--oldARC

This option is the default behavior and specifies that standard syntax is to be used for Abbreviated Combined Relations. This syntax is described in **Section 6.3.8** in the *ACUCOBOL-GT Reference Manual*.

-Za

Causes the compiler to generate code to test array references at runtime. If an index is used which is out-of-bounds, the runtime system displays an error message showing the index value and the allowed bounds. (This causes some extra code to be generated and prevents certain table optimizations from occurring, so it should be turned off once a program is fully debugged.) With this option, the compiler does not re-use previously computed index values.

Because subscript overflow is the most common cause of memory access violation errors, adding this option is a good first step when you are looking for the cause of a memory access violation. (See **section 6.4.1, “Memory Access Violations,”** for more details.)

This option also generates code that performs range checking on reference modification at run time. Reference modification that creates a data item whose size or leftmost position parameter is out of range of the data item it references causes the runtime to display the error message “Reference modifier range error.”

This option also causes the runtime to test the LINKAGE items automatically. When an item defined in the Linkage Section is not referenced in the USING phrase of the Procedure Division statement and has not had an address assigned to it with the SET ADDRESS OF phrase, the runtime returns an error message.

The -Za option also specifies size testing of linkage items passed to subprograms. when a LINKAGE SECTION parameter of a size larger than the caller passed is detected, an error to that effect is reported.

-Zc

This “compact” option causes ACUCOBOL-GT to optimize for smaller code instead of faster code.

- Zd** Although still supported, this option has been replaced by the **“-Gd” option**. Both options produce the same results.
- Zg** Enables the use of segmentation (overlays) in the source. If this option is not used, section numbers will be ignored. This option applies only to Version 7.2 and earlier source files.
- Zi** Causes the program to be compiled as if it had the “IS INITIAL PROGRAM” phrase specified in its PROGRAM-ID paragraph. See book 3 **Section 3.2, “PROGRAM-ID Paragraph”** for details on IS INITIAL PROGRAM. This can be useful, in some cases, when you are compiling programs that do not contain adequate memory management (no CANCEL verbs, for example).
- Zl** All data items may be larger than 64KB. This option is obsolete. (In versions prior to 6.0, this option caused the compiler to allow data items larger than 64 KB.) See Book 3, *Reference Manual*, **Section 5.1.6, “Large Data Handling”** for more information.
- Zm** Causes the compiler to generate code that tells the runtime the size of a data item specified in a Format 7 **SET Statement**, as in:

`SET pointer TO ADDRESS OF data-item.`

This option supports thin client applications that pass pointers in calls to DLLs on the display host. For complete information, see section 7.2.6, “Calling Dynamic Link Libraries (DLLs),” in the *AcuConnect User’s Guide*.
- Zn** This turns off ACUCOBOL-GT’s local optimizer. This is useful primarily to see if the optimizer is introducing errors in the generated object code. This option also prevents the compiler from re-using previously computed index values.

- Zo** Although still supported, this option has been replaced by the **“-Fo” option**. Both options produce the same results.
- Zr** Allows for a recursive **PERFORM Statement**. When this switch is used, the PERFORM verb is modified so that return addresses are stored on a stack. Only the most recent PERFORM statement has an active return address. When this option is used, a paragraph under the control of a PERFORM statement may (directly or indirectly) PERFORM itself. See **PERFORM_STACK**, in Book 4, Appendix H.
- This behavior is the default. For backward compatibility, “-Zr” acts like “-Zr1”. If you compile for compatibility with Version 3.1 or earlier, recursive PERFORMs are turned *off* by default. This behavior may affect existing programs that do not compile for earlier compatibility. Event procedures require the ability to do recursive PERFORMs.
- Zr0** This option tells the compiler not to allow recursive PERFORMs. Event procedures require the ability to do recursive PERFORMs.
- Zr1** This option tells the compiler to allow recursive PERFORMs. Event procedures require the ability to do recursive PERFORMs.
- Zs** Although still supported, this option has been replaced by the **“-Gy” option**. Both options produce the same results.

-Zw

This option prepares a program for import into the AcuBench Screen Designer. It causes the compiler to mark windows and screen controls with additional information that can be used by the graphical screen import utility or Character-to-GUI Wizard. Graphical windows imported into the Screen Designer will be more complete if the program is first compiled with this option, and character-based screens converted by the wizard are more likely to contain the proper control type for each field. Using this option makes your compiled program somewhat larger, so you should use it only when you are importing screens into the workbench. (See the “-import” and “--char2gui” options in **section 2.3, “Using the Runtime System,”** for related information.)

You can also use the “-Ml”, “-Mm”, and “-Mu” options to affect the case of the additional information. This will affect how these items are seen in the Screen Designer, and in your subsequently generated code. Specifying “-Zw -Ml” will cause the data imported into the Screen Designer to be in lowercase. The default is uppercase (“-Mu”). To specify uppercase for the first letter of each word, and lower case for the rest of each word, use “-Mm”.

If you compile with “-Zw”, your screens may use only literal reference modifications, i.e., “data-item(4:20)”, for any of their data items to be imported into workbench correctly. Data items that use more complicated variants of reference modification do not import correctly. For example, “data-item(data-start:data-length)” imports as “data-item(:)”, and you need to insert the correct values into the workbench.

-Zx

Although still supported, this option has been replaced by the “-Fx” **option**. Both options produce the same results.

-Zy

This option lets you treat ACCEPT FROM DATE as ACCEPT FROM CENTURY-DATE, and ACCEPT FROM DAY as ACCEPT FROM CENTURY-DAY. If you use this option, the 4-digit year format will be used for ACCEPT FROM DATE providing that:

1. The receiving field is numeric or numeric edited and contains eight or more integer digits; or
2. The receiving field is not numeric or numeric edited and contains eight or more character positions.

If neither of the above conditions applies, then ACCEPT FROM DATE will return its normal 6-digit format even if you use “-Zy”.

ACCEPT FROM DAY works in the same fashion, except that the receiving field must have seven or more digits/positions in order to receive the new format.

Information on ACCEPT FROM DATE and ACCEPT FROM DAY is located in the *Reference Manual* entry for the **ACCEPT Statement**, Format 3, in section 6.6.

-Zz

This option causes spaces in a USAGE DISPLAY numeric item to be treated as the value zero, and non-numeric data to be treated as numeric. It does this by treating the high-value half of each byte as “3” so as to bring all bytes in the variable within the range of 30 to 3F.

“-Zz” must be specified at compile time in order to prevent the optimizer from mis-constructing the program. Note that this option should be used only if you need it, because it causes less efficient programs to be produced.

- Z3** Causes ACUCOBOL-GT to produce programs that can be run by the Version 1.3 runtime system. Specifying this will limit several features of later versions. If you use a restricted feature, you will receive an error message at compile time. This flag also implies the “-C3” option to ensure compatible behavior of the compiled program (see above).
- Z4** Similar to the “-Z3” option, except that the produced object can be run by the Version 1.4 runtime system. Implies “-C4”.
- Z5** Causes the compiler to produce object files that can be run by the Version 1.5 runtime system. This option implies “-C5”.
- Z20** Creates object code that can be run with a Version 2.0 runtime (chart verbs are not supported). This option implies “-C20”.
- Z21** Creates object code that can be run with a Version 2.1 runtime
- Z22** Creates object code that can be run with a Version 2.2 runtime
- Z23** Creates object code that can be run with a Version 2.3 runtime
- Z24** Creates object code that can be run with a Version 2.4 runtime
- Z30** Creates object code that can be run with a Version 3.0 runtime
- Z31** Creates object code that can be run with a Version 3.1 runtime
- Z32** Creates object code that can be run with a Version 3.2 runtime
- Z40** Creates object code that can be run with a Version 4.0 runtime

-Z41	Creates object code that can be run with a Version 4.1 runtime
-Z42	Creates object code that can be run with a Version 4.2 runtime
-Z43	Creates object code that can be run with a Version 4.3 runtime
-Z50	Creates object code that can be run with a Version 5.0 runtime
-Z51	Creates object code that can be run with a Version 5.1 runtime
-Z52	Creates object code that can be run with a Version 5.2 runtime
-Z60	Creates object code that can be run with a Version 6.0 runtime
-Z61	Creates object code that can be run with a Version 6.1 runtime
-Z62	Creates object code that can be run with a Version 6.2 runtime
-Z70	Creates object code that can be run with a Version 7.0 runtime
-Z71	Creates object code that can be run with a Version 7.1 runtime
-Z72	Creates object code that can be run with a Version 7.2 runtime
-Z73	Creates object code that can be run with a Version 7.3 runtime
-Z80	Creates object code that can be run with a Version 8.0 runtime
-Z81	Creates object code that can be run with a Version 8.1 runtime

2.2.17 Upper and Lower Case

Command-line arguments that begin with a hyphen are *not* case sensitive. For example, “-LO” and “-Lo” and “-lo” all mean the same thing. In this manual, group letters are shown in upper case, and the actual arguments are shown in lower case (for example, “-Lo”). This is done for consistency and clarity.

Other command-line arguments are case sensitive if the host machine is case sensitive, otherwise, they are not. For example, on UNIX machines, “ccbl TEST” and “ccbl test” are different commands because the file names “TEST” and “test” are different on UNIX systems. On VMS systems, these commands are the same because the two file names are treated as the same by the operating system.

2.2.18 File Name Handling

Several compiler options (such as “-o”) take a file name as an argument. As a special abbreviation, the character “@” can be placed in these file names to stand for the *base name* of the source program. To form the base name, the compiler removes all directory information from the source file name, along with any characters following a period, and the period itself. The resulting base name is then inserted into the file name argument at the location of the “@” character.

For example, if you want to compile a program called “TEST.CBL”, and you want the resulting object file to be named “TEST.OBJ”, and the error output to be placed in “TEST.ERR”, you can use the following command:

```
ccbl -o @.OBJ -e @.ERR TEST.CBL
```

This is equivalent to the following:

```
ccbl -o TEST.OBJ -e TEST.ERR TEST.CBL
```

and is also equivalent to:

```
ccbl -oe @.OBJ @.ERR TEST.CBL
```

In this example, the “-o” indicates that the next argument is the name of the object file. The “@.OBJ” argument is this name. The “-e” indicates that the next argument is the name of the error file. The “@.ERR” argument is this name. The “@” character is replaced with the word “TEST”, which is derived from the name of the source file “TEST.CBL”.

As another example, on UNIX machines you can compile the program “test.cbl” and place the object code in the file “/programs/test” with the command

```
ccbl -o /programs/@ test.cbl
```

(Substitute “\@” for “@” if your UNIX machine uses “@” as its default “line kill” character.) This can be particularly useful in compile scripts or in conjunction with the CBLFLAGS variable described below.

2.2.18.1 Remote file name handling

If AcuServer or AcuConnect is running on remote machines, the compiler can read remote source files from and write resulting files to those systems. The remote filename syntax that is used with the compiler is different than that used with the runtime and AcuServer. Because the compiler uses the “at” sign (“@”) as a placeholder for the base name of the source file, you cannot use that symbol as a tag in front of the name of the remote system (for example, “-o @server:/objects/@.acu” is ambiguous). Instead, the syntax is more like standard URL syntax. The remote filename specification must be of the form:

```
acurfap://server:[port]:filename
```

acurfap stands for “Acucorp Remote File Access Protocol.”

Remote file name notation can be used with any compiler option that takes a file name as an argument. The “//server:port” notation can be used with any configuration variable that allows remote name notation. See Appendix H, “Configuration Variables” in Book 4 for details on configuration variables.

If AcuServer is listening on the default port of a server machine, you do not have to specify the port number because the compiler defaults to that number (“6523”). However, if you omit the port number, you must include two colons (“::”) before the file name. If AcuServer is listening on a port other than the default, the “::” notation will not work.

Examples

If AcuServer is running on a UNIX machine named *myserver* and is listening on the default port, you can compile “apmain.cbl” straight to that machine with the following command.

```
ccbl -o acurfap://myserver::/myapp/obj/@.acu apmain.cbl
```

If “apmain.cbl” is located on the server, you can compile the remote file and write the resulting object file to the server with the following command.

```
ccbl -o acurfap://myserver::/myapp/obj/@.acu  
acurfap://myserver::/myapp/src/apmain.cbl
```

If AcuConnect is running on a Windows machine named *myserver* and is listening on port “5632”, you can compile “apmain.cbl” straight to that machine with the following command. Notice that if specifying a Windows directory, you must also include a drive letter followed by a colon (the “c:” in this example) after the port number (or the optional “:.”).

```
ccbl32 -o acurfap://myserver:5632:c:/myapp/obj/@.acu  
apmain.cbl
```

Note: When specifying a remote file name for a Windows machine, you can use the *backward slashes* (“\”) used by Windows, the *forward slashes* (“/”) used by UNIX, or a combination of the two.

2.2.19 Compiler Command-Line Examples

Here are some examples of common commands. The following command will simply compile the program “test.cbl” and produce the object file “test.acu”. The program will be compiled in VAX COBOL compatibility mode.

```
ccbl test.cbl
```

This next command will compile the same program in RM/COBOL compatibility mode. It will treat signs in a manner consistent with early versions of RM/COBOL and suppress reserved words not used by that compiler. These options are particularly common when you are converting programs written for RM/COBOL.

```
ccbl -Cr -Ds -Rva8 test.cbl
```

The following Windows console-mode (DOS-box) command will compile the program “TEST.CBL” and produce an object code file named “TEST” in the “c:\objects” directory, and create a full listing called “TEST.LST” in the current directory.

```
ccbl32 -o c:\objects\@ -Lfo @.lst test.cbl
```

Finally, this command will perform the same function on a UNIX machine except that the listing will be sent to the print spooler “lp”.

```
ccbl -o /objects/@ -Lf test.cbl | lp
```

2.2.20 CBLFLAGS Environment Variable

To simplify the setting of compiler options, the environment variable CBLFLAGS may be set to a list of options. These options will be used each time the compiler is run. The CBLFLAGS variable is particularly useful for setting options that you use all the time.

If you have set the CBLFLAGS variable and need to *unset* it temporarily, you can use the “-x” command-line option to ignore it for any particular run of the compiler.

You may continue to use command-line options when you use the CBLFLAGS variable. All of the options from both places are used. Note that the environment variable CBLFLAGS is examined *after* the command-line options.

On UNIX machines, the CBLFLAGS variable is set in the environment. This can be done with the **export** command of the Bourne or Korn Shell or the **setenv** command of the C-Shell. Under VMS, the CBLFLAGS variable is set as a symbol. Under Windows NT/2000, this variable can be created in the Control Panel “System” applet, or for Windows 98 systems, in the “autoexec.bat” file with the SET command. Windows must be restarted for the new values to take effect. For more information about setting environment variables, see section 1.5, “Environment Variables.”

The '@' file name abbreviation is particularly useful in the CBLFLAGS variable. For example, if you set the variable to "-o /programs/@", then every program you compile will have its object file placed in the "/programs" directory with the same name as the source file.

Note: The number of "acceptable" arguments in the CBLFLAGS environment variable is limited to 50. The compiler accepts the first 50 arguments set and ignores all the rest in excess of that number. File specifications are counted as separate arguments, but combined options from the same group are counted as one argument.

2.2.21 Help, Version Information, and Communication With C Programs

You can get a summary of compiler options with the following command:

```
ccbl -help
```

You can display the copyright notice and version information with this command:

```
ccbl -v
```

You can get configuration information with the following command:

```
ccbl -vv
```

On UNIX systems, you can get additional port configuration information with:

```
ccbl -vvv
```

The information displayed with "-vvv" varies depending on the UNIX system and is subject to change from release to release.

You can find all of the level 78s in a COPY library and create equivalent C-language "#define" statements with the following command:

```
ccbl -defines filename
```

You can use the "-defines" option to simplify communication between COBOL and C programs.

One specific use of this option is to simplify the creation of Windows Help files when you are building context-sensitive help. Use this option to map context ID *strings* to context *numbers* for use with Windows Help files. First create a COPY library with level 78s that map the context IDs (strings) from your help file to unique context numbers (the Windows Help API function WinHelp requires a number as the context ID parameter). Then when you specify “cdbl -defines,” the compiler creates a file containing “#defines” that correspond to the level 78s in the COPY library you created.

The “-defines” option must be followed (as the next separate argument) by the name of the COPY library. *acurfap* syntax can be used to specify a file on a system being served by AcuServer or AcuConnect (see [section 2.2.18.1, “Remote file name handling”](#)).

By default “-defines” creates a file having the same base name as the specified file and the extension “.h”. For example, if your COPY library is named “PRHELP.DEF,” you could use this command:

```
CCBL -DEFINES PRHELP.DEF
```

to create the file “PRHELP.H” that contains one “#define” for each level 78 in “PRHELP.DEF.”

Be aware that any hyphens in the level 78 names are converted to underscores in the corresponding “#define.” Although hyphens are allowed in COBOL and in the help file, hyphens are not allowed in C names. See section 10.4 in Book 2, *User Interface Programming*, for more information about Windows Help files.

2.2.22 The “>>IMP” Directive

The “>>IMP” directive is an *implementor defined* directive, meaning that Micro Focus defines its specific uses (described below). The syntax is:

```
>>IMP (ACU-CBLFLAGS=CompilerOption)
```

CompilerOption can be:

```
-Dln, where n is the integer value 1, 2, 4, or 8
-Sa
-St
```

The directive must start in the indicator area: column 7 for ANSI format source; column 1 for terminal format source.

ACU-CBLFLAGS and its argument must be enclosed in parentheses. An equal sign (“=”) must immediately follow ACU-CBLFLAGS and immediately precede *CompilerOption*. Note that all three options require a leading hyphen.

The supported options allow you to work around two unusual compile time issues.

Specifying an alternate data alignment

You can use the “-Dln” argument to specify an alternate byte alignment for SYNCHRONIZED data items (for details on the “-DI” option, see [section 2.2.10, “Data Storage Options”](#)). For example, the RM COBOL compatibility switch (“-Cr”) implies two byte alignment (which is consistent with legacy RM COBOL). Some variables in your program may require a larger alignment. To apply the directive, in the source code immediately before the variable or set of variables (which may be contained in a COPY file), add the directive in the indicator area. For example, to specify four byte alignment add:

```
>>IMP (ACU-CBLFLAGS=-D14)
```

Immediately after the last variable that requires special alignment, set the alignment back to the original value:

```
>>IMP (ACU-CBLFLAGS=-D12)
```

Forcing compilation of ANSI or terminal format files

When specified with a compiler directive, the “-Sa” and “-St” options override the “-Sa” or “-St” option specified on the command line within the scope of the containing file.

In some instances, due to source origin issues, developers may choose to enforce compilation of a single source format by specifying either the “-Sa” (ANSI) or “-St” (terminal) compiler option in the command line. Without the use of a compiler directive, the compiler will not compile source files of another format within the same compilation unit. (In prior releases this was particularly problematic for terminal format compilation (-St) when

axdefgen had been used to create COPY files for ActiveX components, because **axdefgen** creates ANSI format COPY files. **axdefgen** now automatically inserts a compiler directive in the first line of all COPY files it creates.)

When the “-Sa” or “-St” option is specified in a compiler directive on the first line of the source file, the compiler compiles that source file regardless of the format option specified on the command line. Compilation of that format applies to only the file containing the directive.

For example, when the “-St” (terminal format) option is specified on the command line and you want to compile an ANSI format COBOL source file in the same compilation unit, place the following compiler directive in column 7 of the first line of the COBOL file:

```
>>IMP (ACU-CBLFLAGS=-Sa)
```

2.3 Using the Runtime System

The ACUCOBOL-GT runtime system (referred to in this manual as **runcbl**), runs the programs created by the compiler. Once compiled, programs are ready to run; no linking step is required. Programs compiled with ACUCOBOL-GT are machine transportable. **runcbl** accommodates the differences between machines.

To run an ACUCOBOL-GT program, enter the following command (substitute the name of your runtime for **runcbl**):

```
runcbl [options] [program] [parameters]
```

Program is the name of a compiled program. If omitted, its name defaults to “cbl.out” (or to the name you have set with the runtime configuration variable **DEFAULT_PROGRAM**). Remote name notation is allowed for the name of the compiled program, if your runtime is client-enabled. See [section 5.2.2, “Remote Name Notation,”](#) for more information.

Parameters are one or more arguments that can be passed to the program. These arguments can be accessed through the CHAINING phrase of the Procedure Division header in the compiled program. For details, see the entry for the “**CHAIN Statement**” in Book 3, *Reference Manual*, section

6.6. If *parameters* are specified, then *program* must also be specified. Under VMS, the parameters that are not in double quotes are converted to lower case. Parameters should be enclosed in double quotes to preserve case sensitivity. The maximum number of parameters allowed on the command line is 50.

Options is a series of one or more of the following flags. These options must be preceded by a hyphen. You can specify more than one option by simply combining them. Option characters may be either upper or lower case.

Separately, or in addition to placing options on the command line, options can be specified in the ACUSW environment variable. ACUSW can contain any runtime options, which are specified with the same syntax used on the command line. ACUSW and command-line options can be used together. ACUSW is processed after the command line, however, the command line takes precedence with options that specify a filename. For example, you can specify a default error file in ACUSW (e.g., with the “-e” option) and then override it on the command line for a particular run. The “--no-acusw” option inhibits the processing of ACUSW. This is valuable for programs that directly invoke the runtime and require a fixed set of options that the user is not allowed to modify with ACUSW.

2.3.1 Runtime Options

The allowed runtime options include:

-# This option must be followed (as the next separate argument) by a series of letters that determine which SPECIAL-NAMES switches to turn on. There are 26 SPECIAL-NAMES switches. The letter “a” corresponds to switch 1, “b” to switch 2, and so forth. For example, to start the program with switches 1, 5 and 8 turned on, specify “-# aeh”.

For convenience, you can turn on any of the first 8 switches by simply specifying the switch number or numbers without the “#” argument. For example, “-# aeh” can also be specified as “-158”.

-a This flag is now obsolete and should not be used.

-b Inhibits the terminal initialization done by **runcbl**. This can be useful if the program is run in background because terminal initialization can prevent normal use of the terminal by the operating system. This is particularly true on UNIX systems. If you specify this flag, the behavior of ACCEPT and DISPLAY statements is undefined; therefore use this flag with caution. A program can examine the ACU-NO-TERMINAL field after an ACCEPT FROM TERMINAL-INFO statement to determine whether it was started with “-b”. See Format 3 of the **ACCEPT Statement** in Book 3, section 6.6.

-c

This option must be followed (as the next separate argument) by the name of an alternate runtime configuration file. It causes **runcbl** to use this configuration file instead of the default file. See **section 2.8, “Runtime Configuration.”**

Remote name notation is allowed for this option if your runtime is client-enabled. See **section 5.2.2, “Remote Name Notation.”**

--char2gui

This option is used to convert character-based screens into their graphical equivalents for use in the AcuBench Screen Designer. When you run your program with this option, ACUCOBOL-GT’s Character-to-GUI Wizard launches in the background.

After your program starts, navigate to the screen you want to convert and right-click on the window’s background. A pop-up menu is displayed. Select “Build Graphical Screen” to continue with the conversion. The Character-to-GUI Wizard then creates a graphical version of the current screen and displays it together with a Properties dialog box. You can use the Properties dialog to make some basic changes to the screen. Repeat this process for each screen you want to convert.

When you are done, exit the application. When the application exits, the runtime writes an “import.out” file into your current working directory that contains information describing the converted screens. You can then start AcuBench and, using the “Add Screen” function, display the contents of the “import.out” file in a Screen Designer window. *If you already have a file called “import.out” in your current working directory, the wizard overwrites it; therefore, if you intend to convert screens in stages, you should rename the file and save it in a separate directory.*

If you execute the program in AcuBench, then after you exit the application, the workbench creates a new program in the workspace Structural View. The program’s Screen node contains entries for each screen described in the “import.out” file. Those screens open in the workbench development area, where they can be modified. Screen node entries can be moved in the workspace as needed.

It is important to note that the purpose of the Character-to-GUI Wizard is to simplify the initial task of converting traditional text-based applications into ones that use a graphical user interface. Although the wizard greatly reduces the task of converting character-based screens, it is only a first step in the process. It is expected that after you use the wizard, you will spend time manipulating the screens to your liking using AcuBench Screen Designer. You will also need to integrate the newly generated screen section code back into your program. For more information on using the Character-to-GUI Wizard, please refer to the *AcuBench User’s Guide*.

-d

This starts the program in debugging mode. See **section 3.1, “Runtime Debugger.”**

-e

This option must be followed by the name of a file (as the next separate argument). This option causes the error output from the runtime system to be placed in this file. This can be used to trap runtime system error messages and trace output. “-e” creates a new file or overwrites an existing file. Use “+e” to cause error output to be appended to the file. The format of the output can be tailored with the **TRACE_STYLE** configuration variable. See Book 4, Appendix H.

When specifying a runtime error file name you can use the following format specifiers:

“%p” If the name contains the string “%p”, that string is replaced with the process ID (PID) of the runtime.

“%d” If the name contains the string “%d”, that string is replaced with the current date in the form YYYYMMDD where YYYY is the year, MM month, and DD day.

“%t” If the name contains the string “%t”, that string is replaced with the current time in the form HHMMSSTTT where HH is the hour, MM minute, SS second, and TTT milliseconds.

“%u” If the name contains the string “%u”, that string is replaced with the username.

“%h” If the name contains the string “%h”, that string is replaced with the hostname.

Note that these specifiers may also be used in the file names configured with the **ACU_MON_FILE** and **ACU_DUMP_FILE** configuration variables.

Under UNIX systems, redirecting error output causes problems for “more” and “vi”. For this reason, we offer two options for redirecting error messages under UNIX:

“-e” - causes all of the runtime’s tracing and error messages and DISPLAY UPON SYSERR output to go to “errorfile”. It does not redirect stderr. This means that error output from programs called by CALL “SYSTEM” is not redirected. If you call “more” or “vi” from within COBOL, you can safely use “-e” to redirect error messages.

“-ee” - If you expect programs called by CALL “SYSTEM” to send their errors to the error file, use the option “-ee” instead of “-e”.

Remote name notation is allowed for this option if your runtime is client-enabled. See **section 5.2.2, “Remote Name Notation.”**

--embedded-config-file

This option causes the runtime to load and use a configuration file embedded in a COBOL object library. The name of the embedded configuration file can be specified with the runtime -c option. Otherwise, it must be named “cblconfi” or “cblconfig”.

The configuration file may be embedded either by using cblutil or the “COPY RESOURCE” statement.

The object library must be preloaded using the runtime -y command-line option. This is so that the configuration file settings will be available before the primary module is loaded.

Certain configuration variables must be set before the object library is loaded. Therefore, these variables cannot be set in an embedded configuration file. The following is a list of variables that *cannot* be set in an embedded configuration file:

CGI
MESSAGE-QUEUE-SIZE
ICON
NO-CONSOLE
LOCKS-PER-FILE
TEST-CHAR
MAX-FILES
MAX-LOCKS
WINDOW-TITLE

The runtime uses the following higher default values for the LOCKS-PER-FILE, MAX-FILES, and MAX-LOCKS variables when “--embedded-config-file” is specified:

256 LOCKS-PER-FILE

255 MAX-FILES

512 MAX-LOCKS

-f

This option ensures that the runtime does not perform user interface functions when the COBOL program is functioning as a Common Gateway Interface (CGI) program on the Internet. This option causes the runtime to suppress warning messages that are normally displayed in a message box. If the runtime shuts down due to an error that is not handled by the COBOL program, it constructs an HTML page containing the shutdown message and sends it to the standard output stream before terminating. This option performs the same function as the environment variable “A_CGI” but does not affect the entire environment.

-g This option causes the error file (specified after the “-e” option) to be compressed with the gzip compression method. A compressed file must be decompressed with gzip before reading or editing. For clarity, it is best to give the error file a “.gz” extension. When appending to an existing file (with the “+e” option), you must use the same format—compressed or uncompressed—in which the file was originally created.

-h This option causes the runtime to explicitly ignore hang-up signals. You can also ignore hang-up signals by specifying both the “-s” and “-b” options. However, the “-sb” combination also inhibits terminal initialization and prevents the user from killing a program with an abort key such as “Control-C” or “Delete”. Unlike “-sb”, the “-h” option ignores only the hang-up signals.

-i This option must be followed (as the next separate argument) by a file name. This causes the keyboard input to be taken from this file. It can be used as an alternate to input redirection on UNIX systems. Remote name notation is *not* allowed for this option.

Examine your input files carefully, paying particular attention to the way the <enter> key is represented. On many systems, it is represented by a hex “0A” (line feed). Note that the line feed does not, by default, terminate an ACCEPT. So, when you use the “-i” option, you will want to add the following to your “cblconfig” file:

```
KEYSTROKE TERMINATE=10 ^J
```

This option has no effect on Windows platforms.

-import

This option is available only on Windows and Windows NT systems. It requires the file “WEXPRT32.DLL”, which must be installed in the same directory as the runtime executable. This option is used to import graphical screens created with ACUCOBOL-GT Version 3.x or AcuScreens so that these screens can be used with the AcuBench Screen Designer. If you are running with this option, simply right-click on any window to have the opportunity to add it to the file “import.out”. See the AcuBench documentation for details.

It is important to note that the original purpose of the screen import utility was specifically to upgrade users from AcuScreens to AcuBench, and it was not intended as a permanent device to keep importing all the new screens you create either from scratch or from AcuBench. For that reason, when new control types are added, the screen import utility is not necessarily updated at all, or it may be updated with basic information about the new control type but not all the different properties and styles of the new control type. You should not rely on this utility to be able to import all new screens you create.

When the screen import utility tries to import an unrecognized type or property of a control, you will see the following message on your screen:

This screen contains at least one control type that the Screen Import Utility does not know about. You should add these controls manually.

- k** This option causes the immediate playback of a keystroke file. It must be followed (as the next separate argument) by a file name. The *filename* argument is the name of a file containing recorded keystrokes. The runtime internally calls W\$KEYBUF using opcode “9” and this file name prior to executing the first COBOL program. The effect is that the keystrokes recorded in the file are treated as the runtime’s first user input. For more information see **W\$KEYBUF**, in Appendix I in Book 4, *Appendices*. Remote name notation is *not* allowed for this option. Use this as an alternative to “-i” in Windows systems.
- l** Causes a listing of the contents of the runtime configuration file to be printed on the error output. Prints the runtime’s version number on the first line. Also prints the steps taken by **runcbl** when it is trying to load a program, along with any problems encountered. This is useful for debugging problems with the configuration file or program path resolution (see **section 2.8, “Runtime Configuration”**). This is best used in conjunction with the “-e” option to capture the debugging information in a file.
- m *value file*** Turns on memory handling descriptions. These descriptions report detailed information about memory allocation, reallocation, and frees. For more information, see **section 6.4.3.1, “Memory handling descriptions.”**
- no-acusw** Inhibits the processing of the ACUSW environment variable.
- no-save-debug** This option has two effects: (a) it prevents the debugger from reading the “.adb” file, thus causing the debugger to start in its default state, and (b) it prevents the debugger from writing out a new “.adb” file when it exits.

The debugger saves state information in a “.adb” file which is used when the debugger is executed in another run. This information includes window placement and breakpoint settings. There are some cases when you may find this inconvenient, and the “--no-save-debug” option provides a way to eliminate this behavior.

--no-signal-handlers

This switch allows you to initialize the runtime without installing its signal handlers. This option is designed for use in environments like CICS that call the ACUCOBOL-GT runtime from a C main program and want to install their own signal handlers. For more information, see the entry for `acu_abend()` in section 6.4.3 of *A Guide to Interoperating with ACUCOBOL-GT*.

-o

This option must be followed by the name of a file that will take the display output from the program. This is similar to output redirection on UNIX systems. If “+o” is used instead, then the output is appended to the named file. Remote name notation is *not* allowed for this option.

This option has no effect on Windows platforms.

-p

Activates a built-in execution profiling facility, prompting the runtime to collect information about I/O operations and CALLs, and to install a timer to track the amount of time spent in different parts of the code. Information collected by the runtime is placed into an output file called “acumon.xml”. Note that if you want zero execution count paragraphs included in the report, you should use the “-p0” option. For more information, see [section 3.7, “The Profiler.”](#)

- p0** Tells the profiler to include zero execution count paragraphs in the “acumon.xml” file. For more information, see [section 3.7](#).
- r** Starts the program in debugging mode (like “-d”). This option must be followed by the name of a file containing debugging commands. The debugger is run under control of this file. Remote name notation is *not* allowed for this option.
- s** Runs the program in “safe” mode. On non-UNIX systems, the “-s” option prevents the user from killing the program with the operating system’s abort key (Control-C, Delete, etc.). However, any kill command will interrupt the program run. On UNIX systems only, the “-s” option must be issued *twice* (runcbl -ss) to protect it from the system’s abort key. This option allows only a kill -9 to stop the program run.
- “Safe” mode can help preserve the integrity of files used by the program. If the program is not in “safe” mode, then **runcbl** will automatically close its files if the user kills the program. Note that this keeps each file intact but does not keep separate files synchronized with each other, which may be required by the user’s application.

-t

This option can be used to capture the runtime's terminal output to a disk file. This option must be directly followed by a filename of the output file.

The -t option can be used instead of piping the output to the "tee" command. Notice that piping runtime output to "tee" can cause the runtime to hang. This is because runtime detects that the output is not a terminal and so it will not set terminal attributes for the terminal. In such state, the runtime has a hard time accepting input, and the output may not be flushed to the screen in a timely manner.

When the "-t *filename*" flag is set, all the output to the terminal goes to this file, including cursor addressing. This option can be used only with a version of the runtime which has an addressable terminal capability. It will not work with any of the graphical runtimes, nor will it work with the Windows console runtime.

--time

Causes the runtime, at shutdown, to write the total real time spent executing to its error output file. This option can be used if you want to measure the time it takes to execute a stand alone batch program.

Note that such real time measurements are inexact, because they do not account for time spent on other tasks or waiting for external output.

- u** By default, the runtime tests each use of a LINKAGE data item to check that the item passed by the calling program is at least as large as the item declared by the called program. This ensures that unallocated memory is not accidentally referenced. The “-u” option disables that test, as well as the test that verifies that all parameters of a subprogram were passed by the caller. (The same can be accomplished with the **CHECK_USING** configuration variable. See Appendix H.)
- v** Prints the current version number of **runcbl**, the serial number, and the maximum number of users licensed to use the runtime simultaneously. No program is run.
- vv** (double “v”) Prints the current version number of **runcbl**, along with extended information. No program is run.
- vvv** (triple “v”) This option is valid on UNIX systems and causes **runcbl** to display additional configuration information about the UNIX port. The information displayed varies depending on the UNIX system and is subject to change without notice. No program is run.
- w** This has the same effect as specifying “WARNINGS 0” and “MAKE-ZERO 0” in the runtime configuration file. This option is provided for compatibility with previous versions of ACUCOBOL-GT. We recommend that the corresponding configuration entries be used instead.

-x

When a file error “30” occurs, the root cause of this error is often not apparent. Specifying “-x” will cause the runtime system to display the operating system’s corresponding error number on the error output. This information may help in determining the problem. You can use the “-e” option to direct the error output to a file.

-y

This option causes the runtime to pre-load the specified ACUCOBOL-GT object library, UNIX/Linux shared object library, or Windows DLL. This option must be followed (as the next separate argument) by the name of the library to load. You can pre-load multiple libraries by specifying multiple “-y” options. If the library is a DLL, the C calling convention can be specified after the name (see section 3.3.2, in *A Guide to Interoperating with ACUCOBOL-GT*).

The directory of the object module and ENTRY points contained in the library are loaded by the runtime before it loads the main program. All of the object modules in the library are thus available to be called at any time. Note that the main program may be contained in the library because the library is loaded first.

When specifying a shared object library, you can include the file suffix or use the **SHARED_LIBRARY_EXTENSION** configuration variable to specify the filename extension.

Note that shared libraries can also be loaded with the **SHARED_LIBRARY_LIST** configuration variable. You can also use the **SHARED_LIBRARY_PREFIX** configuration variable to specify a set of directories that the runtime will search when attempting to locate a shared library. For more information on these variables, see their entries in Appendix H of Book 4.

Libraries loaded with the “-y” option remain in memory until the process exits. The CANCEL statement cannot be used to unload the library.

ACUCOBOL-GT object libraries are described in more detail in **section 3.2, “Object File Utility — cblutil.”** Windows DLLs and UNIX shared libraries are described in Chapters 3 and Chapter 6 of *A Guide to Interoperating with ACUCOBOL-GT*.

Remote name notation is allowed. See **section 5.2.2, “Remote Name Notation.”**

Note: “-y” does not load client-side DLLs for thin client applications that make calls using the CALL verb “@[DISPLAY]:” syntax. These applications must explicitly load the DLL by calling it with the CALL verb before calling a function within the DLL.

-z

After an unexpected runtime termination resulting from a memory access violation, this option causes the program to output the current contents of memory where the violation occurred.

2.4 Compatibility Modes

ACUCOBOL-GT is designed to make porting programs between different environments as easy as possible. To aid in this, the compiler runs under one of five *compatibility modes*. These five modes support source code portability to/from VAX COBOL, ICOBOL, RM COBOL (version 2), IBM DOS/VS COBOL, and HP COBOL programming environments. ACUCOBOL-GT supports a reasonable subset of these COBOL implementations. This is accomplished in two fashions. When these compilers have differing but distinct syntaxes for accomplishing the same thing, all syntaxes are supported. For example, in RM/COBOL, the word BLINK is used to turn on the blinking attribute. In VAX COBOL, the phrase used is WITH BLINKING. Because these phrases do not conflict with each other, both are supported.

In some cases, the compiler cannot simultaneously accept both methods of doing something because they conflict. For example, VAX COBOL and RM/COBOL have different rules for determining the default cursor position for a DISPLAY statement. In this case, ACUCOBOL-GT defaults to using the VAX COBOL interpretation unless the “-Cr” or “-Ci” flag is used at compile time. The use of these flags is detailed in **section 2.2.5, “Compatibility Options.”**

Many operational characteristics can be configured at runtime. See **Section 2.8, “Runtime Configuration,”** for details.

See Book 3, **Chapter 2, section 2.3.1, “ANSI ACCEPT and DISPLAY Verbs”** for compatibility information concerning ANSI ACCEPT and DISPLAY verbs.

2.5 Source Formats

ACUCOBOL-GT supports two different source formats. One format is ANSI compatible. The other format is suitable for interactive COBOL development from a terminal.

ANSI format has the following characteristics:

1. Columns 1 - 6 are used for **Sequence Numbers**. This area is ignored by the compiler.
2. Column 7 is the **Indicator Area**.
3. Columns 8 - 11 are **Area A**.
4. Columns 12 - 72 are **Area B**.
5. Columns 73 - 80 are the **Identification Area**. This area is ignored by the compiler (but see Source Code Control below).
6. Lines are 80 characters long. Lines shorter than 80 characters are padded with spaces, and longer lines are truncated.

Terminal format is convenient for developing programs interactively. This format has the following characteristics:

1. The **Sequence Number** area is eliminated.
2. The **Indicator Area** is in column 1. All of the usual COBOL indicators are accepted here except for the conditional debugging line indicator “D”. This indicator must be preceded by a backslash (\) in column 1 (placing the “D” in column 2).
3. **Area A** also starts in column 1 unless an indicator is present, in which case it starts immediately after the indicator character.
4. **Area B** starts in column 5 and extends to the end of the line.
5. The **Identification Area** starts when a “[” or “*>” is encountered, provided it is not part of a literal. The Identification Area extends to the end of the line. This can be used to introduce in-line comments.
6. The line ends when a carriage-return or new-line is found. Lines may be longer or shorter than 80 characters.

Note: Although the compiler accepts lines longer than 80 characters, the runtime debugger does not display characters past the 80th column. If possible, use the AcuBench runtime debugger in such cases.

ACUCOBOL-GT Terminal mode is compatible with the VAX COBOL terminal source format, except for the introduction of the Identification Area, which VAX COBOL does not support. ACUCOBOL-GT Terminal mode is also compatible with ICOBOL terminal source format except that Area A must start in column 1 (in ICOBOL Area A may start in column 2).

Both formats expand tab characters to every eight spaces. Both formats also translate lower-case characters to upper-case except in literals. Finally, both formats translate the underscore character to a hyphen when it is found in identifiers.

Normally, the compiler determines the source format automatically by examining the first character of the first non-blank line. If this character is blank or a digit, the file is assumed to be an ANSI file; otherwise it is assumed to be in terminal format. This is done independently for the main source file and all COPY libraries. This allows mixing of formats among a source file and its COPY libraries. If desired, the format to use for the entire input source can be set to either mode via the “-Sa” or “-St” compile flags.

2.6 COPY Libraries

The COPY verb accepts the following forms:

```
COPY library-name [ OF path-name ] [ SUPPRESS ].
COPY RESOURCE resource-name [ OF path-name ].
```

where *library-name* and *path-name* are either user-defined words or alphanumeric literals. *Resource-name* is an alphanumeric literal.

Environment variables may be used in the *path-name*--this is indicated by a \$ sign before the name of the variable. For example:

```
COPY library-name OF "$COPYLIB".
```

You may also use multiple environment variables to define a path:

```
COPY library-name OF "$LIB/$COPYLIB/$SUBDIR".
```

Library-name is the file name of the copy file to include in the object library. *Resource-name* is a file (other than a COBOL object) that is to be included in a COBOL object library. If *path-name* is specified, then it is treated as a directory specification for *library-name* or *resource-name*.

The file name is derived by concatenating *path-name* with *library-name* or *resource-name*, using the appropriate syntax for the host operating system. For example, on a UNIX system,

```
COPY MYCOPY OF "/usr2/acctdir"
```

is translated to

```
COPY "/usr2/acctdir/MYCOPY".
```

Note: Please note the use of quotation marks on *path-name* and on *library-name* and *resource-name* when they include lower case letters. This is a requirement on systems that are case sensitive, because the runtime would otherwise convert lower case to upper case.

The “-Ce” compatibility option can be used to specify an alternate default file extension. See **section 2.2.5, “Compatibility Options.”**

A library name can be specified by a user-defined word which may include a period. This is a special extension to the meaning of “user-defined word”. This allows you to specify a file extension without putting the name in quotes. For example, the two statements:

```
COPY "MYFILE.CPY"
```

and

```
COPY MYFILE.CPY
```

are both allowed and mean exactly the same thing.

When a period is placed in a library name which is a user-defined word used without quotes, it must be preceded and followed by a (non-period) character normally allowed in an identifier (thus “.MYFILE”, “MYFILE.” and “MYFILE..CPY” are all disallowed).

In addition, the COPY RESOURCE statement allows the resource name to be a user-defined word just like in a Format 1 COPY statement (the resource name does not have to be an alphanumeric literal).

Note: User-defined words are always treated as uppercase on machines where file names are case-sensitive.

If the derived file name is a full path name or contains a drive designation, then that name is used unmodified. Otherwise, a series of directories is searched to find the COPY file. The default search path is just the current directory.

You can modify the search path by setting an operating system variable. On UNIX, Linux, and Windows systems, you do this by setting the environment variable COPYPATH to be the list of directories to search. For VMS systems, the *symbol* COPYPATH is set instead.

The COPYPATH variable consists of a series of prefixes to apply to the file name. Each prefix is terminated by a special character that depends on the host operating system.

System	Separator
Windows	Semicolon

System	Separator
UNIX/Linux	Colon
MPE/iX	Colon
VMS	Comma

For example, to search the current directory, the directory “/u/pr” and the directory “/u/ap” on a UNIX system, the appropriate COPYPATH setting would be:

```
:/u/pr:/u/ap:
```

The initial colon indicates an empty prefix (i.e., the current directory). Note that the “-Sp” compiler option can also be used to specify the search path for COPY libraries. If used, it takes precedence over the COPYPATH setting.

You may use the word SUPPRESS to exclude the contents of a COPY file or any COPY files nested within from the program listing. For example:

```
COPY MYFILE OF "$MYLIB" SUPPRESS.
```

COPY statements may contain other COPY statements. This nesting is limited by the total number of files that the operating system will allow a program to open at once.

2.6.1 Resource Files

A resource is a piece of static data (such as a bitmap) that is required by the program and is embedded directly into the object file. Because a resource is not actually a separate file in the target environment, using resources reduces the number of physical files required on the target machine and thus can simplify the installation of your programs. Using resources is essential if you want to include bitmap images or sound files in an application that will be dynamically loaded over the World Wide Web. Bitmap files (BMP and JPEG), sound files (WAV), ActiveX resource files, and runtime configuration files are the only types of resources supported in the current version of ACUCOBOL-GT.

Note: In order to use JPEG files, you must have the file “ajpg32.dll” installed in the same directory as the runtime. Only 32-bit runtimes support JPEG format images. If you need JPEG support on 64-bit Windows, run the 32-bit runtime or the Thin Client. You can also run the Thin Client with the 64-bit runtime.

2.6.1.1 General Rules for Resources

The following rules apply to resources in general.

1. Resources are files (other than COBOL objects) that are included in a COBOL object library.
2. Resources are named as if they were files, but without any directory information. A resource takes on the same base name as its source file (including suffix). For example, if you include the resource “c:\mystuff\toolbar1.bmp”, that resource would be called “toolbar1.bmp” in the COBOL object library.
3. Resources preserve the case of their names in the object (as specified at the time they were included in the object). However, resource names are matched regardless of case. Thus, in the preceding example, a program could refer to the resource as “toolbar1.bmp” or “ToolBar1.Bmp” with identical results.
4. A resource name with a hyphen (“MY-FILE”) is considered equivalent to the same resource name given with an underscore (“MY_FILE”).
5. Resources are accessed by various specific COBOL subroutines or operations. Currently, there are five useful resource types. These are: bitmap, JPEG, WAV, ActiveX resource, and runtime configuration files. The library routines “W\$BITMAP”, “WIN\$PLAYSOUND” and “C\$RESOURCE” can access resources. See **Section 2.3, “Using the Runtime System,”** for information about embedding a runtime configuration file in an object library.
6. The presence of one or more resources in an object converts that object into a library. A library consists of a collection of COBOL objects and resources (either of which may be absent). COBOL objects are named in a library by their PROGRAM-IDs, while resources are named by

their file names. The CALL verb ignores resources when trying to find a COBOL object, and resource processing routines (such as W\$BITMAP) ignore COBOL objects when trying to find a resource.

7. You can include a resource in an object file by using either the COPY RESOURCE statement or by using “cblutil -lib”.

2.7 Source Code Control

ACUCOBOL-GT provides a method for conditionally modifying the source program at compile time. This is accomplished by having lines in the source that can be excluded or included at compile time. This can be used to maintain different versions of the program, perhaps to support different machine environments.

The “-Si” (include) flag controls the actions of the source code control system. It must be followed by an argument that specifies a pattern that the compiler will search for in the Identification Area of each source line. If the pattern is found, then the line will be included in the source program, even if it is a comment line. However, if the pattern is immediately preceded by an exclamation point, then the line will be *excluded* from the source (i.e., commented out). The exclamation point here stands for the notion *not*. Note that the pattern is case sensitive; enclose it in double quotes on systems such as VMS where you need to preserve its case.

The “-Sx” (exclude) flag works in the same fashion except that its meaning is reversed (i.e., lines with the pattern will be commented out and lines with a preceding exclamation point will be included).

Here is an example. Suppose that a source program is being maintained for both the UNIX and VMS environments. The following piece of code is in the program:

```
MOVE "SYS$HELP:HELPPFILE" TO FILE-NAME.    VMS
* MOVE "/etc/helpfile" TO FILE-NAME.      UNX
OPEN INPUT HELP-FILE.
```

This program fragment is ready to be compiled for the VMS system. If a UNIX version is desired, then the following command line will correct the source during compilation:

```
ccbl -Si UNX -Sx VMS source
```

The first “-Si” flag will cause lines marked with “UNX” to be included in the source. The second flag will cause lines marked with “VMS” to be excluded from the source. An alternate way of doing the same thing would be:

```
MOVE "SYS$HELP:HELPPFILE" TO FILE-NAME.      !UNX
* MOVE "/etc/helpfile" TO FILE-NAME.         UNX
OPEN INPUT HELP-FILE.
```

This would be compiled for UNIX systems with:

```
ccbl -Si UNX source
```

The line marked with “!UNX” is commented out when this command is run because of the exclamation point. This alternate method is particularly appropriate if only two versions are being maintained.

This source code control system can be especially convenient if the source is being maintained for both ACUCOBOL-GT and non-ACUCOBOL-GT environments. The ACUCOBOL-GT extensions can be commented out and marked with a source-code control flag. When the program is compiled under ACUCOBOL-GT, these lines can be included.

Note: The patterns maintained in the Identification Area should be in upper case because some systems convert command lines into upper case. Also note that exclamation points usually need to be quoted on command lines. As a final note, the entire Identification Area is searched for the pattern; other information may be placed there too.

2.8 Runtime Configuration

Many aspects of the runtime system can be controlled through *configuration variables*. Configuration variables can be modified by each **runcbl** site as well as directly by an ACUCOBOL-GT program. This allows a great deal of flexibility in adapting **runcbl** to a particular system.

Configuration variables are maintained in a *configuration file*. This is a standard text file that can be modified by the host system's text editor. See **Appendix H: Configuration Variables** in the Appendices Manual for details on variable syntax and usage.

The following sections describe some of the most frequently used configuration variables. A complete list of configuration variables along with a description of their use can be found in Appendix H.

2.8.1 File Name Assignments

File names referenced in an ASSIGN clause can be dynamically reassigned at runtime by configuration variables (this is sometimes referred to as *name aliasing*). Each file's ASSIGN name is searched for in the configuration environment and, if found, replaced by the value of the matching configuration variable. The exact meaning of this reassignment is detailed in the **section 2.9, "File Name Interpretation."**

For example, if your code contains a SELECT statement like:

```
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT idx-file  
    ASSIGN TO DISK "idx.dat"  
    BINARY SEQUENTIAL  
    STATUS IS idx-status.
```

you could define a name alias for "idx.dat" in your runtime configuration file by adding the line:

```
idx.dat  usr2/data/idx.dat
```

Whenever your application references "idx.dat" in a SELECT statement, the alias "/usr2/data/idx.dat" is substituted.

Remote name notation is allowed in file aliases if your runtime is client-enabled (for indexed files, remote name notation requires the Vision file system). See **section 5.2.2, "Remote Name Notation."**

Likewise, Internet notation is allowed if you are using AcuXML. For example, to read the XML file “bookfile.xml” over the Internet, you could map the file to its URL in your runtime configuration file:

```
BOOKFILE http://myserver.mycomp.com/data/bookfile.xml
```

Note that the XFD files required by AcuXML must still be available locally (or via AcuServer) in the named XFD_DIRECTORY, but the data stream will be read from the server myserver.mycomp.com via HTTP.

Note: Aliasing is strongly recommended for accessing physical devices such as printers. This allows a program to easily adapt to the device naming conventions used at each individual site.

For portability across applications, the following names are recommended for device files:

PRINTER	System spooler, default printer
PRINTER1-9	System spooler, printers 1-9
FORM	Default special-form printer
FORM1-9	Special-form printers 1-9
TAPE	Default tape/floppy transport
TAPE1-9	Tape/floppy transports 1-9

For example, a COBOL program might assign a print file to the system spooler, printer number 2, with the following line:

```
ASSIGN TO PRINT "PRINTER2"
```

Although the assignment of these names to physical devices is arbitrary, the following conventions are recommended:

1. “PRINTER” devices should be assigned to the host operating system’s spooler, if applicable. If the host operating system does not have a spooler, then they should be assigned directly to the print devices.
2. “FORM” devices should not be assigned to a spooler. They should, instead, be assigned directly to the printer. These are intended to be used by programs that need to control the printer directly without intervention by another program.

3. “TAPE” devices can be assigned in any sensible manner.

No names are initially assigned. The process of assigning names is covered in [section 2.9, “File Name Interpretation.”](#)

For more selective aliasing, for example, changing the name of a single Vision segment, see the [filename](#), [filename_DATA_FMT](#), and [filename_INDEX_FMT](#) configuration variables described in Book 4, Appendix H.

2.8.2 Code and Data File Search Paths

A series of directories can be specified that are used to locate program object files and data files. Object files are located via the [CODE_PREFIX](#) configuration variable. Data files are located by the [FILE_PREFIX](#) variable.

Each of these variables specifies a series of one or more directories to be searched for the desired code or data file. The search method is detailed in [section 2.9, “File Name Interpretation”](#) and [section 2.10, “Calling Subprograms.”](#)

The directories are specified as a sequence of space-delimited prefixes to be applied to the file name. All directories in the sequence must be valid names. The current directory can be indicated by a period (regardless of the host operating system). For example, the following line:

```
FILE_PREFIX      . /usr/data
```

specifies that data files should first be searched for in the current directory and then in the “/usr/data” directory. For convenience, colons can be used along with spaces as a delimiter (except on Windows systems where semicolons are used, and VMS systems where commas are used). As an example of this, the following may be specified for a Windows system:

```
CODE_PREFIX      C:\;C:\OBJ
```

The “^” (carat) character can be specified in [CODE_PREFIX](#) to indicate the directory containing the calling program.

Note: Directory names with embedded spaces can be used as `FILE_PREFIX` and `CODE_PREFIX` configuration variables if the directory name is enclosed in quotes, for example:

```
FILE_PREFIX C:\"Sales Data"
```

This example will cause the program to search for files first under the referenced directory name, and then in the current directory.

You can also specify a default file name extension for both code and data files. By default, ACUCOBOL-GT makes no assumptions about the file name extension used by files. A default name extension can be specified for code files by the configuration variable **CODE_SUFFIX** and for data files with the variable `FILE_SUFFIX`. These suffixes are automatically appended to any file name that does not explicitly contain an extension. A period is automatically placed between the file name and the extension, if needed.

For example, you can cause all of your object files to have the implicit extension of `“.COB”` by placing the following line in your configuration file:

```
CODE-SUFFIX COB
```

2.8.3 File Status Codes

ACUCOBOL-GT supports five different sets of FILE STATUS code values. These five sets correspond with the values used by RM/COBOL-85, RM/COBOL version 2, Data General ICOBOL, VAX COBOL, and IBM DOS/VS COBOL. By default, ACUCOBOL-GT uses the RM/COBOL-85 values.

You can select another set of file status codes at runtime by setting the configuration variable **FILE_STATUS_CODES** to one of the following values:

“74”	-RM/COBOL version 2 codes
“85”	-RM/COBOL-85 codes (default)
“DG”	-ICOBOL codes
“VAX”	-VAX COBOL codes
“IBM”	-IBM DOS/VS codes

The exact values used by each set are covered in Appendix E in Book 4, *Appendices*.

Note: The set of status codes used depends solely on the current setting of the `FILE_STATUS_CODES` variable and not on any compile-time settings. This is done to provide a consistent set of status codes to every program in a run unit. If you need to change the set of status codes during a run unit, you can do so with the `SET ENVIRONMENT` verb. Some programmers use this verb to ensure status code settings for a particular program, or in situations where programs that conform to the 1985 COBOL standard and programs that do not conform are both in use at a site.

2.8.4 Terminal Handling Options

Many aspects of the terminal I/O sub-system can be modified by configuration variables. These are described in detail in [section 4.2, “Getting Your Terminals Ready.”](#)

2.8.5 File Handling Options

Some COBOL systems do not abort when a file error occurs and there is no Declarative to handle it. If you desire this behavior, set the runtime configuration entry `ERRORS_OK` to one of the non-zero values described in the entry for [ERRORS_OK](#) in Appendix H of Book 4.

Caution: When `ERROR_OK` is enabled, file errors that would normally cause ACUCOBOL-GT to abort will instead allow processing to continue. This means that when `ERRORS_OK` is enabled, you will not get the usual error reporting. Under most circumstances this is undesirable.

Some compilers also automatically create files when they are opened for I/O or `EXTEND`, if the files are missing. You can simulate this behavior by setting [EXTEND_CREATES](#) to “1” if you want to create files opened for `EXTEND` and setting [IO_CREATES](#) to “1” to create files opened for I/O.

RM/COBOL version 2 automatically closes all files except print files when a program executes an EXIT PROGRAM statement. By default, ACUCOBOL-GT does not do this, because it is a violation of the ANSI standard. If the configuration variable **CLOSE_ON_EXIT** is set to 1, then this behavior is emulated by **runcbl**. You can also set this variable to 2 to cause all files (including print files) to be closed when a program exits. Setting the variable to zero resets this option.

RM/COBOL version 2 also automatically causes a page eject when a print file is closed unless the WITH NO REWIND option is used on the CLOSE statement. The configuration variable **PAGE_EJECT_ON_CLOSE** can be set to “1” to cause this behavior. Setting it to “0” causes the normal behavior of not ejecting a page.

You can improve the performance of indexed files that are opened WITH LOCK by setting the configuration variable **MASS_UPDATE** to “1.” This causes ACUCOBOL-GT to treat these files as if they were opened with the “MASS-UPDATE” phrase specified. For details on this phrase, see **section 6.1.6.2, “Mass update.”**

2.8.5.1 Sort files

The SORT verb often makes use of temporary files. By default these files are stored in the current directory. You can specify an alternate directory to hold the sort files by setting the configuration variable **SORT_DIR** to the desired directory. This value is treated as a prefix, so any trailing directory syntax (such as “/”) is required. You can improve the performance of the SORT verb by placing the temporary files on a fast device. Care should be taken, however, that the device has enough free space to hold twice the size of the data to be sorted.

2.8.5.2 Carriage control

RM/COBOL handles carriage-control characters in a line sequential file differently on different systems. By default, both ACUCOBOL-GT and RM/COBOL-85 remove carriage-control characters from input records for line sequential files. This is the ANSI standard. RM/COBOL version 2, however, does not remove form-feed characters on MS-DOS machines and does not remove form-feed or carriage-return characters on UNIX systems. Some existing RM/COBOL version 2 programs depend on this behavior.

ACUCOBOL-GT can optionally retain any or all of these characters in the input record. If the configuration variable **CARRIAGE_CONTROL_FILTER** is set to “1”, then form-feed characters will be retained in the input record. If it is set to “2”, then carriage-return characters will be retained. If it is set to “4”, then line feeds will be retained. You may specify combinations of characters to retain by adding their corresponding values together. For example, specifying “6” causes carriage-returns and line feeds (2 plus 4) to be retained. Setting the variable to zero causes the default action of removing all three characters. Note that on VMS systems, carriage control information is not placed directly into data records and is instead maintained separately. For this reason, the **CARRIAGE_CONTROL_FILTER** setting has no effect on VMS systems and should not be considered portable to those machines.

2.8.5.3 Device locking

Finally, users on UNIX systems who want to make use of **runcbl**'s automatic device locking facility may set the variable **LOCK_DIR** to the desired lock directory. This process is covered in detail in **section 6.1.5, “Device Locking Under UNIX.”**

There are other configuration options that can affect file operations and performance. See Appendix H, in Book 4, *Appendices*, for a complete listing of configuration options.

2.9 File Name Interpretation

ACUCOBOL-GT employs a rich set of rules when translating a file name specified in an ASSIGN clause to an actual file name used on the host system. These rules provide a great deal of flexibility in placing files and dynamically reassigning them. These rules also allow for convenient handling of printers and other special devices. See section 2.8.2 for examples that illustrate many of the rules listed below.

When interpreting a file name, **runcbl** performs the following steps in order:

1. The initial name is taken from the ASSIGN clause of the file. If not specified in the ASSIGN clause, then the internal name specified by the SELECT clause is used instead.

2. The name is examined to see if `FILE_ALIAS_PREFIX` should be applied.
 - The file name is constructed by prepending the first string listed in **`FILE_ALIAS_PREFIX`** to the file name. The runtime searches for that name in the environment or the configuration file.
 - If the name is not found, the runtime constructs a new name by prepending the second string in `FILE_ALIAS_PREFIX` and searches for that alias.
 - The process is repeated with each string in `FILE_ALIAS_PREFIX` until a file alias name is found or the end of the list is reached.

If the file name includes a dollar sign (“\$”) and **`EXPAND_ENV_VARS`** is set to one (“1”), the `FILE_ALIAS_PREFIX` logic is applied to the environment variable name. So, for example, “\$FILE1” and “FILE1” are treated the same.

3. If this name does not start with a hyphen, then it is searched for in the environment. First the runtime system’s configuration variables are searched, followed by the host system’s environment. The value of the variable found becomes the new name. This search is then repeated until either no new translation is found or the new name starts with a hyphen. Note that when the runtime system’s configuration variables are initially loaded, any name found in both the runtime’s configuration file and the user’s environment is taken from the user’s environment.

For example, if the name “PRINT-FILE” is assigned to “PRINTER1” in the ACUCOBOL-GT configuration file, and you have “PRINTER1” assigned to “/dev/lp” in the host’s environment, then opening a file called “PRINT-FILE” will actually open the file “/dev/lp”. “PRINT-FILE” is first translated to “PRINTER1” and then “PRINTER1” is subsequently translated to “/dev/lp”.

4. If the name starts with a hyphen, it is interpreted in a special fashion described below. If it does not start with a hyphen, then it is considered to be a normal file name. This name is further processed as follows.
5. If the **`FILE_CASE`** configuration variable has been specified, the change to upper case or lower case is applied if appropriate.

6. The name is examined to see if it has an extension (zero to three characters following a period). If it does not, and the **FILE_SUFFIX** configuration variable has been defined, then the value of the **FILE_SUFFIX** is added to the name, with an intervening period if necessary.
7. The name is examined to see if **FILE_PREFIX** should be applied.
 - If the name contains a disk drive designation, or begins with a “\” (back slash), processing continues at step 8.
 - If the name begins with a “/” (forward slash) and the configuration variable **APPLY_FILE_PATH** is set to off (0), processing continues at step 8.
 - If the name begins with a “/” (forward slash) and the configuration variable **APPLY_FILE_PATH** is set to on (1), the current **FILE_PREFIX** is applied to the name and processing continues at step 8.
 - If the name does not contain a full path or drive designation, the current **FILE_PREFIX** is applied to it and processing continues at step 8.

FILE_PREFIX is a configuration variable that contains one or more strings that are prefixed to the filename. After each string is prefixed and the **EXPAND_ENV_VARS** variable is checked (see step 8), **runcbl** tries to find a file by that name. If it finds the file, processing of the name stops. If it does not, the next prefix is tried until no more prefixes are available. A prefix consisting of a single period (“.”) is treated as an empty prefix (i.e., the current directory is used).

8. Before the runtime tries to find the file, if **EXPAND_ENV_VARS** is set to on (1), the runtime expands any environment variables found in the name. A file specification that includes a “\$” character will have all the characters from “\$” to the end of the name or to the next “/” or “\” replaced with the value of the matching environment variable. For more information, see the entry for **EXPAND_ENV_VARS** in Book 4, Appendix H. The runtime now attempts to find the file. If the file is found, processing stops. If the file is not found and there is another **FILE_PREFIX** to apply, the prefix is applied and step 8 is repeated. If

all the prefixes have been tried and the file has not been found, the first prefix is re-applied, and processing of the name stops. This means that files newly created by **runcbl** will reside, by default, in the first directory specified in the `FILE_PREFIX` configuration variable.

Upper or lower case in a file name is significant if it is significant to the host operating system. The same is true when the host environment is searched for a translation. Upper or lower case is *not* significant when **runcbl**'s local environment is searched for a translation.

2.9.1 File Names Starting With a Hyphen

File names that start with a hyphen are treated specially. The hyphen and the following character are removed, as are any spaces following that character. The rest of the name is interpreted depending on the character after the hyphen as follows:

1. If the character is an “F”, then the name is treated as a normal file name. This differs from the default handling only in that the name is not further translated (i.e., no further searching in the environment and no application of the **FILE_PREFIX** or **FILE_SUFFIX**). This is useful if you want to ensure that a file name is placed in the current directory or remains untranslated.
2. If the character is a “D”, then the name is treated as the name of a device. This is treated no differently than the case for “F” above, except that file locking rules are slightly changed for devices. Devices may be assigned only to `SEQUENTIAL` files.
3. If the character is a “P”, then the name is treated as the name of a program to run. On UNIX systems, a “pipe” is created between **runcbl** and the named program. This program will then either receive data written to the file or provide the data read from the file.

On Windows and VMS systems this is handled somewhat differently. The translation of the name must contain at least two space-separated words. The first word is treated as the name of a file to write or read. This name is not further translated and is handled as a standard `SEQUENTIAL` file. The second and following words are treated as the program to run. If the file is being opened for input, the program is run first and then the named file is opened. If the file is being opened for

output, then the named file is first opened normally; when this file is finally closed, the named program is run. Typically the program should either create or read the named file as is appropriate.

On Windows and VMS systems you have the option of using the phrase “%TMP%” in place of a file name. In this case a unique name is created for you, and that same name is substituted for “%TMP%” every place on the line that the phrase is found. It is advisable to finish with (or rename) the file within 19 days, to avoid the possibility of name conflicts with names generated in the future. (See the examples below.)

If you use the “%TMP%” option to assign a file to a simulated pipe (by using “-P” in the assign name), you can specify where the temporary file will reside. This is done with the TEMP_DIR configuration entry. It acts just like the SORT_DIR entry, except that it applies only to %TMP% files.

On Windows systems, if you append an ampersand (&) character to the command line, the program will run asynchronously. This should not be done for programs providing input files, but is often useful for programs processing output files. For example:

```
assign to output "-P %TMP% cmd /c edit %TMP% &"
```

Program files may be assigned only to SEQUENTIAL files and may not be opened for I/O or EXTEND.

2.9.2 File Name Examples

Following are some examples of the file name translation rules. These examples point out some of the more useful aspects of the translation rules.

2.9.2.1 Example 1: Default name handling

Suppose you wanted to place data files in the “\DATA” subdirectory on a Windows system. You could then make the following assignment in the configuration file:

```
FILE_PREFIX      C:\DATA  D:\DATA
```

In this case, a file being created by an OPEN OUTPUT statement will be searched for first in the C:\DATA and D:\DATA directories. If it is found in either of these directories, that file will be removed and the new file placed in the same directory. If it is not found, however, then the new file will be created in the C:\DATA directory because that is the first one in the list.

Notice that the current directory is not mentioned in the above FILE_PREFIX configuration entry. This means that the current directory will not be searched for files. If you want the current directory to be one of the directories that is searched, add "." to the FILE_PREFIX configuration entry. If a particular file must be located in the current directory, use the "-F" flag. The "-F" can be specified in the ASSIGN name. For example, if you want to ensure that file "DIRLIST" is located in the current directory regardless of the value of FILE_PREFIX, place the following clause in your source code:

```
ASSIGN TO "-F DIRLIST"
```

The "-F" flag ensures that "DIRLIST" is not further translated or modified. Note that the "-F" flag also ensures that the name is not translated when it is used in a configuration variable. Thus, another way to accomplish the same result, is with these two steps:

1. Set the file name in the ASSIGN statement as:

```
ASSIGN TO "DIRLIST"
```

2. To cause the file to be placed in the current directory, place the following line in your configuration variable file, or set it in the environment:

```
DIRLIST -F DIRLIST
```

Note: In the above case, the file is placed in the current directory regardless of the contents of FILE_PREFIX.

Normally, you should avoid using the full directory path names or the "-F" flag in your source code. If you can, use FILE_PREFIX in the configuration file. This will provide the most flexibility in file management for each individual site.

2.9.2.2 Example 2: Accessing printers

ACUCOBOL-GT's name translation rules make it particularly easy to access a site's printers (and other devices) in a machine-independent fashion.

We recommend that you use some pre-designated names in your ASSIGN statements for files that are to be directed to a printer. Using these names will simplify the installation of runtime systems for sites using several ACUCOBOL-GT applications. The ASSIGN name for standard print files should be "PRINTER", or one of its variants "PRINTER1", "PRINTER2", etc. By convention, "PRINTER" is associated with the default printer on the host system, while the alternate names are associated with additional printers. For print files where you need direct control over the printer (to align special forms, for example), you should use one of the names: "FORM", "FORM1", "FORM2", etc. By convention, "FORM" devices are directly attached to the printer and "PRINTER" devices access printers through the system spooler (if available).

Each site can then place the appropriate definitions of these names in the configuration file (see **section 2.8, "Runtime Configuration"**). The installation of "FORM" devices is easy--simply name the device using the "-D" flag. For example, to associate the name "FORM" with the "/dev/lp" device on a UNIX system, place the following line in the configuration file:

```
FORM    -D  /dev/lp
```

It is important to ensure that the users have access permissions to the named devices. For more information about print spooler issues, see the *Getting Started* book and Appendix I, the **WINSPRINTER** library routine, in Book 4, *Appendices*.

The installation of "PRINTER" devices is only slightly more complicated. Because these should be spooled (if possible), you will usually need to use the "-P" flag to specify a program to receive the print output. The following notes give examples for various operating systems:

1. On UNIX systems, the name should be translated to the appropriate system spooler. This is usually either "lp" (for System V) or "lpr" (for Berkeley UNIX). You will usually need to specify additional flags to access devices other than the default one. If you are using the "lp" spooler, you should also specify the "-s" flag to prevent the "request id is . . ." message from appearing in the middle of your program.

For example, suppose the site has two printers, a high-speed line printer (the default) and a laser printer. Furthermore, suppose System V UNIX is being used and the laser printer is accessed with the flag “-dlaser”.

The following two lines should then be placed in the configuration file:

```
PRINTER...    -P lp  -s
PRINTER1      -P lp  -s  -dlaser
```

2. On VMS systems the usual way to print files is with the PRINT system command. One recommended way of doing this is to enter the following line in the configuration file:

```
PRINTER -P %TMP% PRINT /NOIDENT /DELETE %TMP%
```

Be sure to use the options shown above (NOIDENT and DELETE), and feel free to add other options as desired.

2.9.3 Assigning Files to Local Printers

For runtimes that use “a_termcap,” you may assign files to local printers. This makes it easy to switch between local and spooled printing in a single program--all you need to do is assign to a different device.

To assign a file to a local printer, assign the file to the filename “LOCALPRINT“. The file must be either a print file or a line sequential file, and it must be opened for OUTPUT or EXTEND.

If the terminal does not have enable-print/disable-print control sequences defined for it, you will receive an error “35” when you try to open the file. Otherwise, the open will succeed and you should be able to use WRITE statements normally. Note that the runtime has no way of knowing whether or not a printer is actually attached to the terminal.

The runtime sends an enable-print sequence prior to each line and a disable-print sequence after each line. If you prefer, you may assign to “LOCALPRINT-C” instead (the “C” stands for “continuous”). If you do this, then a single enable-print sequence is sent prior to printing the first line, and a disable-print sequence is sent when the file is closed. Note that this will cause any terminal output to be printed, so this should be used with care.

For runtimes that do not use “a_termcap” (such Windows), you should assign directly to the print device instead. This facility is not currently available under VMS.

2.10 Calling Subprograms

ACUCOBOL-GT is not a linked language. Instead, when a subprogram is called it is loaded dynamically at runtime. ACUCOBOL-GT does not use the name specified in the PROGRAM-ID paragraph. Instead, the name specified in the CALL statement is treated as the filename of the object file to load. Actually, **runcbl** maintains two names for each called object file: its *call name* (the exact name specified in the CALL statement) and its *filename* (the name by which the object file is actually found).

An object module can be in one of three states:

1. **Active.** This is the state of any program that has been called and has not yet exited. It also applies to the first (or main) program of a run unit. An active program is always memory resident.
2. **Loaded but inactive.** A loaded program is one that is memory resident but not running. This is the state of a program that has been called and has also exited. It remains in memory until it is the object of a CANCEL verb. A program in this state keeps its files and data items in the same state that they were in when the program exited. Note that programs with the INITIAL PROGRAM attribute are never in this state because they are automatically canceled when they exit.
3. **Not loaded.** This is the initial state of all object files. By default, not loaded indicates that the program is not in memory. However, if the *logical cancel* mechanism is enabled, the program may be cached in memory. For more information, see **section 6.3, “Memory Management.”**

2.10.1 CALL

When a CALL occurs, **runcbl** performs the following steps. If during any step the call is resolved, call processing ends:

1. **runcbl** searches the **DYNAMIC_FUNCTION_CALLS** configuration variable for a matching call name or call name prefix. If a match is found and the “dlopen(NULL)” feature is supported (true on most UNIX systems) the call name is searched for as a dynamic function in the current process. If it is not found, the call name is searched for as a dynamic function in each of the loaded shared libraries or DLLs specified with the “-y” option or with the **SHARED_LIBRARY_LIST** configuration variable.
2. If the runtime is acting as an application host in a thin client environment, it checks for the “@[DISPLAY]:” prefix and, if found, passes the call to the thin client.
3. The runtime next attempts to call any C subprograms that have been linked into the runtime system. (Interfacing to C routines is detailed in section 6.3 of *A Guide to Interoperating with ACUCOBOL-GT*.)
4. If the **CODE_CASE** configuration variable is defined, the change to upper case or lower case is applied as appropriate.
5. Next, the list of loaded but inactive programs and their ENTRY points is searched for a matching call name. If a name is found, that program is made active, execution is initiated at the proper point in the program.
6. Next, all loaded programs (both active and inactive) are searched to see if any are part of an *object library* (see section 3.2). For each object library found, the call name is searched for in the list of modules contained in the library. If found, that module is loaded out of the library and made active.
7. Next, on Windows systems the call name is searched for as the name of a routine in a loaded DLL.
8. Next, any libraries specified with the “-y” runtime command are searched for a matching call name. Note that programs in a library are searched for by their PROGRAM-IDs.
9. If the program has still not been found, the disk is searched for the object module.

The name is examined to see if it is a full path name.

- If the call name begins with a “\” (back slash), the call name is treated as the filename, and the object file (if it exists) is loaded and made active.
- If the call name begins with a “/” (forward slash) and the configuration variable **APPLY_CODE_PATH** is set to *off* (the default value), the call name is treated as the filename, and the object file (if it exists) is loaded and made active.
- If the call name begins with a “/” (forward slash) and the configuration variable **APPLY_CODE_PATH** is set to *on*, the current **CODE_PREFIX** is applied to it. Or, if the call name is not a full path name, the current **CODE_PREFIX** is applied to it.

The configuration variable **CODE_PREFIX** (see [section 2.8.2, “Code and Data File Search Paths”](#)) consists of a series of prefixes to apply to the call name. These prefixes are applied, in order, until a matching file is found or all of the prefixes have been tried.

For example, to search for object files in “/usr/obj/ar”, “/usr/obj/ap”, and the directory containing the calling program, you could set **CODE_PREFIX** to:

```
CODE-PREFIX /usr/obj/ar /usr/obj/ap ^
```

When each file name is formed, the configuration variable **CODE_SUFFIX** is checked to see if it has been defined. If it has, and the call name does not have an explicit extension specified, the current value of **CODE_SUFFIX** is appended to the filename with an intervening period. An extension is zero to three characters following a period at the end of the name. (See the entry for **CODE_SUFFIX** in Book 4, Appendix H.)

If **CODE_SUFFIX** is not defined, “.acu” is appended to the filename. If the file is not found, the file is searched for again, this time without the “.acu” extension.

Note: As with all filenames, upper and lower case are significant on UNIX machines but not on VMS and Windows systems.

10. Next, on Windows, the call name is searched for as the name of a DLL. If the DLL is found it is loaded and the call name is searched for as the name of a routine in it or any other loaded DLL (see section 3.3 in *A Guide to Interoperating with ACUCOBOL-GT*).
11. Next, on UNIX, the call name is searched for as the name of a shared library (see section 6.3.1 in *A Guide to Interoperating with ACUCOBOL-GT*).
12. Finally, the call name is searched for as the name of a dynamic function in the current process, and then in each of the loaded shared libraries or DLLs specified with the “-y” runtime command-line option or the **SHARED_LIBRARY_LIST** configuration variable. Note that to match as a dynamic function in Step 1 of this process, the call name must match a name in the **DYNAMIC_FUNCTION_CALLS** configuration variable.

Recursive CALLs

A program may directly or indirectly call itself. A CALL statement that calls the active program (itself) is a *recursive call*. For more information, see the **RECURSION** configuration variable in Appendix H, Book 4, *Appendices*. For information on sharing data in recursively called programs (such as in the HP e3000 environment), see the **RECURSION_DATA_GLOBAL** configuration variable.

2.10.2 CANCEL

When a CANCEL verb is executed, the following steps are performed:

1. If a CANCEL ALL statement is executed, all loaded programs that are inactive are set to the not loaded state.

Note: The **CANCEL_ALL_DLLS** configuration variable can be used to exclude DLLs and shared object libraries from the results of a CANCEL ALL statement. See Book 4, Appendix H for details.

2. Otherwise, the list of active programs is searched for a matching call name. If found, an active program is being canceled and the CANCEL is ignored.
3. Next the list of loaded but inactive programs is searched for a matching call name. If found, this program is set to the not loaded state.
4. If no matching programs are found, the CANCEL verb has no effect. This is not considered an error.

By default, placing the canceled program in the “not loaded” state (initial state) includes removing the program from memory. However, if the *logical cancel* mechanism is enabled, the canceled program is cached in memory. For more information, see [section 6.3, “Memory Management.”](#)

2.10.3 CHAIN

A CHAIN verb performs these steps:

1. Any USING parameters are first copied to a safe place. These are made available to the chained program through the CHAINING phrase of the Procedure Division header.
2. All active programs are made inactive.
3. A CANCEL ALL is implicitly executed.
4. A new program is loaded using the same rules that apply to the CALL verb. This becomes the main program of the new run unit.

For more information see the entry for [CHAIN Statement](#) in Book 3, *Reference Manual*, section 6.6.

2.10.4 Alternate ENTRY Points

The ENTRY statement is used to establish an alternate entry point in a program. Such alternate points can occur many times in a single program. To activate a program at the ENTRY point, a CALL statement from a calling program is used. See “[CALL](#)” in section 2.9 for details.

See Book 3, *Reference Manual*, section 6.6, “Procedure Division Statements,” for complete syntax diagrams and rules regarding the **CALL Statement** and **ENTRY Statement**.

2.11 Reducing the Size of the Runtime

For a variety of reasons including limited system memory or concerns about runtime system overhead, it is sometimes necessary or desirable to reduce the size of the runtime by removing unused elements. This is accomplished by modifying the special file “config85.c” and relinking the runtime. Of course, any component removed from the runtime is no longer supported by the system. Programs that attempt to use those components may terminate in unexpected ways.

You can find the file “config85.c” in the “lib” subdirectory of your ACUCOBOL-GT installation. Instructions on relinking the runtime, as well as specifications for the C compiler that is required for relinking on each platform, are described in section 6.3.5 of *A Guide to Interoperating with ACUCOBOL-GT*.

“config85.c” supports the removal (or re-inclusion) of the following runtime components:

- source debugger
- SORT/MERGE support
- extended exponent support
- Screen Section support
- charting support
- AcuServer client support
- AcuConnect client support
- AcuSQL support

Comments in “config85.c” describe how to modify the file and indicate the approximate reduction in space that will result from the removal of each component.

2.12 acushare Utility Program

The **acushare** utility program is included on most UNIX and Linux systems. On those systems it provides three key services for deployments that use the ACUCOBOL-GT runtime system and/or AcuServer. These services include:

- ACUCOBOL-GT runtime license management
- ACUCOBOL-GT runtime shared memory management
- AcuServer license management

acushare's role in the *extend* license management scheme is described in section 8.2.1 of the *Getting Started* book. **acushare**'s role in shared memory management in UNIX/Linux environments is detailed in [section 2.12.1](#). Comprehensive instructions on the use of **acushare** are given in [section 2.12.2](#).

Note that versions of **acushare** are compatible as follows:

- Versions of **acushare** shipped with *extend8* products are compatible with all Version 8.x products but cannot be used with previous versions.
- Versions of **acushare** shipped with pre-*extend8* products can be used with all pre-*extend8* products but not with any *extend8* products.
- You can run both *extend8* and pre-*extend8* versions of **acushare** on the same system concurrently.

2.12.1 Using Shared Memory

In most UNIX and Linux environments, ACUCOBOL-GT supports the ability to have multiple users share the same copy of a COBOL program's object code in memory. This conserves memory and can lead to improved system performance by reducing the amount of memory paging that the system must do.

Note: Use of shared memory is recommended only in cases where there is a problem with excessive swapping due to too many users for the amount of memory in the machine. If you are not experiencing this problem, enabling shared memory will probably not improve performance. If you are having a problem with limited memory and excessive swapping, then the advantage of reduced swapping usually more than offsets the overhead added by using shared memory. Note that the overhead for using shared memory varies from machine to machine.

The UNIX code sharing facility is built on top of the UNIX System V shared memory facility. In order to use this code sharing, your machine must support shared memory in accordance with the UNIX System V Interface Definition (SVID) and must also have shared memory support enabled in its system kernel. Many UNIX and Linux vendors supply machines with shared memory already enabled, but others require that you reconfigure your kernel to use shared memory. Contact your UNIX vendor if you need additional information on this subject.

One easy way to tell if ACUCOBOL-GT supports code sharing on your machine is to check the files that are installed with the runtime system. If you receive a file called **acushare**, then that system has the ability to share code. If you do not receive this file, code sharing is not available on that machine (most likely because that machine does not adequately support shared memory).

To share program code under ACUCOBOL-GT, you must perform the following steps:

1. Install **acushare**.
2. Edit your COBOL configuration file to specify the programs you want to share (see **section 2.12.1.1**).

3. Start the **acushare** program. After it's started, you can use the program to perform other tasks. See **section 2.12.2** for details.

2.12.1.1 Indicating programs to share

Use your COBOL configuration file to indicate which programs you want to share code. By default, no programs share code. To use shared code for all of your programs, add the following line to the configuration file:

```
SHARED_CODE 1
```

This causes all programs to attempt to share code. Every code segment loaded into memory is placed into shared memory until the shared memory area becomes full. If the system runs out of shared memory and the shared code requests start failing, each runtime will have its own copy of the program in its own memory space. The UNIX default for **SHARED_CODE** is "0" (no programs share code).

Because shared memory is a limited resource under UNIX and Linux, you will probably want to restrict the use of shared code to those programs that render the most benefit. This ensures that other programs do not unnecessarily use up the available shared memory. To do this, specify in your runtime configuration file each program that you want to share as follows:

```
SHARED_CODE Program1
SHARED_CODE Program2
SHARED_CODE Program3
```

(The program name may also be enclosed in single or double quotes, for example, "Program1" or 'Program2'.) When you are using this method, "*Program1*", "*Program2*", and so forth, specify the PROGRAM-IDs from the programs' Identification Division (note that a program's object file name is *not* used). If you use this method, setting SHARED_CODE to "1" has no effect.

To maximize the benefits of code sharing, begin by restricting the use of shared code to large programs that have many users. Later, if you find that you have enough shared memory in your system, you can extend its use to small programs that have many users. Use **acushare**'s reporting facility to help you optimize the use of shared memory.

2.12.2 Using acushare

On most UNIX and Linux systems, the **acushare** utility program is provided to handle ACUCOBOL-GT runtime and AcuServer license management, as well as shared code segments used by the ACUCOBOL-GT runtime. On these systems, **acushare** runs as a background server process that responds to requests from various client runtimes (a “daemon” in UNIX terminology). See section 8.2.1 of the *Getting Started* book for a description of **acushare**’s role in *extend* license management. See [section 2.12.1](#) for a description of **acushare**’s role shared memory management.

acushare has several command-line formats. They include:

```
acushare -start [-f] [ -p portnumber ] [ -e errorfile [ -g ] ]
acushare -kill
acushare -clean
acushare -version
acushare
```

These formats are described in the following sections.

2.12.2.1 acushare -start

The command for starting **acushare** is:

```
acushare -start [ -p portnumber ] [ -e errorfile [ -g ] ]
```

A successful start creates a background process that handles license management and shared code. You can specify:

```
acushare -start -f ...
```

to have **acushare** run in the foreground.

By default, **acushare** obtains a port number (for the *listening port*) from the operating system. If you want to direct **acushare** to listen on a specific port, you can include the “-p” option followed by the desired port number. For example:

```
acushare -start -p 12345
```

If the “-e” option is included, **acushare** error output is appended to the file named after “-e”. If “-e” is not used, error output is sent to /dev/console by default. If output is not allowed on /dev/console, **acushare** attempts to

append to a file named “acushare.err” in the current directory. If that fails, **acushare** prints the message “acushare: cannot open error output file” to standard output and the process exits.

If “-e” is specified, you can optionally use “-g” to cause the error file to be compressed with the gzip compression method. Such files must be manually decompressed with gzip before reading or editing. For clarity and to reduce the risk of confusion or error, it is recommended that you specify a “.gz” extension in the filename. For example:

```
acushare -start -eg acushare_trc.gz
```

Automatic startup at system boot

If you want **acushare** to start automatically when the system boots, you can add a small amount of code to the system boot file. The name of the boot file varies from system to system. Typical names are “/etc/rc.local”, “/etc/brc” or “/etc/rc”. Identify the proper startup file and add lines similar to the following:

```
if [ -f /usr/etc/acushare ]; then
    echo Starting ACUCOBOL-GT shared-code and license daemon > \
        /dev/console
    /usr/etc/acushare -start > /dev/console
fi
```

The preceding example assumes that you’ve placed **acushare** in “/usr/etc”. You will need to adjust the code to match the conventions used in your environment.

2.12.2.2 acushare -kill

To halt **acushare**, simply enter “acushare -kill” on the command line.

Should **acushare** terminate unexpectedly (for example, due to a SIGKILL signal), you should remove the stranded shared memory segment with the “-clean” option before restarting. See [section 2.12.2.3](#).

2.12.2.3 acushare -clean

Should **acushare** terminate unexpectedly, its master shared memory segment may be left behind. Before restarting **acushare**, you should use the “acushare -clean” command to remove the stranded memory segment.

2.12.2.4 acushare -version

This option causes **acushare** to print its version number.

2.12.2.5 acushare (with no options)

If **acushare** is *not* running and you enter the **acushare** command without any options, the following message is displayed:

```
acushare: not running
```

If **acushare** *is* running and you enter the **acushare** command without any options, you will get a detailed report of **acushare** status and usage. The following is the output of the **acushare** server running with two shared programs:

```
ACUCOBOL-GT shared memory and license manager version 8.0.0
(2005-04-18)
Copyright (c) 1993-2008, Micro Focus (IP) Ltd.
```

```
Server Statistics:
```

```
=====
```

```
server PID: 19134
IPC key: 0x01010101
shmid: 40370176
listening port: 44659
start date: Tue Apr 19 03:04:34 2005
clients: 2
client deaths: 0
```

message type	sent	received
-----	-----	-----
HANDSHAKE	8	8
ACK	2	0
INFO	0	2
KILL	0	0
DATA	1	0
LOAD	0	2
ATTACH_FAILED	0	0
LOADED	0	2
UNLOAD	0	0
STATUS_PROGRAM	2	0
USER_ADD	0	2
USER_SUBTRACT	0	0
STATUS_USER	2	0

Shared Program List:

=====

program-id	compilation date	code size	users	shmid
-----	-----	-----	-----	-----
PGM1	Thu Nov 18 10:51:22 2004	32	1	40435719
PGM2	Thu Nov 11 09:45:28 2004	32	1	40402950

shared programs in use: 2

total bytes shared: 64

total bytes saved: 0

Product/License List:

=====

product: ACUCOBOL, SN: real, users: 2/50, processes: 2/4096

terminal: pts/0, user: mark, processes: 1

PID: 21188 (0x42650df9), refs: 2, added: Tue Apr 19 06:56:09 2005

terminal: pts/1, user: mark, processes: 1

PID: 21190 (0x42650e0e), refs: 2, added: Tue Apr 19 06:56:30 2005

2.12.3 acushare errors

The runtime system gracefully handles errors relating to shared code and license management. If the runtime cannot use shared code for some reason (such as running out of shared memory), the runtime simply loads the program into conventional memory and execution continues.

If **acushare** stops running or is stopped while networked runtime processes are active, the runtime issues a warning message, alerting the user to restart **acushare**. If the runtime later detects that **acushare** has not been restarted, the runtime exits.

By default certain errors cause a warning message to be displayed in a message box, which requires a response from the user. You can change this behavior by setting the **LICENSE_ERROR_MESSAGE_BOX** runtime configuration variable to “0”. This will send error messages to the error output (stderr, or an error file if specified). These messages include:

“Error sending message to acushare”

The system has returned an error of an unknown nature when it tried to send a message to **acushare**. For shared memory, execution usually continues. If you kill **acushare** after some processes have attached to shared memory, those processes continue to use shared memory but new processes use conventional memory.

“License manager (acushare) is not running”

This is a one-time message warning that the product will exit soon if **acushare** is not restarted immediately. It indicates that a multiple-user license file is in effect, and the product has detected that **acushare** has been stopped and not restarted.

“Shared memory and license manager (acushare) is not running”

This indicates either: (1) code sharing has been requested (with the SHARED_CODE configuration entry), but cannot be implemented because **acushare** is not running, or (2) a multiple-user license file is in effect, and a runtime process cannot register itself with **acushare** because **acushare** is not running. After outputting the message, the runtime exits.

“The license manager (acushare) has been killed and restarted. You have exceeded the licensed number of users for ACUCOBOL-GT. If you would like to add users, please contact your Customer Service representative.”

This message indicates that a process detected that **acushare** has been stopped and restarted, so the product attempted to re-register itself. However, it could not register itself, either because the maximum number of users has already been reached, or the maximum number of processes has already been reached.

“You have exceeded the licensed number of users for ACUCOBOL-GT. If you would like to add users, please contact your customer service representative.”

A new process cannot be registered with **acushare**, either because the maximum number of users has already been reached, or because the maximum number of processes has been reached.

Note: If there are no shared memory identifiers, **acushare** aborts and prints the following error message:

“acushare: cannot create shared memory”

This message indicates that you do not have enough shared memory configured in your system. Either your UNIX kernel does not have the resource configured, or all of the resources are in use by other programs. In either case, you should regenerate your UNIX kernel for more shared memory. See your UNIX system documentation.

2.13 General Preprocessor Interface

ACUCOBOL-GT includes a general preprocessor interface that allows you to connect preprocessors to the ACUCOBOL-GT compiler. A preprocessor is a program designed to take application source code and *preprocess* it before compilation into an executable object. In the case of ACUCOBOL-GT, the conforming preprocessors should be designed to translate source code written in a variety of COBOLs into ACUCOBOL-GT.

The preprocessor interface in ACUCOBOL-GT has two general functions: (1) To invoke a preprocessor from the compiler and then have the compiler process the resultant “expanded” COBOL; and (2) to signal the compiler that there are fatal errors in the preprocessor and therefore the compiler should not be run.

The interface is designed to accommodate a wide variety of preprocessors and to work on every platform supported by ACUCOBOL-GT. It is designed to call preprocessors as *chained* programs (instead of linking them into the ACUCOBOL-GT compiler) for three reasons:

1. You can develop and debug preprocessors without access to the ACUCOBOL-GT compiler.
2. You can invoke preprocessors directly from the command line.
3. The stand-alone design gives much better portability among platforms.

The general preprocessor interface can be used with any preprocessor that conforms to the requirements of the interface. Most preprocessors will be written in ACUCOBOL-GT or C/C++, but preprocessors can be written in any language that produces an executable object file that can read and write text files and accept command-line arguments.

The specific preprocessors that will be used with ACUCOBOL-GT will be written by you, the user. The ACUCOBOL-GT general preprocessor interface supports many features that you can build into your preprocessor to best handle the job at hand. They include:

- a set of options that the compiler automatically forwards to each preprocessor
- command-line preprocessor options that allow you to specify certain preprocessor parameters at the time of its invocation
- a set of line and file directives that are used to pass special information to successive preprocessors, when two or more preprocessors are used in a chain
- support for preprocessor-generated one-line error messages

The use of conforming preprocessors is described in [section 2.13.1](#), and how to write conforming preprocessors is discussed in [section 2.13.3](#). The use of the AcuSQL pre-compiler is discussed in [section 2.13.2](#).

Source code for a sample preprocessor written in ACUCOBOL-GT is available for your reference on the Web at <http://supportline.microfocus.com/examplesandutilities/index.asp>. Select **Acucorp Samples > General Programming Techniques > ansi2term.zip**.

2.13.1 Use of Preprocessors

This section discusses various aspects of the use of preprocessors, including rules that apply to the command-line syntax for calling preprocessors. These aspects of use are presented in the context of two sample preprocessors, one written in ACUCOBOL-GT, the other written in C/C++. They include information on calling more than one preprocessor with the same command-line statement, command-line options forwarded to preprocessors by the compiler, and calling a preprocessor without the compiler.

2.13.1.1 Calling a preprocessor

The compiler option “-Pg” is used to call a preprocessor written in any language that produces a directly executable object file:


```
-Pg "preproc-1 [options]"
```

Here, *preproc-1* is the name of the preprocessor's executable file. The compiler looks for this file first in the directory in which the compiler executable is located. We recommend that you place preprocessors in that directory. If you place them elsewhere, you need to provide a complete path specification. Under Windows, the compiler appends the extension .EXE automatically (if it is not already there).

The name of the preprocessor, and of any directories in its path, cannot contain spaces. Note that Windows creates a space-free alias for any name with embedded spaces. Every file and directory under Windows has an "MS-DOS" name and extension (maximum eight character name with a three character extension; the name can be viewed in the Properties dialog by right-clicking on the file in the File Explorer). For example, the MS-DOS name and extension for `acuodbconfig.dll` is `ACUODB~1.DLL`. The MS-DOS name does *not* contain spaces, even if the original name does. The MS-DOS name and extension can be used by any Windows command or application in lieu of the original name and extension.

If the preprocessor requires or permits special options that are not specified by the ACUCOBOL-GT interface, they must follow the preprocessor executable file name and must be separated from it, and from each other, by spaces or tabs.

If a preprocessor option contains embedded spaces, it must be enclosed in delimiters so that it will be recognized as a single option. Standard double quotation marks ("...") cannot be used for this purpose because they are used to enclose the preprocessor executable filename and its associated options. Therefore, two apostrophes ('...') are used instead.

For example, the following option might be used to call a preprocessor:

```
-Pg "preproc -d 'x=:the quick brown fox' "
```

Preprocessor written in ACUCOBOL-GT

Preprocessors written in ACUCOBOL-GT use extension ".ACU" by default and are not directly executable. In this case, options of a slightly different form are required:

```
-Pg "runtime preproc-2.ACUC [options]"
```

runtime is the name of the ACUCOBOL-GT runtime executable file. By default, under Windows it is “wrun32”, and under UNIX it is “runcbl”. A path is usually not required because this file is normally installed in the same directory as the compiler’s executable file.

preproc-2.ACU is the name and extension of the preprocessor, i.e., your ACUCOBOL-GT object file. The extension “.ACU” is required; it will not be added automatically. The runtime will look for this file in the current directory. If the file is elsewhere, it is necessary to include a complete path specification.

Rules regarding the use of options with this type of preprocessor are the same as for other preprocessors.

2.13.1.2 Calling two or more preprocessors

If you want your program to call two or more preprocessors sequentially, you must separate them with *vertical line* (or *pipe*) characters (“|”):

```
-Pg "preproc1 [options] | preproc2 [options] ... "
```

The preprocessors are called in the order in which they appear in this list. The first preprocessor accepts its input from the source file submitted to the compiler. Each subsequent preprocessor accepts as its input the output produced by the previous preprocessor. The output of the last preprocessor is then compiled by the compiler.

Although the *pipe* notation suggests the use of UNIX-type pipes, the compiler actually stores the output of each preprocessor in a temporary file for use as input to the next preprocessor. The temporary files are erased when the compiler is finished with them.

Note: The temporary file used by the *n*-th preprocessor for its output is called “acu_pp#.out”, where “#” is a decimal representation of the number *n*. It is in the current directory.

This method of calling multiple preprocessors is incompatible with the standard method of calling the AcuSQL pre-compiler (with the “-Pn” or “-Ps” option). If the AcuSQL pre-compiler is to be called in conjunction with other preprocessors, it must be called with the “-Pg” option. See **section 2.13.2, “AcuSQL Pre-compiler.”**

2.13.1.3 Compiler options forwarded to preprocessors

The following compiler options are automatically forwarded to each preprocessor, including the AcuSQL pre-compiler:

- Sa** indicates that the input file is in ANSI format
- Sd** directs the preprocessor to include debugging lines
- Si** directs the preprocessor to include source lines according to a *pattern* in the Identification Area of the source code. The next separate argument is the *pattern* to match.
- St** indicates that the input file is in terminal format COBOL
- Sx** directs the preprocessor to exclude source lines according to a *pattern* in the Identification Area of the source code. The next separate argument is the *pattern* to match.
- e *file*** directs the preprocessor to write error messages to specified file (if not specified, error messages are directed to standard error output)

Note: Where some options are mutually exclusive (“-Sa” and “-St”), the compiler command line will include at most one of those options. That one will be forwarded to the preprocessor.

Every preprocessor (written to the standards of the ACUCOBOL-GT General Preprocessor Interface) honors the “-e” option. Every preprocessor accepts the other options, although it may not make use of them.

Note: A preprocessor *honors* an option if it does what the option requires. A preprocessor *accepts* an option if it tolerates its presence on its command line but does not necessarily honor it. An option that is not honored or accepted causes a preprocessor to generate an error message.

Refer to section 2.1 for a description of all compiler options.

2.13.1.4 Calling a preprocessor without the compiler

Every preprocessor can be called as a separate application, without the ACUCOBOL-GT compiler, by a command line of one of the following forms (the second example is used with a preprocessor written in ACUCOBOL-GT):

```
preproc-1 -Po output-file [options] input-file
```

```
runtime preproc-2.ACU -Po output-file [options] input-file
```

These command lines are very similar to the one used with the “-Pg” option, but there are some important differences:

- The preprocessor executable, or the ACUCOBOL-GT runtime if the preprocessor is an ACUCOBOL_GT program, will be located according to the procedures used by the operating system for locating executables.
- The input and output files must be specified as shown.
- The option list must include any compiler options that the compiler would normally pass through to the preprocessor (see [section 2.13.1.3, “Compiler options forwarded to preprocessors”](#)).
- Options that might be passed to the compiler in combined form must be separated for the preprocessor. For example, the compiler accepts “-Six” for include/exclude, while a preprocessor requires: “-Si” for include and “-Sx” for exclude.

For diagnostic purposes, a preprocessor also accepts either of the following two options:

- help** do not preprocess; show command-line help on standard output device
- v** do not preprocess; show preprocessor version information on standard output device

When either of these options is used, the input and output files need not be specified.

Note: The “-help” and “-v” options are not acceptable when the preprocessor is called by the compiler.

Some preprocessors may allow other combinations of options and files when called without the compiler.

The input and output files should be files, not devices, because the preprocessor may not read or write them in strictly sequential order.

A preprocessor returns an exit value “0” (zero) if the preprocessing was successful and the value “1” (one) if there were errors.

2.13.2 AcuSQL Pre-compiler

The AcuSQL pre-compiler is a preprocessor for programs with embedded SQL. AcuSQL is offered by Micro Focus as a separately licensed product. You can invoke the AcuSQL pre-compiler from the ACUCOBOL-GT command line.

The AcuSQL pre-compiler is discussed in detail in a separate book, *AcuSQL User's Guide*. This section gives insight into the AcuSQL pre-compiler's compatibility with the ACUCOBOL-GT general preprocessor interface and other conforming preprocessors, and discusses calling the pre-compiler from the ACUCOBOL-GT compiler via the general preprocessor interface. This information is especially useful to those who want to use the AcuSQL pre-compiler in combination with another preprocessor.

2.13.2.1 Compatibility with ACUCOBOL-GT general preprocessor interface

The AcuSQL pre-compiler is compatible with the ACUCOBOL-GT general preprocessor interface. It can be used in its standard fashion if it is the only preprocessor being invoked, or it can be used with other preprocessors with some minor adjustments.

2.13.2.2 Calling the AcuSQL pre-compiler

You can start and use the AcuSQL pre-compiler from the compiler by using the “-Ps” compiler option. For example:

```
ccbl -Ps [options] input_filename
```

Where *options* are AcuSQL commands and *input_filename* is the source file to be pre-compiled and compiled. See Chapter 3 of the *AcuSQL User's Guide* for names and descriptions of the pre-compiler options. Note that the AcuSQL User's guide also discusses other ways to run AcuSQL and the reasons for doing so.

Note: When the AcuSQL pre-compiler is called by the compiler along with other preprocessors, “-P” options other than those specified in the *AcuSQL User's Guide* are not recognized and should not be used.

2.13.3 Writing a Preprocessor

This section presents information on how to write a preprocessor that conforms to the ACUCOBOL-GT general preprocessor interface. You may choose to write a preprocessor in any of a number of languages and architectures, so we limit our discussion here to the general *features*, including syntax examples, that are relevant to any conforming preprocessor. This section contains subsections on using command-line options, line and file directives, and error messages in your preprocessors.

2.13.3.1 Command-line options

A preprocessor is written as though it were always called without the compiler by a command line. In particular, every preprocessor must accept at least the following command-line options:

- e *file*** causes error messages to be written to the specified file. This option must be followed by the name of the error file.
- help** do not preprocess; show command-line help on standard output device

- Po file** causes preprocessor output to be written to the specified file. This option must be followed by the name of the output file.
- Sa** indicates that the input file is ANSI format COBOL
- Sd** instructs the preprocessor to include lines marked as debugging lines (“D”) in the indicator area
- Si** instructs the preprocessor to include lines based on *pattern*. The next separate argument is the *pattern* to match.
- St** indicates that the input file is terminal format COBOL
- Sx** instructs the preprocessor to exclude lines based on *patter*. The next separate argument is the *pattern* to match.
- v** do not preprocess; show preprocessor version information on standard output device

A preprocessor should accept these options in any order. A preprocessor must honor the “-e”, “-Po”, “-help” and “-v” options. Other options need not be honored but must be scanned if they appear on the command line.

Note: A preprocessor *honors* an option if it does what the option requires. A preprocessor *accepts* an option if it tolerates its presence on its command line but does not necessarily honor it. An option that is not honored or accepted causes a preprocessor to generate an error message.

Another command-line option is highly recommended:

- n** indicates to the preprocessor that no preprocessor directives should be included in output. A standard preprocessor output includes directives, which may make the output very difficult to read. (See [section 2.13.3.2, “Line and file directives,”](#) for more on directives.) If you want to examine the output of a preprocessor, this option lets you suppress the directives and receive *clean* output.

The preprocessor should abort with an appropriate error message if a required option is missing.

2.13.3.2 Line and file directives

Line and file directives are comment strings that a preprocessor uses to pass information to the next preprocessor in the sequence. The information may include directory paths and filenames for the following preprocessor to access or specific instructions to be carried out. (For more information related to directives, see [section 5.3.2](#) and [section 5.3.3](#).)

Every preprocessor that follows the first one invoked must scan its input for directives, and every preprocessor must put directives at the appropriate places in its output.

A directive always begins with an asterisk (“*”), which is placed in column 7 if the file is in ANSI format or in column 1 if the file is in Terminal format. The asterisk indicates a COBOL comment. A preprocessor should accept directives in either ANSI or Terminal format.

A directive of the following form indicates that subsequent input lines came from the specified source code file:

```
*(( PREPROC PNAME FILE "<file specification>" ))
```

A preprocessor should replace the letters “PPNAME” in a directive with its own name (or any other name containing at most six alphanumeric characters). This field is used only for diagnostic purposes.

The *file specification* must be abbreviated if necessary, so that the directive will fit into a line of ANSI format without encroaching on the Identification Area.

Note: In the ANSI format, code is limited to columns 1-72. Everything beyond this is part of the Identification Area and is not used, except for conditional compilation. If the directive is too long, it will extend into this area, possibly triggering spurious conditional compilation.

Note: Examples in this section show single spaces between items in directives. Actually, any reasonable number of spaces is acceptable.

The first preprocessor writes a directive of the preceding kind at the very beginning of its output file. Each subsequent preprocessor will read a directive of this kind at the very beginning of its input file and will write it at the beginning of its output file with only the preprocessor name changed.

Directives of this kind also appear whenever a preprocessor honors a COPY statement or its equivalent in other languages. There will usually be one such directive at the beginning of the copied code and another at its end.

Within each source file, the lines are numbered consecutively, beginning with line 1. If a preprocessor always produced one line of output for each line of input, it would need no other directives for line numbers. However, that is not usually the case and most preprocessors do need other directives.

A directive of the following form indicates that, until line numbering is changed by a subsequent directive, every line that follows came from the line whose number is embedded in the directive.

```
*(( PREPROC PNAME LINE BEGIN <line number in decimal> ))
```

Normally, the embedded line number will be the number of the next line in the current source file. However, it might be larger if a previous preprocessor generated nothing from one or more lines in the source file.

When there are two or more lines following this directive, it is presumed that *all* of them were generated from the same line of source code. Although this is not always true, it is a necessary convention because preprocessors that do a lot of parsing and translation cannot always assign a specific source code line to each line of output.

A directive of the following form restores regular line numbering; that is, it indicates that the first line following the directive came from the line after the one whose number is embedded in the directive, and that until line numbering is changed by a subsequent directive for the same source file, every subsequent line came from the line after the previous line.

```
*(( PREPROC PNAME LINE END <line number in decimal>))
```

These two directives normally come in matching pairs (although this is not required). For example, the source file may contain the following code (line number in parentheses):

```
(55) display "Making connection".
```

```
(56) EXEC SQL CONNECT TO :dsn-name as C1
(57)     END-EXEC.
(58) display "Connection made".
```

The output may be as follows:

```
display "Making Connection".
*(( PREPROC ACUSQL LINE BEGIN 56 ))
    PERFORM CALL "SQL$START" END-CALL CALL "SQL$CONNECT" USING
    dsn-name 'C1' END-CALL IF SQLCODE OF SQLCA < 0 THEN GO TO
    Error-Exit END-IF END-PERFORM
*(( PREPROC ACUSQL LINE END 57 ))
    display "Connection made".
```

Preprocessors that process COBOL code may also use two other directives to indicate places where the source code format (ANSI or Terminal) may change.

The following directive indicates that subsequent lines may be in a different format because they were taken from a COPY file (or its equivalent in modified COBOL):

```
*(( PREPROC PPNAME INCLUDE BEGIN "<file specification>" ))
```

The following directive indicates that the code in the new format has ended and the format reverts to the one that prevailed before the matching INCLUDE BEGIN directive.

```
*(( PREPROC PPNAME INCLUDE END "<file specification>" ))
```

The file specification must be abbreviated, if necessary, so each directive will fit into a single line of ANSI format without encroaching on the Identification Area.

In the above examples the INCLUDE directives are necessary to tell the compiler that the format may have changed, while the FILE directives are necessary to tell the compiler that a new file has begun and error messages should refer to it.

Note: In the ANSI format, code is limited to columns 1-72. Everything beyond this is part of the Identification Area and is not used, except for conditional compilation. If the directive is too long, it will extend into this area, possibly triggering spurious conditional compilation.

Preprocessors that do not distinguish between COBOL formats should pass such directives along.

2.13.3.3 Error messages

Each error message produced by a preprocessor must be one line long and must include the source file and line number, as in the following examples:

```
"myprogram.cbl, line 31: Include file not found at END-EXEC."  
"myprogram.cbl, line 31: parse error at END-EXEC."
```

The comma following the source file name, the word "line," and the colon following the line number are required. They are used by the compiler and other software to parse the error message.

Error messages produced by preprocessors other than the first one invoked must use special means, described in **section 2.13.3.2, "Line and file directives,"** to identify the original source file and line that contained the erroneous code. You should abbreviate the source file specification if it is too long to be read easily.

3

Debugger and Utilities

Key Topics

Runtime Debugger	3-2
Object File Utility — cblutil	3-58
Vision File Utility — vutil	3-66
File Transfer Utility — vio	3-98
Indexed File Record Editor (alfred)	3-107
logutil	3-107
The Profiler	3-111
External Sort Utility — AcuSort	3-121
Remote Preprocessing Utility — Boomerang	3-135

3.1 Runtime Debugger

This chapter describes how to use the ACUCOBOL-GT runtime debugger and other utility programs supplied with ACUCOBOL-GT.

runcbl contains a built-in source-level debugger. This debugger runs in a window that overlays the screen so that the active program is not disturbed.

In all environments, the runtime debugger interface contains a menu bar and command window. To navigate through source code in character environments, use the “Up” and “Down” menu items. You can also use the arrow keys and Page Up and Page Down keys to move through the code.

```

File View Run Source Data Breakpoints (Selection) Up Down
78 AREGEXP-NONE VALUE 0.
78 AREGEXP-BASIC VALUE 1.
78 AREGEXP-WINDOWS VALUE 2.
78 AREGEXP-POSIX VALUE 3.

* end of acucobol.def

procedure division.
main-logic.
@ perform initialization.
accept label-font from standard object "large-font".
display standard graphical window, background-low.

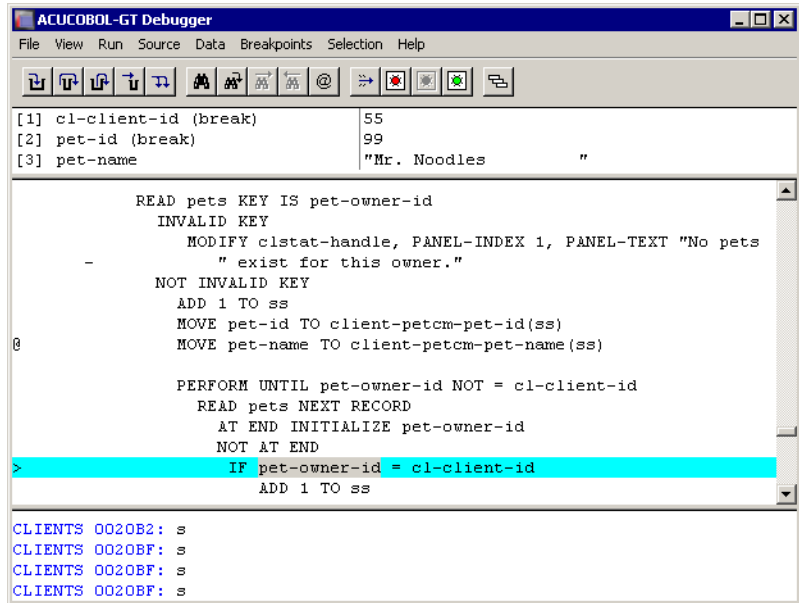
display label "Sample Lines", line 3, col 35,
color magenta, font label-font.

* Display the magenta lines around "Sample Lines" label.
display bar, line 3.0, col 8, size 23, width 2, color magenta.
----- ACUCOBOL-GT Debugger -----
Running all threads
TEST1 000005:

```

The Runtime Debugger (UNIX)

In Microsoft Windows environments, the debugger also contains a toolbar. When you perform full source debugging in Windows, a scroll bar appears to the right of the source, offering an easy way to scroll through the code.



The screenshot shows the ACUCOBOL-GT Debugger window. At the top is a menu bar with 'File', 'View', 'Run', 'Source', 'Data', 'Breakpoints', 'Selection', and 'Help'. Below the menu is a toolbar with various icons for debugging operations. The main window is divided into three sections:

- Breakpoint List:** A table with three rows:

[1] cl-client-id (break)	55
[2] pet-id (break)	99
[3] pet-name	"Mr. Noodles"
- Source Code Editor:** Contains COBOL code. The line `IF pet-owner-id = cl-client-id` is highlighted in cyan. A vertical scroll bar is on the right side of the editor.
- Output Window:** Shows the following text:


```
CLIENTS 0020B2: s
CLIENTS 0020BF: s
CLIENTS 0020BF: s
CLIENTS 0020BF: s
```

The Runtime Debugger (Windows)

You can run the debugger at any time, but in order to reference the program's symbols by name, or view the source code, you must have compiled the program with some special options.

The runtime debugger supports three modes of operation: source debugging, symbolic debugging, and low-level debugging.

Source Debugging

At the development stage, *source debugging* is the most useful, because it allows you to view the source code while you are debugging. To use source debugging, compile the program with the `"-Gd"` or `"-Ga"` option. Because these compiler options cause all of the source code to be bundled with the object code, you'll notice that the size of your object code grows considerably.

Note: Although the compiler accepts lines longer than 80 characters in TERMINAL format files, in source debugging mode the debugger does not display characters past the 80th column. If possible, use the AcuBench integrated debugger instead.

Symbolic Debugging

Symbolic debugging does not allow you to view the program source, but does allow you to reference paragraphs and variables by their COBOL identifiers. The advantage to using symbolic debugging rather than source debugging is that the compiled object module is much smaller. This may be useful if disk space is very tight. Some application developers compile their programs with symbolic debugging for delivery to clients in order to facilitate the resolution of client questions over the phone. You must compile the program with the “-Gy” or “-Gs” option to use symbolic debugging.

Low-Level Debugging

Low-level debugging is available at any time *even if the program was not compiled with any debugging options*, but you must use absolute addresses to access variables, so you’ll need a listing of your program. Low-level debugging is convenient when you’re debugging a data-dependent problem on a client’s machine, if the client does not have a debug-version of your program. The “Trace Files” command described in **section 3.1.4** operates in this mode. “Trace Files” is particularly useful for tracking data-specific problems in complex applications.

Debugging in background mode

If your ACUCOBOL-GT programs are called from programs written in other languages, or if you are running in an environment that includes an application server or OLTP software, you likely have programs running in background mode (executed with the “-b” flag). Complete instructions for debugging programs running in background mode is available in Chapter 9 of *A Guide to Interoperating with ACUCOBOL-GT*.

The Abend Diagnostic Report

When a program experiences an abnormal shutdown, running in debug may not reveal the source of the problem. In such cases, the ACUCOBOL-GT runtime can produce a report to show the state of the program at the moment of termination. This Abend Diagnostic Report, or ADR, can help you to analyze the cause of an abnormal shutdown.

More information about this report is included in [section 3.1.9, “Using the Abend Diagnostic Report \(ADR\).”](#)

3.1.1 Entering the Debugger

When a program is executed in debug mode, the debugging window pops up over the lower portion of the screen. Commands to the debugger and their results are displayed in this window. You can control the size of the command window from within the debugger by pulling down the Source menu and selecting Window Size.

If you are running the debugger under Windows, you can change the size of the entire debugger window. Point to a border or corner, and when the mouse pointer changes into a double arrow, hold the mouse button down and drag the border or corner to reach the size you want. Release the button when you are ready.

You can enter the debugger in several ways; the most common is to specify the “-d” option to **runcbl**. Here’s a list of all the ways the debugger can be entered initially:

- When you specify the “-d” option to **runcbl**. This causes the program to start in the debugger. For example:

```
runcbl -d payroll
```

- Whenever a STOP statement executes that is not a STOP RUN. In this case, the argument to STOP is displayed in the debugging window. This method functions even if **runcbl** is not run in debugging mode. Note, however, that symbols and source will not be available in this case. To do source level debugging, compile with the “-Gd” or “-Ga” option and run with the “-d” option.

- When the program has been started in debugging mode, and the abort key (such as **Ctrl + C**) is pressed. On Windows systems, the same effect is achieved by selecting the “Enter Debugger” menu option. In either case, when the command is received, the program finishes execution of the current instruction and enters debugging mode. Note that if the current instruction is an **ACCEPT** statement, the program will not enter debugging mode until the **ACCEPT** statement is satisfied by having something entered. Using the abort command to enter the debugger does not work on all machines.

If you have already entered the debugger, you may reenter it in one of the following ways:

- When a breakpoint is reached. Breakpoints are set by the user through the debugger.
- When the program is being “stepped” through by the debugger and the step count has been reached.
- When a variable that is being monitored changes. In this case an automatic breakpoint is generated at the beginning of the next statement.
- When you’ve compiled with “-Za” along with “-Gd”, and an array violation occurs. In this case, you automatically break to the debugger and see the line on which the array violation occurred.

Each of the situations described above causes the debugging window to pop up over the lower portion of the screen. If source-level debugging is being used, then the upper half of the screen displays the source at the location currently being executed. When the debugger exits, these windows are removed and the application screen is restored. The application screen is not restored, however, until an **ACCEPT** or **DISPLAY** verb is executed. This allows you to debug a section of code without the distraction of having the screen being constantly repainted.

3.1.2 Cursor and Mouse Handling in Source-level Debugging

In source-level debugging, the entire source code is available for viewing. An “@” sign is displayed in column one of the *current line* (the line of code that’s being executed). The line containing the cursor shows a “>” sign in column one. (If the cursor is on the current line, then the cursor is hidden by the “@” sign.) For terminals that support reverse video, the cursor line is highlighted. Use the arrow keys to move the cursor. Press F10 to access the menu bar and to toggle back to source code from the menu bar.

If your runtime offers mouse support, then you may use a mouse in the area of the screen that displays the source code. The mouse allows you to perform the following common actions:

Move the cursor line	To move the cursor to a different line, simply click anywhere on the line you want.
Scroll the source	To scroll the source up or down, hold the mouse button down and move the mouse off the top or bottom edge of the source window. The source will scroll to track the mouse. The source scrolls slowly, to make it easy to adjust the current display by a small amount.

Highlight a variable or procedure	<p>Point the mouse at a variable or a procedure name and click on it to highlight the name and enable the “Selection” entry on the menu bar (discussed below). Several operations are available under “Selection” that act on the highlighted item. The highlighted item will also become the default value used by many menu options.</p> <p>Using variables, you may specify data names that require arguments, such as tables that require indexes. You cannot specify literals.</p> <p>You can use the mouse, F7 (display variable on current line), or the Tab key (highlight variable on current line) to view qualified and indexed data items in the source. As long as a variable and all of its qualifiers and indexes are on one line, the entire expression is evaluated by these keys. If a variable and all of its qualifiers and indexes span multiple source lines, the entire expression is ignored, but component items are still found.</p>
Display a variable	<p>To view the value of a variable, double-click on that variable.</p>
View procedure	<p>To scroll quickly to a paragraph or section, double-click on its name.</p>
Run to desired line	<p>To set a temporary breakpoint, double-click on a verb. This establishes a temporary breakpoint at the line containing the verb. The program runs to that line (unless it encounters another breakpoint before it reaches the line). When it reaches a breakpoint, the runtime returns to the debugger prompt and awaits your next command.</p>

3.1.3 Debugger Commands

Debugger commands are displayed in a menu bar with pull-down submenus, and on the debugger’s toolbar. Commands can be selected either from the menus or from the keyboard. A menu item that is followed by three dots (such as “Accept...”) requires a value. You are prompted for the value unless you highlight it within the source code before you choose the option. Some,

but not all, commands may be selected from the toolbar. You can determine toolbar functions by placing the mouse over a button and holding it there for a brief period.

If you do not have a mouse, use the F10 to access the debugger menu bar. Then use the arrow keys to move within the menu system. Press **Return** or **Spacebar** to make your selection. Typing the *key letter* is another way to make a selection, if key letters are available on your system. From the menu bar, press **F10** to toggle back to the debugger command line.

On systems such as Windows that include a System Menu in the Debugger window, you can activate the System Menu by pressing the function key F9. F9 also activates the System Menu of any window displayed over the Debugger window.

The debugger displays the first ten characters of the name of the current program, followed by the current address (in hexadecimal). This name is derived from the PROGRAM-ID in the Identification Division of the source code.

The commands described on the following pages may be used in all debugging modes, unless marked with one or two asterisks.

- One asterisk (*) indicates that the option is available in source-level debugging only (“-Gd” compiler option).
- Two asterisks (**) indicate that the option is available in either source-level or symbolic-level debugging (“-Gd” or “-Gy” compiler option), but not in low-level debugging.

Keep in mind that you must compile with “-Gd” or “-Gy” in order to reference variables *by name*. If the program was not compiled with one of these options, refer to each variable by its *absolute address* as shown in a program listing.

The tables below list all debugger commands available through the keyboard, with their menu equivalents given in parentheses. The same listing is accessible through the H (Help) debugger command.

3.1.3.1 Source-level commands

Command	Menu Option	Description
<F1> or <Page Up>		Scrolls source up one page
<F2> or <Page Down>		Scrolls source down one page.
<F3>	Run/Go to Cursor Line	Sets a temporary breakpoint at the current cursor line and continues execution of your program.
<F4>	Breakpoints/Togg le at Cursor Line	Sets or removes a breakpoint at the source line containing the cursor.
<F5> or <Up Arrow>		Moves the source cursor up one line.
<F6> or <Down Arrow>		Moves the source cursor down one line.
<F7>		Causes the cursor line to be searched for program variables. If one is found, its name and current contents are displayed.
Tab		Search the current line for selectable text. If selectable text is found, select it.
@!	Run/Skip to Cursor Line	Moves the current program location to the line containing the cursor.
F	Source/Repeat Find	Repeats the last Find command, starting at the current cursor line.
FB <i>text</i>	Source/Find Backwards	Locates <i>text</i> in the program's source code. The debugger searches backwards from the current cursor line.
FF <i>text</i>	Source/Find Forward	Locates <i>text</i> in the program's source code. The debugger searches forward from the current cursor line.
FT <i>text</i>	Source/Find from Top	Locates <i>text</i> in the program's source code. The debugger starts at the top of the current program source.

Command	Menu Option	Description
VP	View/Perform Stack	Lists all of the nested paragraphs leading up to the current statement, starting from the beginning of the program.
<i>W procedure</i>	Source/Paragraph	Positions the cursor at the procedure you name. The procedure must be located in the current program.
<i>W@</i>	Source/Current Line	Positions the cursor at the current line in your program.
WB	Source/Last Line	Positions the cursor at the last line (bottom) in your program.
WT	Source/Line 1	Positions the cursor at the first line (top) of your program.

3.1.3.2 Other commands

Command	Menu Option	Description
!	File/Shell	Invokes the operating system's command processor, allowing you to enter commands.
!!		Returns the process ID (PID) for the current runtime execution.
<i><script-file</i>	File/Run Script	Runs a script file. Causes all input (debugger and program) to be read from the script. Control returns to the keyboard when the script is finished.
>	File/Stop Recorder	Ends your recording. If you do not end your recording, the script is saved and closed when the debugger closes.
<i>>script-file</i>	File/Record Script	Turns on a recorder that saves all of your keyboard input and menu selections to a file of your choice.
<i>A variable</i>	Data/Accept	Lets you modify the contents of a variable.

Command	Menu Option	Description
B	View/Breakpoints Breakpoints/View	Displays a dialog box with all existing breakpoints. You can add/modify breakpoints from this dialog box.
B <i>address</i> , [<i>skip #</i>]	Breakpoints/Set	Sets a breakpoint with a skip count. The breakpoint will not be activated until it has been hit <i>skip#</i> times.
B <i>address</i> , [<i>skip #</i>], [<i>WHEN cond</i>]	Breakpoints/Set	Sets a breakpoint with a skip count and/or condition. The breakpoint will not be activated unless <i>cond</i> is true # times.
C <i>address</i>	Breakpoints/Clear	Removes a breakpoint. You can enter either the breakpoint's paragraph name or hexadecimal address.
CA	Breakpoints/Clear All	Removes all breakpoints.
CM <i>number</i>	Data/Monitor/Clear	Clears variable monitor <i>number</i> .
CMA	Data/Monitor/Clear All	Clears all variable monitors.
CWA		Clears all variable watches.
D <i>variable</i> [, <i>X</i>]	Data/Display	Shows the contents of a variable. The value is shown in the debugger command window. If <i>X</i> is appended to the display command, the variable is displayed in hexadecimal. If the variable is specified by its <i>absolute address</i> from a program listing, it must be preceded by "." (a period)
D <i>variable(x:y)</i>		Display a reference modified variable. The command "d my-var(2:5)", for example, displays five characters, starting with the second character of the variable string.

Command	Menu Option	Description
E	File/Exit Debugger	Turns off the debugger while continuing the execution of your program.
G	Run/Continue	Resumes execution of your program from its current location.
<i>G address</i>		Sets a temporary breakpoint at address, and continues execution.
GE	Run/Go until Program Exits	Runs your program until the current program exits to its calling program.
GP	Run/Go until Paragraph Returns	Runs your program until the current paragraph returns to the point from which it was performed.
H		Displays the online help files.
L		Displays the name of source paragraph or section which is being executed.
M	View/Monitors Data/Monitor/List	Shows all monitored variables and their values. This also displays a sequence number for each monitor, which is used to clear the monitor.
<i>M variable</i>	Data/Monitor/Set	Causes the program to stop whenever the named variable changes its value. The variable is shown in the Watch Window.
P [#]	Step Over	Steps over the next statement. With a count, the program will step count times. Use this command if you want to step through a program following only the original thread.
Q!	File/Quit	Halts your application and exits the debugger.
<i>R script</i>		Run a debugging script. The debugger reads commands from a script (but user-input is gathered normally).

Command	Menu Option	Description
RA [#]	Run/Run all Threads	Toggles or sets the “Run All Threads” setting. If # is 0, only the current thread will run. If # is non-0, all threads will run.
S [#]	Step Into	Executes one statement of your program and then returns control to the debugger. You may follow the command with the number of steps to take. This command will follow a new thread if one is created. If you want to follow the original thread, use the “step over” command (P) described above.
SA	Run/Auto Step	Causes your program to execute “step” commands repeatedly until it reaches the end of the program., or until you stop auto-step by pressing the spacebar while the debugger is active. Like the “step into” command (S), this follows a new thread if one is created.
ST [#]	Run/Thread	Switches to the thread identified by the given number (or the next thread, if no number is given). The “Run” menu displays the number assigned to each threads.
T flush		Causes the error file to be flushed to disk after each write, if you are writing to an error file.
TF [#]	File/Trace Files	Turns on file tracing. The # indicates the level of tracing, from 1 to 9, where 1 is the lowest and 9 is the highest.
TP	File/Trace Paragraphs	Toggles paragraph tracing, which is a listing of all paragraphs and sections entered at runtime.
U	View/Memory Usage	Displays the amount of dynamically allocated memory currently used by the runtime system.

Command	Menu Option	Description
V	View/Screen	Displays your application's current screen. Press any key or click the left mouse button to return to the debugger.
WA	Data/Monitor/Set	Places a variable in the Watch Window. The difference between a watched variable and a monitored variable is that watched variables do <i>not</i> cause program execution to halt when they change.
WS <i>number</i>	Source/Window Size	Specifies the number of lines to show in the command window.
WW <i>number</i>	Source/Watch Size	Specifies the number of lines to display in the Watch Window. The number cannot exceed the number of watched/monitored items.
F8		Recalls the last command entered for editing.
Ctrl + N		Shows the next line in the Watch Window.
Ctrl + P		Shows the previous line in the Watch Window.

3.1.3.3 Multithreading Issues

When a program is running under the debugger, by default the “run all threads” (“RA”) mode is turned on. In this mode, you step through only one thread at a time, but the background threads run normally. If a background thread reaches a breakpoint, it returns control to the debugger and becomes the current thread. The last debugging mode you select is saved into your “.ADB” file, so the default mode applies only when you do not have a “.ADB” file.

You can choose to execute one thread at a time in the debugger. This allows you to trace a thread without interference from other threads. When a new thread starts, the debugger informs you, but continues tracing the parent thread. Use the “ST” (Switch Threads) command to switch between threads.

You can find a list of the current threads under the “Run” menu item. This list shows you the current program and address where each thread is executing. You can select the appropriate menu item to switch to that thread as an alternative to the “ST” command.

A list of all current threads appears at the bottom of the “Run” menu. The list shows both the name of the program associated with the thread and the address where each thread is executing. To switch between threads, you can select a thread from the list as an alternative to the “ST” command.

The debugger can manage up to ten threads simultaneously.

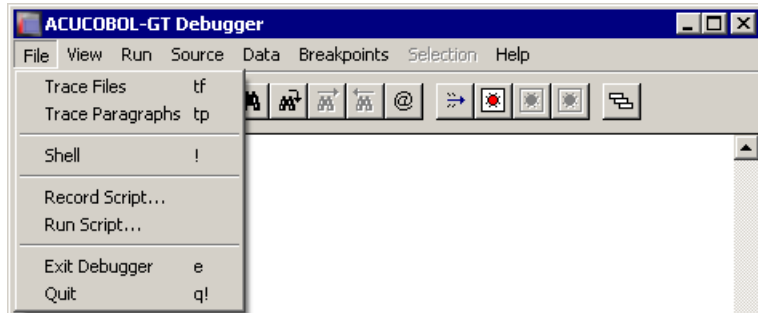
3.1.3.4 Getting help

Under Windows, you can access the online debugger documentation from the Help menu at the far right of the debugger menu bar.

In other environments, access help by typing the letter “H” at the debugger prompt and then pressing Enter.

3.1.3.5 File menu

The *File* menu contains commands relating to the overall operation of the debugger.



The File Menu (Windows)

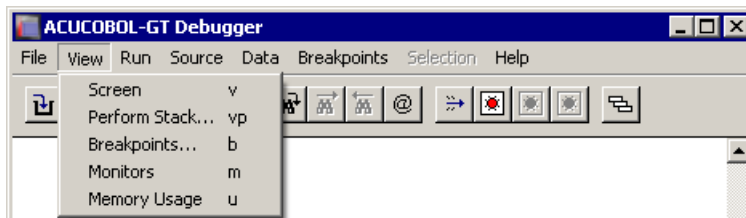
Menu Option	Description
Trace Files	<p>Toggles file tracing on or off.</p> <p>A file trace is a listing of all file operations performed at runtime. Trace output can be tailored with the TRACE_STYLE configuration variable. See Book 4, Appendix H.</p> <p>Trace output is sent to the same place that error output is sent. So, to prevent the trace from overwriting your application's screen, be sure to use the runtime's "-e" command-line option (followed by a file name) to direct error output to a file. See the runtime configuration variable MAX_ERROR_LINES in Appendix H to limit the size of the error file.</p> <p>Some file systems can print extra information if a higher level of tracing is enabled. This extra information is mostly useful to the Technical Support department at Micro Focus, and they may ask you to execute a "tf n" for some integer "n".</p> <p>File trace example:</p> <pre>runcbl -de trace.fil program1</pre> <p>Make sure that the error file you designate (trace.fil in the example above) does not exist in the current directory. If it does, it will be emptied.</p> <p>The keyboard form of this command is "TF [#]".</p>
**Trace Paragraphs	<p>Toggles paragraph tracing on or off.</p> <p>Paragraph tracing is a listing of all paragraphs and sections entered at runtime.</p> <p>A paragraph trace is sent to the same place that error output is sent. So, to prevent the trace from overwriting your application's screen, be sure to use the runtime's "-e" command-line option (followed by a file name) to direct error output to a file.</p> <p>The keyboard form of this command is "TP".</p>

Menu Option	Description
Shell	<p>Pulls up the operating system's command processor, allowing you to enter commands. Shell is not supported for programs running in thin client mode. Attempts to use the Shell command with programs running in thin client mode will result in the error message: "Unable to start shell in thin-client mode".</p> <p>Note: Under the default Windows setup, the command processor will run as a full screen application.</p> <p>The keyboard form of this command is "!".</p>
Record Script	<p>Turns on a recorder that saves all of your keyboard input and menu selections to a file of your choice. Debugger commands and input to the program being debugged are both saved.</p> <p>Play back the recording with the Run Script command.</p> <p>See also the description of the W\$KEYBUF routine in Appendix I.</p>
Stop Recorder	<p>When the recorder is running, the Record Script menu option is replaced by a Stop Recorder option. Use this to end your recording. If you do not end your recording manually, the script information is saved when the debugger closes.</p> <p>While the recorder is active, you will not be able to use the mouse for anything except selecting menu items. Mouse actions are very position-dependent and are often difficult to replay.</p> <p>The recorder can save up to 4096 characters of information. Normal keystrokes use one character. Special keys such as function keys and menu selections typically use up to four characters.</p> <p>The keyboard form of this command is "> <i>script-file</i>". The runtime does not process the filename. To turn off the recorder, use ">" by itself.</p>
Run Script	<p>Runs a debugger script file. Control returns to the keyboard when the script is finished.</p> <p>The keyboard form of this command is "< <i>script-file</i>".</p>

Menu Option	Description
Exit Debugger	Turns off the debugger but continues execution of your program. The keyboard form of this command is “E”.
Quit	Halts your application and exits the debugger. The keyboard form of this command is “Q!”.

3.1.3.6 View menu

The *View* menu contains commands related to viewing and monitoring your program.



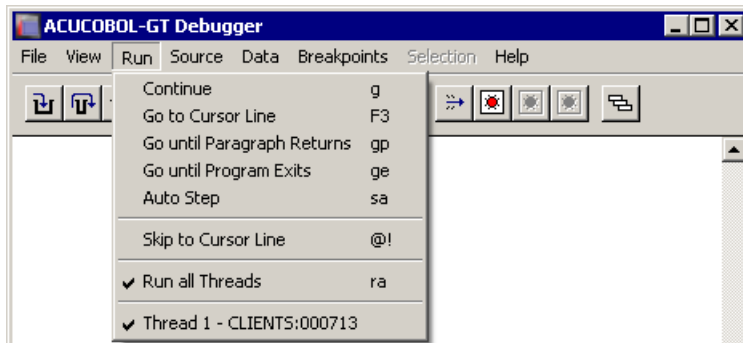
The View Menu (Windows)

Menu Option	Description
View Screen	Displays your application’s current screen. Press any key or click the left mouse button to return to the debugger. The keyboard form of this command is “V”.
*View Perform Stack	Lists all of the nested paragraphs leading up to the current statement, starting from the beginning of the program (or the beginning of the thread, if a new thread was started). Double-clicking on one of the names in the list takes you to that paragraph and highlights the current statement in that paragraph. The trace also accounts for embedded procedures and declaratives. In order to use this command, you must have compiled for source-level debugging (-Gd), and your program must allow for recursive performs (-Zrl). Recursive performs are the default. The keyboard form of this command is “VP”.

Menu Option	Description
View Breakpoints	<p>Displays a dialog box that lists all of your breakpoints and allows you to modify them, add new ones, view the next line of code containing a breakpoint, disable a breakpoint, and clear a breakpoint. It shows the location and skip count for each breakpoint. For breakpoints that are located in the current program, the paragraph they are contained in is also listed.</p> <p>The keyboard form of this command is “B”.</p>
View Monitors	<p>Shows all monitored variables and their values. It also displays a sequence number for each monitor. You need the sequence number to clear an individual monitor. See Data/Monitor/Clear.</p> <p>The keyboard form of this command is “M”.</p>
Memory Usage	<p>Displays the amount of dynamically allocated memory currently used by the runtime system. There are five types:</p> <p>Program memory is the memory directly used by your programs’ Data and Procedure Divisions. This includes all programs in memory—not just the current program.</p> <p>File memory is memory used by your open files, including the indexed file cache.</p> <p>Window memory is memory used by your pop-up windows. This includes the debugger’s own pop-up window.</p> <p>Overhead memory is memory used directly by the runtime system that is not controlled by your program.</p> <p>Dynamic memory is memory allocated by the program via the M\$ALLOC library routine.</p> <p>The keyboard form of this command is “U”.</p>

3.1.3.7 Run menu

The *Run* menu contains commands related to executing your program.



The Run Menu (Windows)

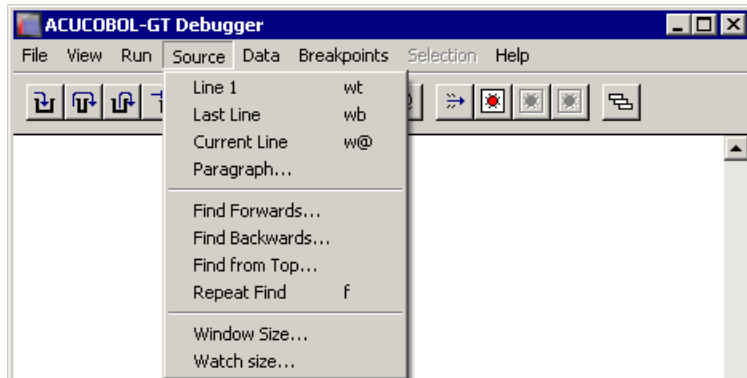
Menu Option	Description
Continue	Resumes execution of your program from its current location. The program returns to the debugger when it reaches the next breakpoint. The keyboard form of this command is “G”.
*Go to Cursor Line	Sets a temporary breakpoint at the current cursor line and continues execution of your program. Press the F3 key to use this command from the keyboard. The F3 key works on lines that do not contain verbs. The closest previous line with a verb is the location used to set the breakpoint.
Go until Paragraph Returns	Runs your program until the current paragraph returns to the point from which it was performed. The keyboard form of this command is “GP”.
Go until Program Exits	Runs your program until the current program exits to its calling program. If used from inside your main program, this command runs the program until it finishes. The keyboard form of this command is “GE”.

Menu Option	Description
Auto Step	<p>Causes your program to execute “step” commands repeatedly until it reaches the end of the program. When you select this mode, the debugger immediately begins stepping through your program. The debugger will follow new threads as they are created. (If you want to continue to follow the original thread, use the Step Over command) You can change the speed at which it is stepping by typing a digit from “1” (slowest; approximately three seconds per step) to “9” (fastest; several steps per second). Press the spacebar to leave Auto Step mode and return to the debugger prompt.</p> <p>The keyboard form of this command is “SA”.</p>
Step and P-Step	<p>Are not shown on the menu, but are available from the keyboard. Windows users can find equivalent commands, and others, on the toolbar provided with the debugger. The toolbar is explained at the end of section 3.1.3.</p> <p>Step executes one statement of your program and then returns control to the debugger. New threads are followed as they are created. (If you want to continue to follow the original thread, use the Step Over command.)</p> <p>The keyboard command is “S”. You may follow the keyboard command with a number of steps to take.</p> <p>P-Step executes a “perform step.” This is the same as a normal Step command, except that it includes the entire range of a PERFORM statement as a single statement. The effect is to step to the end of the performed paragraph. Use this command if you want to step through a program following only the original thread.</p> <p>The keyboard command is “P.” You may follow the keyboard command with the number of “perform steps” to take.</p>

Menu Option	Description
*Skip to Cursor Line	<p>Moves the current program location to the line containing the cursor. Further execution of your program will proceed from this line. The cursor line must contain a verb; otherwise the current program location does not change.</p> <p>Use this command with care, because the skipped lines are not executed. You may skip important sections of code and experience unexpected results.</p> <p>The keyboard form of this command is “@!”.</p> <p>Note: The “@!” command is not available for debugging a native-code module.</p>
Run all Threads	<p>Toggles or sets the “Run All Threads” setting. Once it is set, all threads run simultaneously under the debugger. Though all threads run simultaneously, only the debugger’s current thread is traced when you are stepping through a program. However, breakpoints in other threads are active and can transfer control to the debugger, as can a trapped error (such as a table boundary violation). When a thread other than the current thread returns control to the debugger, that thread becomes the current thread.</p> <p>The keyboard form of this command is “RA [#]”.</p>
Thread	<p>Shows the threads contained in the program, and places a check mark next to the current thread.</p>

3.1.3.8 Source menu

The *Source* menu contains commands related to viewing your source code. These commands are available with source-level debugging.



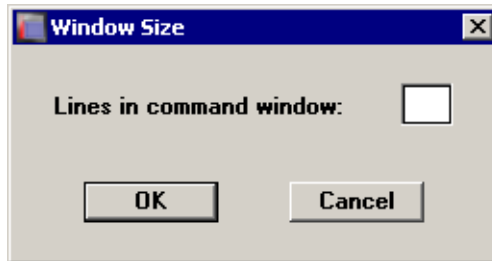
The Source Menu (Windows)

Menu Option	Description
*Line 1	Positions the cursor at the first line of your program. The keyboard form of this command is “WT”.
*Last Line	Positions the cursor at the last line in your program. The keyboard form of this command is “WB”.
*Current Line	Positions the cursor at the current line in your program. The keyboard form of this command is “W@”.
*Paragraph	Prompts you for a procedure name and positions the cursor there. The procedure must be located in the current program. The keyboard form of this command is “W <i>procedure</i> ”.

Menu Option	Description
*Find Forwards	<p>Prompts you for text to locate in the program's source code. The debugger searches forward, starting at the cursor line. Case is not considered, so you do not have to match the capitalization of the text you want to locate.</p> <p>The default text for the search is shown in a dialog box. This is the current <i>selection</i> (the currently highlighted variable or procedure name). If nothing is selected, the default is the last search string. If you do not want the default, simply type over it.</p> <p>Before you choose Find Forwards, you can highlight a variable or procedure name by clicking on it. If you do not have a mouse, use the arrow keys to move to the desired line and then press the Tab key to highlight the desired name.</p> <p>The keyboard form of this command is "FF <i>text</i>".</p>
*Find Backwards	<p>Prompts you for text to locate in the program's source code. The debugger searches backwards, starting at the cursor line. Case is not considered, so you do not have to match the capitalization of the text you want to locate.</p> <p>The default text for the search is shown in a dialog box. This is the current <i>selection</i> (the currently highlighted variable or procedure name). If nothing is selected, the default is the last search string. If you do not want the default, simply type over it.</p> <p>Before you choose Find Backwards, you can highlight a variable or procedure name by clicking on it. If you do not have a mouse, use the arrow keys to move to the desired line and then press the Tab key to highlight the desired name.</p> <p>The keyboard form of this command is "FB <i>text</i>".</p>
*Find from Top	<p>Prompts you for text to locate in the program's source code. The debugger searches for the text, starting at the top of the current program source. Case is not considered, so you do not have to match the capitalization of the text you want to locate. This is usually a convenient way to find the definition of a COBOL data item.</p> <p>The keyboard form of this command is "FT <i>text</i>".</p>

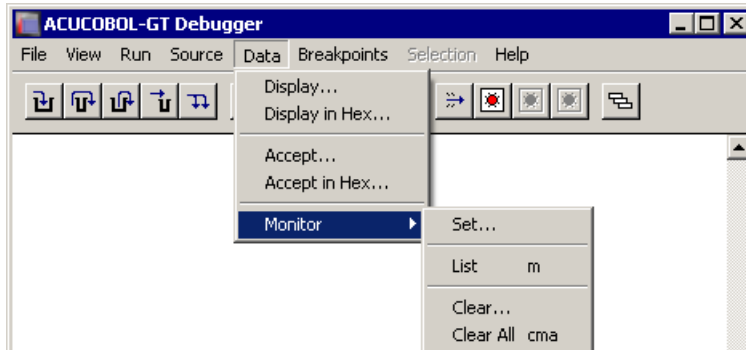
Menu Option	Description
*Repeat Find	Repeats the last Find command, starting at the cursor line. The keyboard form of this command is “F”.
Window Size	Sets the number of lines to show in the command window. This may be any integer from 2 to 14, inclusive. The keyboard form of this command is “WS <i>number</i> ”.
*Watch Size	Displays a dialog box that allows you to specify the number of lines to display in the Watch Window. The number cannot exceed the total number of items being monitored and watched. Specifying a larger number results in no change. The keyboard form of this command is “WW <i>number</i> ”.

The Watch Window size dialog (actually titled “Window Size”) looks like this:



3.1.3.9 Data menu

The *Data* menu contains commands relating to your program's variables.



The Data Menu (Windows)

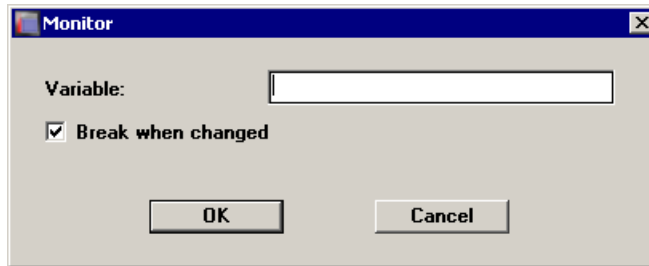
You may use name qualification with the **Display**, **Accept**, and **Monitor** commands. For example, you can use the syntax "FIELD-1 IN GROUP-1" to refer to a field called FIELD-1 that belongs to group item GROUP-1. Name qualification is not supported for on-screen commands (such as F7) or for situations in which you double-click on the data name.

Menu Option	Description
Display	<p data-bbox="575 245 1123 448">Shows the contents of a variable. With source-level debugging, you can either click on the variable name in the code before you select Display, or wait to be prompted.) Numeric variables are converted to show their value. Other variables are shown as text. The value is shown in the debugger command window. The keyboard form of this command is “D <i>variable</i>”.</p> <p data-bbox="575 464 1123 521">Table elements cannot be highlighted with a mouse click. Instead, use this keyboard command:</p> <pre data-bbox="615 537 861 561">d variable (index)</pre> <p data-bbox="575 578 1123 634">The variable’s name is followed by the desired index in parentheses. The index must be a numeric literal.</p> <p data-bbox="575 651 1123 732">To display a reference modified variable (that is, to view some portion or substring of the data item), use the syntax:</p> <pre data-bbox="615 748 821 773">d variable(x:y)</pre> <p data-bbox="575 789 1123 846">This shows <i>y</i> characters of <i>variable</i>, starting from character <i>x</i>.</p> <p data-bbox="575 862 1123 984">When you display a variable, and multiple fields with the same name are defined in the program, the debugger lists all instances of the field. For example, if the following two group items were defined in Working-Storage:</p> <pre data-bbox="615 1000 955 1227">01 start-date. 05 ws-day PIC XX. 05 ws-month PIC XX. 05 ws-year PIC X(4) . 01 end-date. 05 ws-day PIC XX. 05 ws-month PIC XX. 05 ws-year PIC X(4) .</pre> <p data-bbox="575 1243 1123 1325">and you entered the command “d ws-day”, you would see the value for the field in both the “start-date” and “end-date” group items.</p>

Menu Option	Description
Display in Hex	<p>Reference modification and indexing are valid with duplicate names. For example, all of the following are valid:</p> <pre data-bbox="663 354 884 435">d field-1(1:1) d field-1(1) d field-1(1)(1:1)</pre> <p>If your display command contains multiple field names, only the first name specified may be a duplicate. Using the group definitions shown in a previous example, the</p> <p>Shows the contents of a variable in hexadecimal. (With source-level debugging, either click on the variable name in the code before you select Display in Hex, or wait to be prompted for the name.) command “d ws-day(1:ws-month)” would fail, because there is more than one “ws-month” defined.</p> <p>Keep in mind that you must compile with “-Gd” or “-Gy” in order to reference variables <i>by name</i>. If the program was not compiled with one of these options, you must refer to each variable by its <i>absolute address</i> from a program listing, preceded by “.” (a period). For example:</p> <pre data-bbox="663 899 782 922">d .213.5</pre> <p>This option allows you to determine the data stored in each byte of the variable. The value is shown in the command window below the source code.</p> <p>The keyboard form of this command is “D <i>variable</i>, X”.</p>

Menu Option	Description
Accept	<p>Allows you to modify the contents of a variable. (With source-level debugging, either click on the variable name in the code before you select Accept, or wait to be prompted for the name.) For numeric variables, the value entered is converted to the internal storage format of the variable.</p> <p>The keyboard form of this command is “A <i>variable</i>”.</p> <p>When you accept a variable in the debugger, the current value of the variable is shown as the default. To leave the current value in place, press Enter.</p> <p>Table elements cannot be highlighted with a mouse click. To modify a table element, follow the variable’s name with the desired index in parentheses, as shown here:</p> <pre>a variable (index)</pre> <p>The index must be a numeric literal.</p> <p>Keep in mind that you must compile with “-Gd” or “-Gy” in order to reference variables by name. If the program was not compiled with one of these options, you must refer to each variable by its absolute address from a program listing.</p>
Accept in Hex	<p>Allows you to modify the contents of a variable in hexadecimal format. You can enter or display up to 2048 hex characters (1024 bytes of data).</p> <p>To accept a variable in hexadecimal format from the command line, use the command:</p> <pre>a variable x</pre>
Monitor	<p>This submenu contains commands that relate to monitored variables.</p>
Set	<p>Displays a dialog that prompts you for the name of a variable to be included in the Watch Window. (For source-level debugging, either click on the variable name in the code before you select Monitor, or wait to be prompted for the name.)</p> <p>The keyboard form of this command is “M <i>variable</i>”.</p>

The Monitor dialog looks like this:



The Monitor dialog box includes a check box labeled “Break when changed”. When this box is checked, the selected variable becomes monitored, and if it is unchecked, the variable is only *watched*. The default value of this check box is On (checked).

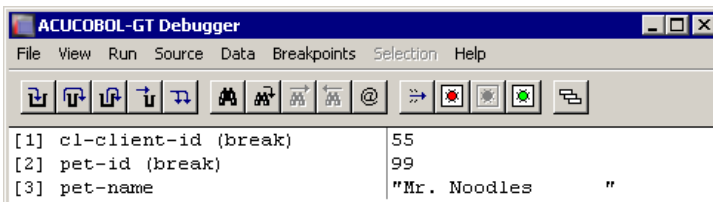
If the “Break when changed” box is checked in the Monitor dialog box, monitoring a variable suspends the program run. Any time a monitored variable changes, the program stops executing and control returns to the debugger, where the new value of the variable is displayed in the command area of the debugger window and in the Watch Window.

If the “Break when changed” box is unchecked in the Monitor dialog box, the item is watched. Though changes to a watched variable’s value are indicated in the Watch Window like those of a monitored variable, these changes do not cause the program to stop executing.

You can tell which variables in the Watch Window are monitored by the phrase “(break)” following the variable name (i.e., those variables for which the “Break when changed” check box was clicked on). The watched variables do not have this phrase displayed after their names.

When any variables are set for monitor/watch, a new window is created as a sub-window of the main debugger canvas, located at the top of the screen. This window, called the “Watch Window”, shows all the monitored/watched variables and their values, one name/value per line (values that exceed the size of the window are truncated). By default, the Watch Window contains as many lines as there are variables being monitored, up to a maximum of three. If you set more than 3 variables, you can scroll through the Watch Window to view them all, or you can make the Watch Window larger with the Window Size option on the Source menu. If your system does not use the

mouse, you can scroll the Watch Window using **Ctrl + P** (for previous item) and **Ctrl + N** (for next item) keys on your keyboard. The maximum number of variables you can set is limited only by system memory. The Watch Window looks like this:



To monitor a table element, follow the variable's name with the desired index in parentheses, as shown here (table elements cannot be highlighted with a mouse click):

```
m variable (index)
```

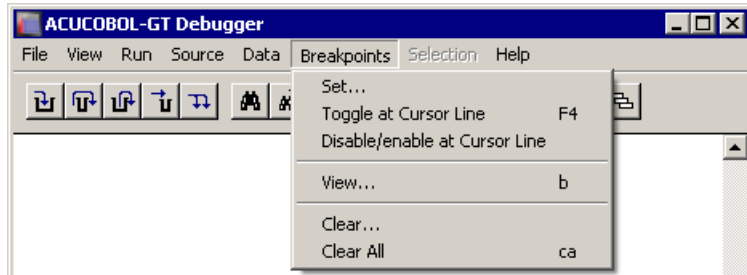
The index must be a numeric literal.

Keep in mind that you must compile with “-Gd” or “-Gy” in order to reference variables *by name*. If the program was not compiled with one of these options, you must refer to each variable by its *absolute address* from a program listing.

Menu Option (continued)	Description
List	Shows all monitored variables and their values. Also displays a sequence number for each monitor. You need the sequence number to clear an individual monitor. See <i>Clear</i> , below. The keyboard form of this command is “M”.
Clear	Clears a monitor from one variable. You will be prompted to identify the variable by number. Use the <i>List</i> option to display all monitors and their numbers. The keyboard form of this command is “CM <i>number</i> ”.
Clear All	Clears all monitors. The keyboard form of this command is “CMA”.

3.1.3.10 Breakpoints menu

The Breakpoints menu contains commands for managing a program's breakpoints.



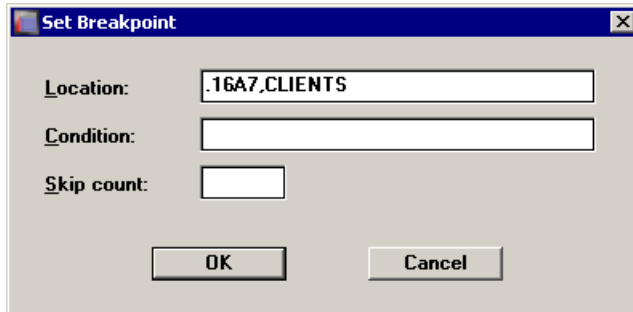
The Breakpoints Menu

A breakpoint is a location in your program's code that you designate. It causes control to return to the debugger. Control is returned before the code at the breakpoint location is executed.

Breakpoints are displayed in the source. An enabled breakpoint shows as "B" in column 1, a disabled breakpoint as "b" (lowercase) instead. The "@" sign (showing the program's current location) displays over the "B" if the current line is also a breakpoint.

Breakpoints are saved between sessions. The breakpoints are stored in a file that is named "*username.adb*", where *username* is your login name, as known by the runtime. This file is placed in the directory named by the "ACUCOBOL" environment variable, or the current directory, if that variable is not set. In addition to your breakpoints, the run-all-threads state is recorded, as well as the last size of the debugger's window. Keep in mind that although breakpoints are saved between sessions, they are not saved between compiles.

Menu Option	Description
Set	Allows you to set a breakpoint at a paragraph. Selecting Set displays the Set Breakpoint dialog.



Set Breakpoint Dialog Box

The Set Breakpoint dialog prompts you for a breakpoint *Location*, *Condition*, and *Skip count*.

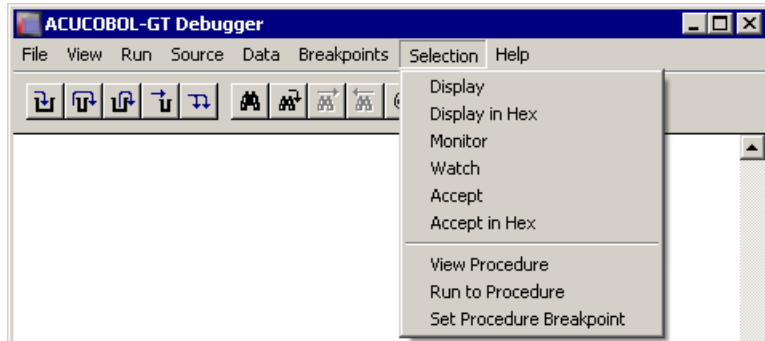
Set Breakpoint Field	Description
Location	<p>This field prompts you for a hexadecimal address and a program name, although the current cursor location is supplied as the default breakpoint address location. Hexadecimal addresses are specified with a “.” (period) as the first character, as described in section 3.1.7.2, “Program addresses.” A breakpoint is set at that address in the program. If you omit the program name, the current program is used. To obtain the hexadecimal address of a line of code, use the compiler’s program listing.</p> <p>Suppose you want to set a breakpoint in a called program in the run unit, but you do not know the exact address. First, make sure you’ve compiled the called program with source-level debugging. Then, from the current program, set the breakpoint at address “0”, called-program-name. The debugger breaks as soon as the called program is entered. You see the called program’s source code on the screen, and the called program’s name on the command line. The called program is now the current program, and you can use <i>Set</i> or <i>Toggle at Cursor</i> to set the desired breakpoint.</p> <p>If you have compiled with line numbers, you can use an alternate notation to set a breakpoint within a called program. Instead of specifying an exact address, provide the file name for the ACUCOBOL-GT source file or COPY file, followed by a colon (“:”) and the line number. The syntax is:</p> <pre>b [path/]filename.cbl:line</pre> <p>For example, in the command window, you could type:</p> <pre>b invoice.cbl:559</pre> <p>or</p> <pre>b /usr/copylib/mtrec.cpy:312</pre> <p>Note that when you use this method to set a breakpoint in a COPY file, the line number should be relative to the specific file, not to the main program. Also, if you set a breakpoint in an object module that has not yet been loaded, you will see a warning message, “Breakpoint saved for future COBOL module.” When the object is loaded, the breakpoint is set without further user intervention.</p>

Set Breakpoint Field	Description
Condition	<p>Breakpoints can have a condition, known as the “When Condition,” specified for them. The condition is entered into the Condition field. The breakpoint is activated only when the condition is true. For breakpoints with a skip count (see below), the skip count is decreased only when the condition is true. Conditions are simple comparisons between two numeric or alphanumeric data items or literals, including figurative constants (exception: the ALL literal is not supported). The allowed comparisons are “=”, “<”, “>”, “<=”, and “>=”. You may place the word “NOT” before any of these operators. The comparisons are done according to the rules for COBOL. Any data items referenced must exist in the program containing the breakpoint. If the condition is not meaningful or is illegal (including table boundary violations), then the breakpoint is immediately activated when it is reached and an error message follows.</p>
Skip count	<p>In the Skip count field, enter the number of times to skip the breakpoint. The breakpoint does not activate until the skip count reaches zero. The keyboard form of this command is</p> <pre>b address, counter</pre> <p>This command can also be set from the command line with:</p> <pre>b address [,program] [,SKIP count] [,WHEN condition]</pre> <p>A second command that is also supported but does not allow conditions to be set is:</p> <pre>b address [,program] [,count]</pre>

Menu Option (continued)	Description
*Toggle at Cursor Line	<p>Sets or removes a breakpoint at the source line containing the cursor.</p> <p>To use this command from the keyboard, press F4. The F4 key works on lines that do not contain verbs. The closest previous line with a verb is the location used.</p>
*Disable/enable at Cursor Line	<p>Allows you to keep a breakpoint location while turning off the breakpoint. You can disable/enable breakpoints from the menu or from the Breakpoint dialog box.</p>
View	<p>Is the same as the “list breakpoints” command (“B”). It displays a dialog box that lists all of your breakpoints and allows you to modify them, add new ones, view the next line of code containing a breakpoint, disable a breakpoint, and clear a breakpoint. It shows the location and skip count for each breakpoint. For breakpoints that are located in the current program, the paragraph they are contained in is also listed.</p>
Clear	<p>Removes a breakpoint. At the prompt, you can enter either the breakpoint’s paragraph name or hexadecimal address. Hexadecimal addresses are specified with a “.” (period) as the first character, as described in section 3.1.7.2, “Program addresses.” Exact addresses are given in the View command described above.</p> <p>To use this command from the keyboard, type “C address”.</p>
Clear All	<p>Removes all breakpoints.</p> <p>To use this command from the keyboard, type “CA”.</p>

3.1.3.11 Selection menu

The *Selection* menu lists actions you can take on the current selection.



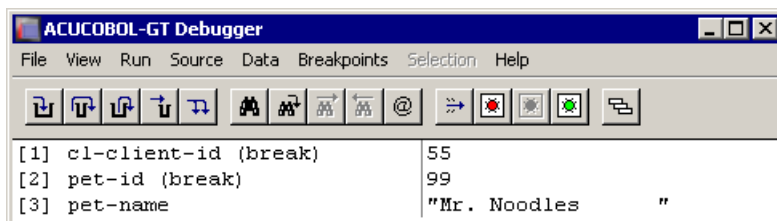
The Selection Menu (Windows)

A selection is a variable or procedure name that you have highlighted in the source window. If you do not have a mouse, use the arrow keys to move to the desired line and then press the Tab key to highlight the desired name.

Menu Option	Description
*Display	Shows the contents of the selected variable. Numeric variables are converted from their internal formats to show their values. Other variables are shown as text. You can also perform this by double-clicking the left mouse button on the desired variable.

Menu Option	Description
*Display in Hex	Shows the contents of the selected variable in hexadecimal notation. This allows you to view the internal storage of every byte in the variable.
*Monitor	<p>Sets a monitor on the selected variable. Changes to a monitored variable cause control to return to the debugger. This feature gives you the option to have the COBOL program stop executing, and the debugger to activate, when the value of a monitored variable changes. When this happens, the debugger window becomes the active window, and the variable and its value are displayed in the command area of the debugger.</p> <p>When any variables are monitored (or watched), a new window is created as a sub-window of the main debugger canvas, located at the top of the screen. This window, called the “Watch Window”, shows all the monitored and watched variables and their values, one name/value per line (values which exceed the size of the window are truncated). By default, the Watch Window contains as many lines as there are variables being monitored, up to a maximum of three. If you select more than three variables for monitoring, you can scroll through the Watch Window to view them, or you can make the Watch Window larger with the Window Size option on the Source menu. The size of the Watch Window cannot exceed the total number of monitored and watched items. Attempting to make the window larger than that results in no change.</p>

This is what a Watch Window looks like.



Menu Options (continued)	Description
*Watch	Sets a watch on the selected variable. Changes to a watched variable do <i>not</i> cause control to return to the debugger. See also *Monitor , above.
*Accept	“Accepts” a new value for the selected variable. For numeric variables, the value you enter is converted to the variable’s internal storage format.
*Accept in Hex	“Accepts” a new value for the selected variable in hexadecimal format. Up to 1024 bytes of data (2048 hex characters) can be entered or displayed.
*View Procedure	Scrolls the source window to the start of the selected procedure. You can also perform this by double-clicking the left mouse button on the desired procedure name.
*Run to Procedure	Sets a temporary breakpoint at the selected procedure and continues program execution. The program runs until it reaches the selected procedure (or another breakpoint).
*Set Procedure Breakpoint	Sets a permanent breakpoint at the selected procedure.
Up and Down	Are available only for non-Windows environments. Windows users can perform the same tasks by using the scroll bar to the right of the debugger screen. *Up scrolls up towards the top of the source code by one-half screen. *Down scrolls down towards the bottom of the source code by one-half screen.

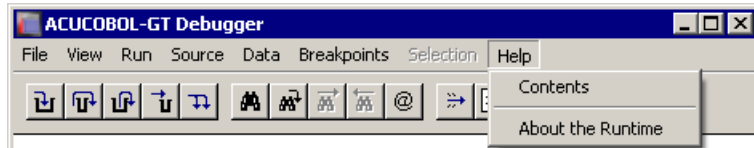
Help for Windows users is discussed in the next section. In other environments, you can get help by typing the letter “H” and pressing Enter at the debugger prompt.

The following debugger commands are available but are not shown on the debugger menus:

Debugger Command	Description
*F1, Page Up	Scrolls source up one page.
*F2, Page Down	Scrolls source down one page.
*F5 (or the Up arrow)	Moves the source cursor up one line.
*F6 (or the Down arrow)	Moves the source cursor down one line.
*F7	Causes the cursor line to be searched for program variables. If one is found, its name and current contents are displayed. Press F7 multiple times to cycle through all of the variables on the line. F7 (display variable on current line) and the Tab key (highlight variable on current line), as well as the mouse, also pay attention to qualified and indexed data items in the source. As long as a variable and all of its qualifiers and indexes are on one line, the entire expression is evaluated by these keys. If a variable and all of its qualifiers and indexes span multiple source lines, the entire expression is ignored, but component items are still found.
*F8, Edit Command	Causes the last command entered to be recalled for editing. Useful for correcting typographical errors.
*H, Help Key	Displays a screen of summary help information.

3.1.3.12 Help menu

The Help menu is available only for Windows users. It provides access to a Windows-style Help facility for the debugger.



The Help Menu (Windows)

Menu Option	Description
Contents	Shows you the Debugger Help table of contents.
Search	Allows you to search for specific words, as you would in a book index.
Help on Help	Opens the native Windows help file that explains how help files can be used.
About the Runtime	Gives you information regarding the runtime, such as the runtime version number, serial number, copyright information, and license number.

The Toolbar

Windows users can use the debugger's toolbar for a variety of operations. To display a description of any button on the toolbar, place the mouse pointer over the button and hold it there for a few seconds. Depending on the state of the debugger, some of the icons may be disabled.



The Debugger Toolbar (Windows)

Toolbar Button	Description
Step Into	Executes one statement of the program and then returns control to the debugger. It is the equivalent of the keyboard command “S.” The debugger will follow new threads as they are created. If you want to continue to follow the original thread, use the Step Over command.
Step Over	Allows you to “step over” a performed paragraph. It is the same as Step Into except that it includes the entire range of a PERFORM statement as a single statement. It is the equivalent of the P-step command. Use this command if you want to step through a program following only the original thread.
Step Out	Lets you run to a performed paragraph’s exit.
Run to Cursor (F3)	Sets a temporary breakpoint at the current cursor line and continues execution of your program.
Auto Step	Causes your program to execute “step” commands repeatedly until it reaches the end of the program. As with the Step Into command, the debugger will follow new threads as they are created. If you want to continue to follow the original thread, use the Step Over command.
Find	Brings up a dialog for entering a word or phrase you want to locate.
Find from Top	Locates the next occurrence of the last found word or phrase.
Find Next	Locates the next occurrence of the last found word or phrase.
Find Previous	Locates a previous occurrence of the last found word or phrase.
Find Current Line	Sets the source view to the current program location.
Go	Runs the program to the next breakpoint.
Toggle Breakpoint (F4)	Sets or removes a breakpoint at the source line containing the cursor.
Disable Breakpoint	Allows you to keep a breakpoint location while turning off the breakpoint.

Toolbar Button	Description
Remove All Breakpoints	Clears all breakpoints from the program.
Perform Stack	Displays the current Perform stack, listing all of the nested paragraphs leading up to the current statement.

3.1.4 File Tracing

File tracing is always available. Programs do not need to be compiled with the debug options to use file tracing. File tracing can be especially helpful in assessing the cause of a problem. File tracing provides valuable information about file OPENS, READs, and WRITEs. File status codes for unsuccessful I/O operations are also shown, and configuration variable settings can be examined. For relative files, file trace includes record numbers.

To enable file tracing, type:

```
runcbl -dlxe errfile myprog
```

where:

- d** turns on the debugger
- l** (optional) causes the contents of the runtime configuration file to be included in the error output
- x** causes the runtime system to display the operating system's corresponding error number for file error "30" on the error output. This information may help in determining the problem.
- e** causes the error output to be placed in the file named immediately after the option

errfile is the user-specified name of the error file. This file is opened as an empty file when the runtime is initiated. Do not forget to specify the error file name--if you run with “-e”, immediately followed by your program name instead of an error file name, your object code file will be deleted and opened as an empty file.

myprog is the name of your object code file

After you press Enter you are at the debugger screen. To turn on file tracing, type:

```
tf [#]
```

“File trace” is echoed on the screen.

Some file systems can print extra information if a higher level of tracing is enabled. This extra information is useful primarily to the Technical Support department, and they may ask you to execute a “tf #” for some integer.

File tracing can also be enabled with the **FILE_TRACE** runtime configuration variable. Some attributes of trace output can be tailored with the **TRACE_STYLE** configuration variable. For more information about both of these variables, see Appendix H of Book 4.

If you are writing to an error file, you can execute this debugger command:

```
t flush
```

to cause the error file to be flushed to disk after each write. This can be useful if your program terminates unexpectedly. It allows the error file to contain everything that the runtime sent to it. Without this command, the error file could be empty following an unexpected program termination, even though a great deal of information had been written to it. Note that this option slows down the processing but ensures that the error file is complete.

To start the program, enter:

```
g
```

Proceed until you encounter the error condition, and then exit. Your error file contains the error information, all COBOL configuration file variables that you have set, and a record of every file operation.

File trace timestamps

If you are directing file trace output to an error file, you can elect to include timestamp information. When this option is enabled, a timestamp is placed at the beginning of every line in the trace file. (When you are debugging a problem, it is sometime helpful to know the exact time of each file operation.) The format of the timestamp is: HH:MM:SS.mmmmmm, where “mmmmmm” is the finest resolution that the runtime can obtain from the system.

There are three ways to enable timestamps in the trace file.

1. In the debugger, before you start the program with the “g” command, enter:

```
t timestamp
```

2. Before you start the program, in the runtime configuration file set the `FILE_TRACE_TIMESTAMP` variable to “1” (on, true, yes). This variable is set to “0” (off, false, no), by default.

When set in the appropriate server configuration file, `FILE_TRACE_TIMESTAMP` can also be used with AcuServer and AcuConnect (see the associated product documentation for more information).

3. Before you start the program, in the runtime configuration file set the `TRACE_STYLE` variable to `TIMESTAMP`.

Timestamp information is included only when file trace information is directed to a file.

Timestamp output can add significant file I/O overhead and may have a noticeable impact on performance.

3.1.5 Screen Tracing

The screen trace feature enables you to save information about `DISPLAYs` of screen section items and `CREATEs`, `DISPLAYs`, `MODIFYs`, and `INQUIREs` of ActiveX objects. You can use screen trace even if the program was compiled without the debugging option.

To perform a screen trace, type:

```
runcbl -dle errfile myprog
```

Because you specified “-d” (for debugger) on your command line, you will be at the debugger screen after you press Enter.

To turn on screen tracing, type:

```
ts
```

“Screen trace ON” is echoed on the screen.

The information output is useful primarily to the Technical Support department.

If you are writing to an error file, you can execute this debugger command:

```
t flush
```

to cause the error file to be flushed to disk after each write. This can be useful if your program terminates unexpectedly. Note that this option slows down the processing but ensures that the error file is complete.

Type:

```
g
```

You will now be running your program normally. Proceed until you encounter the error condition, and then exit. Your error file will contain the error information, all COBOL configuration file variables that you have set, and a record of every file operation.

This can be especially helpful as you assess the cause of the problem.

3.1.6 Macro Debugger

The debugger supports a simple macro processor. Twenty-six variables, named “A” through “Z”, are available to be assigned to arbitrary strings. You do this with the command:

```
variable = string
```

where the “=” must appear in column two. After a variable is assigned, you may use it in any command by specifying the variable name with a “\$” in front of it. This provides a convenient way to assign a long symbol name to a shorter string.

For example, if the symbol “EMPLOYEE-NAME” is often referenced in a debugging session, the following commands will assign this to the variable “X” and display the contents of the name:

```
x=employee-name
d $x
```

Macros may not be nested.

3.1.7 Specifying Addresses

Program addresses and variables can be specified directly or with program symbols. In order for a symbol to be used, the program containing the symbol must be currently executing and must have been compiled with either the “-Gy” or “-Gd” options.

3.1.7.1 Variables

Variables can be specified by their symbolic name or by their address. If they are specified by *address*, both the starting address (in hexadecimal, preceded by a “.” (period)) and the variable’s size (in decimal) must be specified, in that order. These values can be found in the symbol table listing produced when you compile with the “-Ls” compiler option. Any variable specified directly by address is treated as if it were an alphanumeric variable. Variables specified by *name* are treated as their correct type, except for edited fields, which are treated as alphanumeric.

Either form of addressing may have an index specified for it. This index is a number in parentheses following the address. Only constant values may be used as table subscripts.

You may use name qualification with the **Display**, **Accept**, and **Monitor** commands. For example, you may type “FIELD-1 IN GROUP-1” to refer to FIELD-1 of GROUP-1. Name qualification is not supported for on-screen commands (such as F7) and for situations in which you double-click on the data name.

Data items may be qualified by a group name. Table indexes may be specified with variables.

Note: For data items of variable size, the debugger always treats the data item as if it were currently defined to be its maximum size.

Examples:

VAR-1	- Variable name
3A4, 5	- Address 3A4 for 5 bytes
ARRAY-1 (2,4)	- Indexed variable

Configuration variables

You can display and accept configuration variables within the debugger. To display a particular variable use the following command:

```
d %var-name
```

where *var-name* is the name of the variable you want to display. Up to 300 characters of the value are displayed. (This is equivalent to executing the ACCEPT FROM ENVIRONMENT command, and can show the same types of configuration variables.)

To accept a configuration variable, execute the command:

```
a %var-name
```

where *var-name* is the name of the variable you want to modify. In this case, the debugger responds with a prompt. Enter the new value of the variable, ACCEPT that variable within the COBOL program, and the runtime will use the new value.

3.1.7.2 Program addresses

Program addresses may be specified by paragraph name. They can also be specified by a hexadecimal address, specified by a “.” (period) as the first character. This allows the debugger to distinguish between the hex address ABC and the paragraph name “ABC”. You can omit the period when there is no ambiguity. Optionally, “.” (period) can be followed by the six-character program name. The numeric form is the only way to specify an address that is not at a paragraph, and the only way to specify an address in a program other than the one that is currently running. The listing produced by the compiler has the address of the start of each sentence along the left-hand side. Usually it is more convenient to use the F3 and F4 commands of the source debugger.

Note: Every program always starts at address zero. If you want to debug a subprogram, you can always set a breakpoint at address zero of the subprogram and run it until this point is reached. Then the subprogram will be active and its symbols will be available (if it was compiled with “-Gd” or “-Gy”). When specifying an address in a different program, use the name contained in its PROGRAM-ID paragraph.

Examples:

MAIN-LOGIC	- Paragraph name
3A7F	- Numeric address
0, PROG2	- Start of program PROG2

3.1.8 Debugger Restrictions

Please note the following restrictions on the debugger:

1. If you have a paragraph (or section) and a data item with the same name, that name will refer to the paragraph (section). If more than one paragraph (or section) has that name, the last one will be the one used. Other data items and paragraphs (sections) with that name can be referenced only by their addresses.

There is one exception to this rule. If a variable has the same name as a paragraph, you can still display (“d ambiguous-name”) and accept (“a ambiguous-name”) that variable in the debugger if you compile with at least “-Gs” and “-Gd”.

2. Although the compiler allows for up to 15 dimensions in a table, the debugger will let you access only the first three dimensions.
3. Although ACUCOBOL-GT object files are portable across all machines, an object file that contains debugging symbols or source may not be. These files can be run on other machines, but may cause errors if run with the debugger on the foreign machine.
4. You can use the debugger on a native-code module in the same fashion as you do for a portable-code module. The only restriction is that you may not begin execution at an arbitrary point in a native-code module (the “@!” command).
5. The debugger identifies a program name by a match on the first 30 bytes. This limitation derives from the compiler behavior, which reserves 30 bytes for the program name in the program object.

3.1.9 Using the Abend Diagnostic Report (ADR)

When you create an Abend Diagnostic Report to analyze the cause of an abnormal program shutdown, the report is divided into three sections:

1. The first contains general information about the program, such as command-line parameters, the reason for the shutdown, and the line number of the operation that caused the shutdown.

This section of the report appears as follows:

```
Dump created: Tue Dec 28 15:00:32 2006
```

```
Reason for dump:
```

```
Index out of bounds, upper bound = 10, index = 11  
COBOL error at 000014 in TwoTables.acu  
("TwoTables.cbl", line 42)
```

```
Runtime version: 8.0.0 (2006-12-23)
```

```
Command line arguments: -c Cfg.txt TwoTables.acu
```

2. The second section contains a call stack summary for each thread being run, including information about inactive programs. Inactive programs are those programs which have been loaded into memory but which are not currently executing.

```
Process ID: 1128

1 thread(s) active

** Thread 487 **
Call stack:
000014 TwoTables.acu

Inactive programs:
(none)
```

3. The third and largest section contains detailed information about each program, including the value of all data items. Programs are listed in CALL order, starting with the program executing at the time of shutdown and working backward to the start of the thread (usually the main program).

- All data items and their values are listed in the order they are declared in the program.
- Group items are named, but have the phrase “(group)” listed as their value to avoid duplicate information in the report.
- Individual elements in a group are listed with their values.
- Table items are expanded to show each element of the table.
- Data is shown in both the appropriate numeric/non-numeric format and as raw hexadecimal data.
- The compile options used to generate the object file.

```
*** DETAIL FOR THREAD 487 ***

*** PROGRAMS IN THE CALL STACK ***

*** PROGRAM "TwoTables.acu" ***
```

```
Current address: 000014
```

```
01 ONE-TOO-MANY                = 11                h30303031 31

01 MY-TABLE                    = (group)
05 FILLER                      = "    1"            h20202020 31
05 FILLER                      = "    2"            h20202020 32
.
.
.
00 SPECIAL REGISTERS           = ""                h

*** END OF PROGRAM "TwoTables.acu" ***

*** END OF THREAD 487 ***

*** END OF DUMP ***
```

3.1.9.1 Generating a report

To generate an Abend Diagnostic Report, you must set the **ACU_DUMP** configuration variable to “1” (on, true, yes). The default value for the configuration variable is “0” (off, false, no). This variable also takes name format specifiers that you can use to add additional identifier information to the report name. See **ACU_DUMP_FILE** in Appendix H-2 for details on using these name format specifiers.

In order to add detailed information to the report, programs must be compiled with line number (“-GI”) and symbol table (“-Gs”) information. The “-Ga” compiler option may also be used, but since this includes full source information in the compiled object, it results in a much larger object file on disk.

Configuration variables

In addition to ACU_DUMP, there are three other configuration variables that affect creation of an ADR.

ACU_DUMP_FILE

This configuration variable determines the name of the report file. It allows two special parameters:

- If the file name starts with a plus sign (“+”), the report is appended to the specified file. By default, a new report overwrites the specified file. Note that the “+” character does not actually appear in the file name.
- If the name contains the string “%p”, when the report is generated, that string is replaced with the process ID (PID) of the runtime from which the report originates.

The default value for `ACU_DUMP_FILE` is “`acudump.#`”, where “`#`” is an integer, starting at one and incrementing by one each time a new ADR is created in the current directory (`acudump.1`, `acudump.2`, and so on). Note that the first available filename is used, so if a directory contains files called “`acudump.1`” and “`acudump.5`”, the next ADR file created in that directory is automatically called “`acudump.2`”.

Because the runtime performs a linear search to determine the next available filename to use, if a directory contains a large number of ADR files, the search can take some time. For this reason, it is a good idea to remove unneeded ADR files regularly.

ACU_DUMP_WIDTH

This configuration variable controls the width of the report and has a default value of 80 characters. The minimum allowed value is 79 and the maximum is 2048. Note that because the report uses dynamically computed columns for its hexadecimal data, making the report very wide can reduce readability by introducing excessive white space.

ACU_DUMP_TABLE_LIMIT

This configuration variable limits how many elements of each table item to list. The default value is 1000. Note that if you increase this value substantially, and if you have tables that allow for large numbers of elements, you may get very large reports.

In the following example, `ACU_DUMP_TABLE_LIMIT` is set to 5:

```
01 MY-TABLE-R                               = (group)
05 TABLE-ENTRY (1)                         =      1      h20202020 31
05 TABLE-ENTRY (2)                         =      2      h20202020 32
05 TABLE-ENTRY (3)                         =      3      h20202020 33
05 TABLE-ENTRY (4)                         =      4      h20202020 34
05 TABLE-ENTRY (5)                         =      5      h20202020 35
Remaining table items suppressed due to ACU-DUMP-TABLE-LIMIT setting
```

3.1.9.2 ADR restrictions

You should note the following restrictions in the use of the ADR:

1. Tables are expanded up to only four dimensions. If you have a table with more dimensions, then only the first element of the higher dimensions is seen. This limitation comes from a limit in the object's internal symbol table.
2. Level 77 data items are listed as level 01 items. This is caused by the way the compiler internally stores the symbol table.
3. Level 88 data items are listed, but show the actual data instead of the true/false evaluation of the data.
4. The report does not show line numbers for programs in the call stack, only for the aborting program. Addresses for calling programs are shown, and these can be found in a listing of the program.
5. Programs must be compiled with line number information (“-GI”) in order to show line numbers, and symbol information (“-Gs”) to see data items in detail.

3.2 Object File Utility — `cblutil`

ACUCOBOL-GT's *COBOL library utility*, **cblutil**, works with ACUCOBOL-GT object files to provide several valuable capabilities. **cblutil** allows you to:

- place object files together to create *object libraries*
- output information about an object file or object library
- create native-code object files from machine-independent ACUCOBOL-GT portable object files

3.2.1 Object Libraries

An object library is a file that contains one or more compiled ACUCOBOL-GT programs. Object libraries can simplify the distribution of an application by reducing the number of files involved. They can also help improve performance by reducing the number of directory operations performed by **runcbl** when it is loading object modules. The advantages are particularly noticeable if the number of object files in a directory is large.

Each object library contains a *primary module*. The primary module is the first (or only) module in the library. When the library is loaded by a CALL statement (or is the first program of a run unit), the primary module is the program that is loaded and run. Other modules in an object library can be loaded by subsequent CALL statements.

In order for the runtime system to access other object modules in a library, the primary module must be loaded. It may either be active or inactive, but it must be physically present in memory. A program is loaded whenever it is called; it is unloaded whenever it is canceled (or when it exits, if it has the INITIAL attribute). See [section 6.3, “Memory Management,”](#) for a more complete description of runtime memory management.

Assuming that the primary module is loaded, then the other modules in the object library can be called if their name matches the name specified in a CALL verb. Modules in an object library are identified by PROGRAM-ID. If a matching name is found in an object library, that object module is then loaded and executed. See [section 2.10, “Calling Subprograms,”](#) for more information.

As suggested by these rules, you should place related object files together. Usually this is done by specifying the main program of a run unit as the primary module and then adding in some or all of the subprograms it calls.

Object libraries may also be pre-loaded. This is done with the “-y” option of **runcbl**. When a library is pre-loaded, all of its modules are always available. Note that pre-loading does not mean that the component object modules and contained ENTRY points are physically loaded into memory. It just means that the directory of the contained modules is loaded. More than one library may be pre-loaded, and pre-loaded libraries may be used with dynamically loaded libraries with no restrictions.

3.2.2 Creating Object Libraries

You can create object libraries with the **cblutil** program provided with the ACUCOBOL-GT runtime system. This command line has the following format:

```
cblutil -lib [options] modules
```

When you create a new object library, the first *module* specified becomes the primary module. All other modules are simply added to the library. If no *options* are specified, then the first module specified is converted from an object file or resource into an object library, and the remaining modules are added to it.

The first *module* may also be an object library. In this case, the remaining modules are added to the library. Any module that has the same name as one already contained in the library automatically replaces the one in the library.

The *modules* may be any type of file. If an input file is a COBOL object, then **cblutil** includes it in the resulting library as a COBOL object. Any other type of file is included as a *resource*. If an input file is another library, then each component of that library is individually added to the resulting library. The resulting library may consist entirely of COBOL objects, entirely of resources, or a mixture of the two.

A total of 1024 modules can be placed in a single library.

Options can be one or more of the following:

- c Used to embed a comment in the object library. This flag must be followed by the comment. Comments with embedded spaces must either be between quotation marks, or include the shell's escape character before each space.
- o This option must be followed (as the next separate argument) by a file name. This file becomes the new object library. If a file exists by that name, it will be deleted first.

- v Causes **cblutil** to be verbose about its progress.
- r Causes the separate *modules* to be deleted after they have been added to the object library. If “-o” has not been specified, then the first module (which becomes the new library) is *not* deleted.

Examples

The following sample command line creates a library called “mylib” that consists of two ACUCOBOL-GT objects named “prog1.acu” and “prog2.acu”:

```
cblutil -lib -v -o mylib prog1.acu prog2.acu
```

You can add a comment to the object library. The comment is visible when you use the “-info” command to retrieve information about the object library:

```
cblutil -lib -o mylib -c "My comment" prog1.acu prog2.acu
```

Alternatively, you can add a comment using an escape character instead of quotation marks as follows:

```
cblutil -lib -o mylib -c My\ new\ comment prog1.acu prog2.acu
```

Wild cards are permitted:

```
cblutil -lib -v -o mylib prog1.acu otherdir/*.*
```

To add modules to an existing library, do not use the “-o” argument. For example, to add “prog3.acu” and “prog4.acu” to “mylib”, do this:

```
cblutil -lib -v mylib prog3.acu prog4.acu
```

Note: There is no way to remove an object module from a library. For this reason, we recommend that you create object libraries after all of the programs involved have been fully debugged.

3.2.2.1 Creating remote object libraries

If AcuServer or AcuConnect is running on a remote machine, **cblutil** can read remote objects and write a remote library. The syntax rules that apply to specifying remote object libraries with **cblutil** are the same as those for compiling to remote object libraries with the compiler. See [section 2.2.18.1, “Remote file name handling,”](#) for details.

This capability allows you to create a remote library from local object files or to create a local or remote library from remote object files.

With **cblutil**, you can also use the regular AcuServer syntax for referring to the remote files. This syntax is not allowed with the compiler because the “@” symbol is reserved for another purpose. See also, *AcuServer User’s Guide*, section 7.2, “Accessing Remote Files,” for additional information.

Note: You *cannot* use wildcard characters to create a library from a collection of remote object files.

In the process of creating a remote library, **cblutil** overwrites the named library at the beginning of the operation. Then if something fails during the process, the library is removed. For that reason, you may consider creating a backup copy of the named library before executing the *build library* command. (Incidentally, when creating a local library, **cblutil** creates a temporary library first. Only after the new library has been successfully compiled is the (existing) named library removed and replaced by the new library.)

Examples

The following command creates a library in /myapp/obj on the UNIX server *myserver* called “myapp.lib” from all the .acu files in the current directory.

```
cblutil -lib -o acurfap://myserver::/myapp/obj/myapp.lib *.acu
```

acurfap stands for “Acucorp Remote File Access Protocol.”

The following command creates a library in `/myapp/obj` on the Windows server *myserver* where *AcuServer* is listening on port 6543. The library is named “*myapp.lib*”. The files used to create the library are all in `/myapp/obj` on *myserver*. Because you cannot use wildcard characters, you need to list each file.

```
cblutil -lib -o
  acurfap://myserver:6543:c:/myapp/obj/myapp.lib \
  acurfap://myserver:6543:/myapp/obj/test1.acu \
  acurfap://myserver:6543:/myapp/obj/test2.acu \
  acurfap://myserver:6543:/myapp/obj/test3.acu \
  acurfap://myserver:6543:/myapp/obj/test4.acu
```

Note: The use of the backslash character (“\”) as *line continuation* delimiter works only on UNIX systems. If you are entering a command for Windows, you must type the entire command as a continuous string.

3.2.3 Getting Object Information

The **cblutil** program can output useful information stored in the header of an object file or object library. The format of the command is:

```
cblutil -info [-x] files
```

The options used to compile a COBOL program are automatically embedded in the program’s object file. The “-x” option to the “-info” command causes **cblutil** to output all the options used to compile the object file.

Each file named on the command line is examined to determine if it is an object module or object library. If it is an object module, its size and other information is output. If the file is an object library, information is output for each module the library holds.

In each report, **cblutil** includes information that indicates whether the module is in debug-mode. Because programs compiled for source-level debugging can be quite large, it can be helpful to run reports on a regular basis to see if you have accidentally left any programs in debug-mode.

For example, the following could be run on a UNIX system every night:

```
cblutil -info /objects/* | grep "debug" > /tmp/debug
```

This command creates a file called “/tmp/debug” that lists every program in the “/objects” directory that is in debug-mode.

The **cblutil** program also reports whether or not table-boundary checking is enabled in an object file, and, if the object contains an embedded comment, lists the comment.

3.2.4 Generating Native Code

The “-native” option of **cblutil** allows you to translate ACUCOBOL-GT portable object modules into native-code object modules. The “-native” option has the following format:

```
cblutil -native [options] object-files ...
```

options can be any of the following:

--intel or --ia-32	produces 32-bit native code for Intel-class processors (386, 486, Pentium, Pentium II, Pentium III or compatible processors).
--pa_risc or --pa	produces 32-bit native code for PA-RISC version 1.0 running the HP-UX or MPE/iX operating systems
--pa_risc_2.0 or --pa2	produces 64-bit native code for PA-RISC version 2.0 running the HP-UX operating system
--power	produces code that is compatible with POWER and POWER2 processors, as well as PowerPC and later POWER series processors. This option allows you to use a wide range of machines, but it may affect performance.
--powerpc or --ppc	produces 32-bit native code for IBM pSeries processors running AIX operating system Note that you can compile native code only for machines with a POWER3 or later chip, not with POWER2 or earlier.

--powerpc_64 or --ppc64	produces 64-bit native code for IBM pSeries processors running AIX
--sparc	produces 32-bit native code for SPARC (v7 - v9) processors.
--sparc_v9	produces 64-bit native code for SPARC version 9 processors.
-o	names the output file. This option must be followed (as a separate argument) by the name of the file to produce. You may use “@” in this name to stand for the base name of the input object file. If you specify “-o” and multiple object files, then you must use “@” in the name. If you omit “-o”, then the output file replaces the input file.
-v	causes cblutil to print the name of each object file as it is being processed.
-Zc	produces code that is more compact and somewhat slower.
-Zn	turns off the more involved optimizations.

If you specify multiple object files, then each one is translated in turn. If ‘object file’ refers to an object library, then each module contained in the library is translated. If an object file contains debugging information, that information is retained.

If you do not specify a target processor, then **cblutil** translates for the processor of the host machine, if native code for that processor is supported. Once an object file has been translated to native code, it cannot be translated again for a different instruction set.

If the object module does not contain ACUCOBOL-GT’s portable instruction set, the “cblutil -info” command includes in its outputs the name of the native instruction set used.

3.3 Vision File Utility — **vutil**

On Windows, UNIX, and Linux systems, ACUCOBOL-GT uses the Vision indexed file system to manage its indexed data files. For these systems, ACUCOBOL-GT comes with an indexed file utility program called **vutil** that contains several useful functions. (The full name of this 32-bit utility is “vutil32”, but throughout this discussion, we refer to it simply as “vutil”.) This section describes this program.

Note: Other file system interfaces have their own file utility packages. On VMS systems, for example, ACUCOBOL-GT uses the RMS file system that is native to VMS, and the **vutil** utility is not supplied. VMS-specific programs such as ANALYZE/RMS and CONVERT can be used to accomplish the same functions that **vutil** provides. See the manual for your specific file utility package for details on its use.

vutil provides several functions in one package. It can be used to:

- display file information (-info, -size, -tree)
- test file integrity (-check)
- rebuild and repair files (-rebuild)
- reset the user count (-zero)
- reset the internal revision number (-fixvers)
- extract data records (-extract)
- create empty files (-gen)
- unload data to binary or line sequential files (-unload)
- load data from binary or line sequential files (-load)
- convert other index files to Vision (-convert)
- change the maximum record size (-augment)
- recover deleted records (-deleted)

- place text in the “comment” field of the header (-note)

Each of these functions is indicated by an initial keyword on the command line (preceded by a hyphen). This keyword may be abbreviated to its first letter. The functions are designed to allow you to specify all possible task parameters up front, so that the utility can run unattended or with a minimum of user interaction. Each function is discussed below.

3.3.1 Examining File Information

The “info” function of **vutil** returns some basic information about Vision indexed files. The command syntax is:

```
vutil -info [ -kpxq ] [ files ]
```

If no files are specified on the command line, then **vutil** reads file names from the standard input. Several options can be specified with “-info”:

- p This option causes **vutil** to pause between files and prompt the user for a “return” key. Otherwise, all the reports are run together.
- k This option prints full details about each key, including the exact layout of a multi-segment, or split, key. Each segment is expressed as a pair of numbers--segment size (sz) and the offset from the beginning of the record (of).
- q This option causes **vutil** to exit (with status 99) if user interaction is required.
- x This option causes **vutil** to report additional (extended) information.

The basic information provided by the “info” function consists of:

- text in the “comment” field (frequently empty)
- Vision file format (Version 2, 3, 4, or 5)
- total number of records
- total number of deleted records

- file size of each segment (Version 4 and 5 only)
- total size of all segments combined (Version 4 and 5 only)
- segment size (maximum possible; Version 4 and 5 only)
- record size (min/max)
- number of keys
- user count

If you request extended information with the “-x” option, the following additional information is output:

- for each key: key size (total size and number of segments, if split); key offsets; whether duplicates are allowed
- block size
- blocks per granule
- tree height (max/min/avg)
- number of nodes
- number of deleted nodes
- total node space
- node space used

The “tree height” is the number of levels in the B-tree and is directly related to how efficient the file is. If the maximum number exceeds four or five, then the file may benefit from rebuilding with a larger block factor (see [section 3.3.3, “Rebuilding Files,”](#) below).

An important piece of information is the *user count*. The user count is initially set to zero, and is incremented each time the file is opened for I/O. The number is decremented when the file is subsequently closed. Under normal circumstances, the user count indicates the number of users who are currently updating the file. Should **runcbl** terminate abnormally, the user count may not be decremented. Therefore, if the user count is a non-zero

value when there are no active users, it indicates that there may have been a sudden runtime failure and that corrective action may be required. At the very least, the file should be checked for integrity (see **section 3.3.2, “Testing File Integrity”**), but depending on the program that died, more significant action may need to be taken. A non-zero user count indicates that someone knowledgeable about the system should intervene and ensure that everything is okay. By monitoring the user count, the user count can be used as an early warning system to head off some types of file problems before they surface in a more serious form. Note that because **runcbl** usually closes all files when it detects an error, it is very unusual that a COBOL coding error will cause a non-zero user count condition.

Note: Unlike RM/COBOL, a non-zero user count is not automatically an indication of a corrupt file. It merely means that a program has died while it had files open.

3.3.2 Testing File Integrity

The “check” option of **vutil** tests a file for internal consistency. The command is:

```
vutil -check [ -afkqx ] [ files ]
```

With no options, **vutil** reads a list of files from the standard input and tests each one for a non-zero user count and other quickly tested errors. Files with errors or a non-zero user count are listed. You may place the list of files to check on the command line instead of using the standard input.

- a** (for “automatic”) This option causes **vutil** to do a thorough test of each file that has a non-zero user count. It will read every record in an attempt to see if the file is broken. Any problems that are detected are printed. You can use this option to test a large number of files for errors without exhaustively reading every record from every file. Only those files that appear to have potential problems (because of the non-zero user count) are tested.
- f** (full) This option forces a file to be checked (including files with a user count of zero). When both “-a” and “-f” are specified, “-f” takes precedence.
- k** (key number) This option is used to specify the key to be used to read the file. All the keys in the file are read sequentially by the specified key during the check of the file. This option must be used in combination with the “-a” or “-f” option. This option has no effect when used with the “-x” option. “-k” must be followed (as the next separate argument) by the number of the key you want to use. Zero (“0”) indicates the primary key, “1” indicates the first alternate, and so forth.
- q** This option causes **vutil** to exit (with status 99) if user interaction is required.
- x** (extended tests) This option causes **vutil** to run extended tests in place of those that are normally run by the “-a” or “-f” options. The extended tests include: reading every record with every key, reading the records in their physical order in the file, and checking the deleted records list. The filename is displayed along with a message that indicates which test **vutil** is currently working on. This option causes a *write* lock to be placed on the file to ensure exclusive access during the tests. You must specify the “-x” option with either “-a” or “-f” on the same command line; used by itself, it does nothing. The “-x” option disables the “-k” option when the two are specified on the same command line.

Note: Although the “check” option tests the file thoroughly, it is possible for a file to be corrupt and still pass the test. If you’re processing an indexed file outside of **vutil** and you receive a file error “98,” that file is corrupt even if it passed the “vutil -check” test.

For convenience in building scripts, the “check” option will not complain if given a non-Vision file. This allows “check” to be run on an entire directory without generating spurious errors from relative and sequential files.

When you perform “vutil -check”, one of the following status values is returned to the host operating system when **vutil** quits:

0	file passed all checks
1	checks not fully performed because the file was in use
2	non-zero user count found
3	file is corrupt
99	user interaction was required, and the “-q” switch was set
255	vutil fatal error or incorrect command line

If more than one file is checked, the highest status value that applies is returned.

3.3.3 Rebuilding Files

The “rebuild” option is used to rebuild or recreate an indexed file. You should rebuild a file that has become corrupt, or one that contains a large number of deleted records that you want to remove from the file. The command is:

```
vutil -rebuild [ --slow ] [ -l ] [ -t tmpfile ] [ -b # ]
[ -2345 ] [ -ac ] [ +ce ] [ -k keynum ] [ -d dir ]
[ -f factor ] [ -s spoolfile [ -r ] [-m size] ]
[ -p pre_factor ] [ -g ext_factor ] [ -q ] [ files ]
```

Each file listed on the command line will be rebuilt. If no files are listed, then the standard input is read for the list. If, under UNIX, the named file is a symbolic link, the link is removed and the restored file is put in its place.

This option by default applies a read lock to the file that is rebuilt. The “-l” option applies a write lock instead.

When a file is rebuilt, a temporary file is created and each record from the original file is written to it. The “-t” option allows you to specify the name of the temporary file used during the rebuild. (You may not specify a directory, just a file name.) When “-t” is not specified, the temporary file’s name begins with “VTMP”, followed by a six-character system-generated sequence. On Windows systems, the file’s name begins with “V”. The rebuilding process reports the number of records found and the number of deleted records that were skipped. After the rebuild is complete, you are given the option of replacing the original file with the new one. If you do not replace it, you can examine the temporary file for correctness and replace it manually later. This is recommended if you suspect any difficulties.

When doing a rebuild, **vutil** places records that are rejected due to illegal duplicate keys into a file. Should this happen, **vutil** will report the name of the file that contains the rejected records. The format of this file is the same as a COBOL binary sequential file with variable-size records.

-a This option may be used to specify automatic replacement of the original file by the newly created one. This is useful when you are calling **vutil** from a program or a script.

When used once, this option causes automatic replacement only if no records are skipped. If any records are skipped, you are prompted before file replacement takes place. When used more than once, this option causes automatic replacement of the file even if records were lost in the process.

The multiple specification of option “-a” may be given in the following syntax formats:

-aa

-a -a

-a (*other options*) -a

- b #** This option sets a new blocking factor for the file. The blocking factor specifies the size of the blocks to be used by the file. Blocks are sized in 512 byte increments. Vision 5 files support blocking factors from 1 to 16 (16 = 8192 bytes). Vision 2, 3, and 4 files support blocking factors from 1 to 2.
- When you rebuild a file, if the file is very large, or has a tree height of more than five, or key lengths in excess of 40 bytes, you may want to experiment with larger blocking factors. You will need to perform some benchmarking to determine if a larger block size improves performance. For more about how block size can affect performance, see [section 6.1.3.7](#).
- If you specify a blocking factor greater than 2 for a Vision 2, 3, or 4 file, the factor is automatically and silently reduced to the maximum of 2.
- c** This option removes record compression from the file.
- +c** This option adds record compression to the file.
- d *dir*** This option specifies an alternate directory for placing the rebuilt file. *Dir* should be the name of a directory on the host machine other than the directory containing the files to be rebuilt. When this option is used, the original files are not modified or destroyed. The rebuilt files are placed in *dir* with the same base name as the original files. This option can be useful if you do not have enough disk space on the device holding the files to rebuild them, but you do have space on another disk. This option implies the “-a” option because you are not prompted before the rebuild completes.
- +e** This option adds record encryption. It is not possible to remove record encryption (this would make encryption pointless).

Record compression and encryption may be added to a file, and compression may be removed from a file, regardless of the presence or absence of the WITH COMPRESSION and WITH ENCRYPTION phrases in the file's SELECT.

-f *factor*

This option allows you to specify a compression factor. The *factor* must be an integer that specifies how much of the space saved by compression is actually to be removed from the record. Zero means no compression. A value of 1 means use the default factor (70).

For factors from 2 through 100, the factor is considered to be a percentage. It specifies how much of the space saved by compression is actually to be removed from each record. For example, suppose an 80-byte record is compressed to 30 bytes. Then the compression factor is used to determine how much of the 50 bytes of saved space is actually to be removed from the record. A compression factor of 70 would mean that 70% of the 50 bytes (35 bytes total) will be removed. This leaves 15 bytes for future expansion, and results in a compressed record size of 45 bytes (30 compressed size plus 15 extra for growth). The larger the compression factor, the more of the saved space is removed. A compression factor of 100 removes all saved space and allows no room for expansion.

-g *ext_factor*

This option sets a new *extension_factor* for the file. This is the number of blocks that are added to a file's size when the file needs to be expanded. The default is one block. Specifying more than one enables you to take advantage of contiguous disk space, and thus may help to prevent fragmentation of the file as it grows.

-k *keynum*

This option specifies that you want to rebuild the file in key order. The “-k” must be followed (as the next separate argument) by the number of the key that you want to use, with zero indicating the primary key, one indicating the first alternate key, and so forth. For example, to rebuild “file1” in primary key order, you would specify:

```
vutil -rebuild -k 0 file1
```

There are two situations in which the “-k” option is particularly valuable. If you are repeatedly processing a file along a particular key, then you can improve performance by rebuilding the file in key order. This is particularly true if you do a great deal of sequential processing (common in reports). When you rebuild in key order, records that are logically adjacent (according to their key values) are placed next to each other on the disk. This maximizes the runtime’s ability to improve performance with its read caching capabilities. It also minimizes the distance that the disk must seek when you are reading records sequentially by that key. Write performance also improves in applications that write large numbers of records in keyed sequence.

A second situation in which the “-k” option is valuable is when the default rebuild method fails to recover a file fully. This can occur if the chain of data records has been corrupted. When “-k” is specified, **vutil** will use the index you provide to try to locate the records, and will often find more records this way.

- p** *pre_factor* This option allows you to specify the number of blocks that **vutil** is to pre-allocate to the file. *pre_factor* must be a numeric value between one and 2,097,152. The maximum pre-allocation factor varies with Vision version. Vision 5 files accept the upper limit of 2,097,152 blocks. Vision 2, 3, and 4 files are restricted to a maximum of 65,535 blocks. If a larger pre-allocation factor is specified than the Vision version allows, the factor is automatically and silently reduced to the allowable limit.
- q** This option causes **vutil** to exit (with status 99) if user interaction is required.
- s** *spoolfile* This option indicates that you want to use the spooling form of rebuild. This is especially helpful if you do not have adequate disk space to hold the new file. This option spools the records to removable media and then rebuilds the file over the existing file. This keeps only one copy of the file on disk and thus allows you to rebuild even when free disk space is limited. Note that the spooled file is *not* compressed.

The “-s” option must be followed by the name of the file to which you want to spool records. This can be any file but is usually the name of a tape or diskette device. For example, you might specify

```
vutil -rebuild -s /dev/rmt0 badfile
```

to rebuild the file “badfile” by spooling records to the tape device “/dev/rmt0”.

When “-s” is specified, **vutil** writes all the records it can recover from the corrupt file to the spool file, and then rebuilds the file using these records. You will be prompted to change media if the spool file gets full.

There are two additional options that can be used with the “-s” option:

-r allows you to recover an interrupted rebuild. When “-r” is specified, **vutil** skips the step of writing records to the spool device. Instead, it prompts you to mount the first volume of the spool file before it begins the rebuilding process.

-m *size* allows you to specify the size of the spool media. It is followed by the number of 1024-byte records that can fit on the media. This is useful when the spool device driver does not handle the end-of-media condition correctly. For example, if you were spooling to a 1.2 MB floppy disk, you could specify:

```
-m 1200
```

--slow This option causes **vutil** to open the file for “mass update” instead of for “bulk addition.” This usually causes **vutil** to run slower. The only reason for using this option is as a possible work-around to some difficulty with using bulk addition.

-# This option causes **vutil** to rebuild the file in the Vision file format specified by the integer. Valid values include 2, 3, 4, and 5. If the “-#” option is not included, the file is rebuilt in the same format as the original file.

When you perform “**vutil -rebuild**”, one of the following status values is returned to the host operating system when **vutil** quits:

- | | |
|------------|---|
| 0 | file successfully rebuilt |
| 1 | rebuild not performed because the file is locked |
| 2 | rebuild not fully performed because some records were not recovered |
| 99 | user interaction was required, and the “-q” switch was set |
| 255 | other errors |

3.3.4 Resetting User Counts

This option resets the user count of each named file to zero. It is much faster than rebuilding when you are certain there are no other problems with the file. The command is:

```
vutil -zero [ -q ] [ files ]
```

The files may be listed on the command line, or may be read from the standard input. For convenience in building scripts, non-Vision files are ignored.

-q This option causes **vutil** to exit (with status 99) if user interaction is required.

3.3.5 Resetting Internal Revision Number

This option resets the internal revision number of all segments in the specified Vision file to the revision number of the first data segment. It does this without rebuilding the entire file. To use this option, you must have exclusive access to the file. The command is:

```
vutil -fixvers [ -q ] [ file ]
```

The files can be listed on the command line, or can be read from the standard input. For convenience in building scripts, non-Vision files are ignored.

This option can be used to repair “98, 89” and “98, 90” conditions that can result from improper shutdowns of a runtime or improper closure of a file. Before using this option, you should be certain that the file is otherwise internally correct (meaning that the data is not corrupted. See [section 3.3.2, “Testing File Integrity”](#)). Improper use may lead to loss of data. After using the “-fixvers” option, you should run “vutil -check -f file” to verify internal consistency of the file.

-q This option causes **vutil** to exit (with status 99) if user interaction is required.

3.3.6 Extracting Records From a File

The “extract” option prints selected records on the standard output. The command syntax is:

```
vutil -extract [ -x ] [ -k# ] [ -n# ]
           [ -v value ] [ -q ] file
```

When using the “extract” option, you may use command-line options to specify the primary key, starting value, and the number of records. If you do that, **vutil** does not interrupt the “extract” later (after printing a synopsis of the file) to prompt for those parameters. **vutil** does a START NOT LESS THAN on the desired key and proceeds to print records on the standard output. Each record is printed on its own line.

Note: If the file contains binary or COMP-3 data, that data may contain carriage returns (binary “0D”s). Each binary “0D” is interpreted as a carriage return, and that is reflected in the display of the extracted record.

These options can be used with “-extract”:

- k#** This option specifies the key number to extract from.
- n#** This option specifies the number of records to extract.
- v value** This option specifies the key value from which to start the extract.
- q** When you use this option, the key number defaults to “0” (zero), the number of records defaults to “all”, and the *keyval* defaults to low-values, unless you specify these values with the “-k”, “-n”, and “-v” options.
- x** This option allows record extraction to continue if an error occurs. Records that generate errors are not included in the output file.

vutil will not let you extract records from an encrypted file.

3.3.7 Recovering Deleted Records

The “deleted” function recovers records that have been marked as deleted, but which have not yet been overwritten by a new record. This function can be used only with Vision 5 files.

The “-deleted” option looks for records marked as deleted and writes their contents to a sequential file. For example:

```
vutil -deleted -vb infile.vis outfile.seq
```

reads through the list of deleted record areas in “infile.vis” and writes the data to “outfile.seq” in the form of a sequential file with variable length records and a portable record header that indicates the size of the record.

The file containing the deleted records may be loaded back into the Vision file with “vutil -load”, or opened by the runtime like any other sequential file.

Note: The “deleted” function works only with Vision 5 files.

The command syntax is:

```
vutil -deleted [-v] [-b] [-t] [-q] source destination
```

- | | |
|---------------------------|--|
| -v | creates a file that contains variable-length records. |
| -b | creates a binary sequential file that is compatible with the ACUCOBOL-GT runtime.

If the “-v” option is also specified, variable length records are written. Otherwise, fixed-length records are written. |
| -t | creates a line sequential file. This option always writes variable length records. |
| -q | tells vutil to perform all actions without prompting the user for input. This is useful when running “vutil -deleted” as a batch job. |
| <i>source</i> | is the name of an existing Vision 5 file. |
| <i>destination</i> | is the name of the file to be filled with the recovered data. |

By default, recovered records are written in a machine dependent binary sequential file format *that is not compatible with the runtime*. To create a file that is compatibility with the ACUCOBOL-GT runtime, include either the “-b” or -t” option.

3.3.8 Creating Empty Files

The “gen” function creates empty Vision files. This is equivalent to doing an OPEN OUTPUT on the files from COBOL and is supplied as an alternative to writing a program explicitly to create empty data files. The command syntax is:

```
vutil -gen [ -2345 ]
```

or

```
vutil -gen [ -2345 ] [ -q ] list directory
```

The first command format invokes a prompting program that asks you for the name of the new file and each file attribute. The second command format allows you to specify all of the file attributes in advance, and store them in a file.

Whether you store the attributes in a file or respond to prompts at the keyboard, the file attributes you provide are the same.

- 5 creates a file in Vision Version 5 format. This is the default.
- 4 creates a file in Vision Version 4 format.
- 3 creates a file in Vision Version 3 format.
- 2 creates a file in Vision Version 2 format.
- q This option causes **vutil** to exit (with status 99) if user interaction is required.

When you perform “vutil -gen”, one of the following status values is returned to the host operating system when **vutil** quits:

- 0 file successfully rebuilt
- 255 unsuccessful

3.3.8.1 Responding to **vutil** generated prompts

If you use the interactive version of the “gen” option, you are immediately given the opportunity to store the session in a file, so that your responses can be used again. (In fact, you can use the session file as the *list* file with the non-interactive version of “gen”.) If you indicate that you do want to save the session, you are prompted for a session file name.

Next you are prompted for the name of the new file, and for its attributes. The exact prompts are shown here, and are described in **section 3.3.8.2, “Specifying file attributes in advance.”** Default values are enclosed in brackets.

```
Save this session [Y]?
Enter session filename:

Enter filename:
Enter the blocking factor [1]:
Enter the number of blocks to pre-allocate [1]:
Enter the # of blocks for extension [0]:
Enter the compression factor (0-100) [0]:
Enable record encryption [N]?
Enter the maximum record size:
Enter the minimum record size (1-maximum) [maximum]:

Enter the # of keys [1]:
-- Primary key --
Enter number of segments (1-6): (For generating Version 2 or 3 files)
Enter number of segments (1-16): (For generating Version 4 or 5 files)
Enter segment size:
Enter segment offset: (Segment size and offset repeat as a pair for each segment)
Duplicates allowed [N]?

-- Alternate key n -- (repeats for each alternate key)
Enter number of segments:
Duplicates allowed [N]?
Enter segment size:
Enter segment offset: (Segment size and offset repeat
                      as a pair for each segment)

Enter translation table filename:
Enter file comment (30 char max):

Generate another file?
```

Collating Sequence

One of the attributes you may specify is the name of a file containing a translation table. This enables you to create a custom collating sequence for the new file, instead of using the standard ASCII collating sequence. The exact format for the translation table is given here.

All white-space characters (space, tab, new line, etc.) are ignored, so the table can have as many lines and spaces as you desire.

The sequence of the characters in the table determines the collating sequence for keys. For example, a file which looks like this:

```
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

would sort keys reverse alphabetically, for the values in the range A to Z.

You may enter special characters by typing a backslash (\) and then the decimal value of the character desired. Thus, “\032” would be used to specify the SPACE character.

Ranges can be specified with a dash (-). The sequence of the starting and ending characters in the range is significant. The reverse-alphabetical table shown above could be specified more concisely as:

```
Z - A
```

Finally, you can give two or more characters the same sort value by using an ampersand (&) between them. For example, the file will not distinguish case if you use a translation table with the following format:

```
a & A b & B c & C d & D e & E f & F g & G h & H i & I j & J  
k & K l & L m & M n & N o & O p & P q & Q r & R s & S  
t & T u & U v & V w & W x & X y & Y z & Z
```

Any characters in the native collating sequence that are not explicitly named in the table assume a position greater than any of the explicitly named characters. The relative order of these unnamed characters remains the same as in the native collating sequence. In the last example, all digits, punctuation, and control characters would be in their usual order, but after all alphabetic characters.

3.3.8.2 Specifying file attributes in advance

```
vutil -gen [ -2345 ] [ -q ] list directory
```

The non-interactive version of “gen” allows you to specify a file (*list*) that contains the attributes for one or more new files. The format of *list* is described below.

The *directory* parameter names the directory in which the new files are to be created. Each file is tested to see if it exists before it is created. If it does exist, and it is a Vision file, then it is left untouched. Thus, you can use the “gen” function to generate missing files from a directory without having to first save the ones that are there.

The file *list* consists of one or more file entries, one per line. Each entry pertains to exactly one file and consists of a series of fields.

The *list* file can have one of three formats. There is a format for creating relative and sequential files. A format for creating Vision Version 2 files (support is provided for compatibility with older applications; the format is not described here). And a format for creating Vision Version 3, 4, and 5 files (documented below).

For indexed files, the fields are divided into five groups, separated with semicolons. Fields within each group are separated with commas.

For relative and sequential files, the fields are all separated with commas.

Indexed format

The fields for the indexed format are listed here and then described below.

```
filename,  
blocking factor,  
number of blocks to pre-allocate,  
number of blocks for extension,  
compression factor,  
Enable record encryption?;  
  
maximum record size,  
minimum record size,  
number of keys;
```



```
For primary key:
    number of segments,
    Duplicates allowed?, (always zero)
    segment size,
    segment offset, (repeat the segment size and offset
                    pair for each segment)

For each alternate key:
    number of segments,
    Duplicates allowed?,
    segment size,
    segment offset, (repeat the segment size and offset
                    pair for each segment);

translation table filename;
file comment
```

In the indexed format, the first field is the (physical) file name. The second field is the blocking factor. For Vision 5 files, the value can range from one to 16. For Vision 3 and 4 files, the value must be one or two (if a larger value is specified, it is automatically reduced to two). All I/O to the disk is done in blocks of one or two sectors. Depending on the file and the underlying disk architecture, performance can be affected by this. Although performance is difficult to predict, files that have very large keys may benefit from a larger blocking factor. See [section 6.1.3.7](#) for a more complete discussion.

The third field is the number of blocks to allocate to the file initially. This is usually set to one. If you want to pre-allocate some disk to the file, then this can be set to a higher number. Pre-allocation in no way limits the file, but may help performance by reducing disk fragmentation.

The fourth field is the number of blocks for extension. This determines how many blocks are allocated each time space needs to be added to the file. This helps keep fragmentation to a minimum.

The fifth field is the compression factor. A compression factor of zero (0) means no compression. A compression factor of one (1) is equivalent to the default compression (70). For factors from 2 through 100, the factor is considered to be a percentage. It specifies how much of the space saved by compression is actually removed from the record. For example, suppose an 80-byte record is compressed to 30 bytes. Then the compression factor is

used to determine how much of the 50 bytes of saved space is actually removed from the record. A compression factor of 70 means that 70% of the 50 bytes (35 bytes total) is removed. This leaves 15 bytes for future expansion, and results in a compressed record size of 45 bytes (30 compressed size plus 15 extra for growth). The larger the compression factor, the more of the saved space is removed. A compression factor of 100 removes all saved space and is advisable only if the file is rarely updated.

The sixth field is a flag that determines whether record encryption is enabled. A value of one (1) enables encryption. A value of zero (0) disables encryption. A semicolon should follow the encryption flag.

The next two fields specify maximum and minimum record size. If the two numbers are identical, the records are fixed-length. If the two numbers are not identical, records are variable-length. The maximum record size allowed in Vision 5 files is 67,108,864 bytes. The maximum record size allowed in Vision 2, 3, and 4 files is 32,767 bytes.

The ninth field is the number of keys in the file, to a maximum of 120. A semicolon should follow the number of keys.

Next, you describe the primary key by at least four entries. The first entry is the number of segments in the key. The second entry is always zero (0). For each segment, you must then specify the segment size in bytes, and the segment offset from the start of the record, in bytes. If there are no alternate keys, a semicolon should follow the final segment offset. Otherwise, a comma should be used.

If there are any alternate keys, describe each one by a series of at least four entries. The first entry is the number of segments in the key. The second entry should be one (1) if duplicate values are allowed, or zero (0) if they are not. For each segment, you must then specify the segment size in bytes, and the segment offset from the start of the record, in bytes. A semicolon should follow the final segment entry of the last alternate key.

After the keys have been specified, enter the name of a file containing the translation table (collating sequence), if you want anything other than standard ASCII sorting. If the name is empty, ASCII sorting is assumed. The format of the translation file is given in the preceding section. A semicolon should follow the name of the translation file.

Finally, you may provide up to 30 bytes of comment. This comment is printed by **vutil** when the “info” option is used.

Here’s a sample file entry. Suppose a file containing G/L account descriptions has a record size of 80 and two keys. The primary key is at the start of the record and is 15 bytes long. The alternate key has two segments; the first is at record offset 40 and is 30 bytes long. The second segment of the alternate key is at record offset 20 and is 5 bytes long (duplicates allowed). A compression factor of 30 and ASCII sorting are desired. The corresponding entry is:

```
glactfil,1,1,0,30,0;80,80,2;1,0,15,0,2,1,30,40,5,20; ;G/L account master
```

Sequential and Relative Files

For convenience, the non-interactive “gen” option can also create empty sequential and relative files if they are missing. The entry contains only three fields. The first field is the file name. The second field is the record size, and the final field is an “S” for a sequential file or an “R” for a relative file. The record size field is only comment, so it can be set to any numeric value.

Whether to use “gen” or a COBOL program to create the data files for an application depends on which is more convenient. Creating the file list can be painstaking, but the symbol table listing of the compiler can help to compute the size information. Once the files are created, however, it is easier to replace missing files this way than with a program that must explicitly test for a file’s existence before creating it.

3.3.9 Unloading to Binary and Line Sequential Format

The “unload” option will create a binary sequential file or a line sequential file from a Vision file. The command is:

```
vutil -unload [ -v ] [ -b | -t ] [-l] [-q] source destination
```

The *source* file is the Vision file to unload; the *destination* is the name of the file to create. If a file with the name *destination* already exists, it is deleted first. The records in the destination file are ordered by the primary key of the source file. This can be used to export data to other applications. **vutil** will not let you unload records from an encrypted file.

The source file is buffered according to the value in the `A_SEQ_DEFAULT_BLOCK_SIZE` variable. The variable must be set in the environment for **vutil** to use it. If the variable is not set, the default buffer block size is 4096 bytes. If the variable is set to “0”, **vutil -unload** into a sequential file will perform record-based I/O. If the variable is set to a positive value, that value will be rounded up to the power of two equal to, or greater than the value. This will be the buffer size in bytes. The maximum buffer size is 1GB.

By default, the destination file is assumed to be a binary sequential file with an alternate format that is not compatible with the ACUCOBOL-GT runtime.

These are the destination file format options:

- v** This option produces a file that has variable-length records. Variable-length records occupy only as much disk space as necessary. Two or four bytes indicating record size are placed in front of each variable-length record when it is written to disk. (Different machines generate different prefixes. Thus, files produced with “**vutil -unload -v**” can be loaded with “**vutil -load -v**” on the source machine but are not necessarily portable to other machines.) *The two- or four-byte field that is added to the record is not specified in your COBOL program, but some programs that access the records need to be aware of the extra bytes.*
If “-v” is not present, fixed-length records are written.
- b** This tells “**vutil -unload**” to produce a binary sequential file that is compatible with the ACUCOBOL-GT runtime.
If “-v” is not present, fixed-length records are produced.
The “-v” option causes **vutil** to produce variable-length records. The record length is stored in a two-byte record header.
- l** This option places a read lock on the input Vision file. This improves performance, because the records can be read without needing to place and release locks on the individual records.

- t** This tells “*vutil -unload*” to produce a file that has line sequential format. This means that the destination file is a simple text file, with records separated by line feeds.

This option implies “-v” (variable-length records), so the “-v” option is not necessary, although it is allowed.
- q** This option causes **vutil** to exit (with status 99) if user interaction is required.

3.3.10 Loading a File

The “load” option will create an indexed file from a binary sequential file, a relative file, or a line sequential file. The command is:

```
vutil -load [-b|d|t] [-lnv(r|s|x)] [-q] source destination
```

The *source* file is the name of the binary, relative, or line sequential file to read. The *destination* file is the name of the Vision file to add to. This file must already exist; it is used to determine the record size and key information.

By default, records from the *source* file are added to the *destination* file. If the “-n” flag (new file) is used, then any data in the *destination* file is eliminated before the records are loaded from the *source* file.

When doing a load, **vutil** places records that are rejected due to illegal duplicate keys into a file. Should this happen, **vutil** will report the name of the file that contains the rejected records. The format of this file is the same as a COBOL binary sequential file with variable-size records.

The input file is buffered according to the value in the `A_SEQ_DEFAULT_BLOCK_SIZE` variable. The variable must be set in the environment for **vutil** to use it. If the variable is not set, the default buffer block size is 4096 bytes. If the variable is set to “0”, **vutil -load** into a sequential file will perform record-based I/O. If the variable is set to a positive value, that value will be rounded up to the power of two equal to, or greater than the value. This will be the buffer size in bytes. The maximum buffer size is 1GB.

By default, the source file is assumed to be a binary sequential file with an alternate format.

These are the source file format options:

- b** This loads a binary sequential file that is compatible with the ACUCOBOL-GT runtime into a Vision file.
If “-v” is not present, fixed-length records are read.
The “-v” option causes **vutil** to read variable-length records. The record length is stored in a two-byte record header.
- d** Records marked as deleted in the relative file are discarded.
The “-v” option is not allowed for relative files.
- l** You can use the “-l” flag to prevent **vutil** from locking the file if you need to allow simultaneous access to the destination file while **vutil** is operating. Normally, **vutil** locks the destination file to improve the performance of the load operation. When “-l” is not used, **vutil** adds records to the file using “bulk addition” mode, which generally runs faster.
- n** If the “-n” flag (new file) is used, then any data in the *destination* file is eliminated before the records are loaded from the *source* file.
- q** This option causes **vutil** to exit (with status 99) if user interaction is required.
- r** This option causes any “duplicate key” write errors to be retried as rewrites to the file. This option should be used with caution, because duplicate key write errors often indicate that an error exists in the target file description. Warnings about this problem are not seen when you use the “-r” option.
This option is incompatible with the “-load -s” option.
- s** This option indicates that duplicate records should be skipped rather than written to a file. When this option is used, any duplicate records found while loading the indexed file will be discarded.
This option is incompatible with the “-load -r” option.

- t This loads a file that has line sequential format into a Vision file. This means that the source file is a simple text file, with records separated by line feeds. The source file may not contain any line feeds within the data fields, because a line feed denotes the end of a record.

This option implies “-v” (variable-length records), so the “-v” option is not necessary, although it is allowed. Line sequential files are assumed to contain variable length records. As such, they can only be loaded into Vision files that have been generated to accommodate the needed range of record sizes. If, however, the file contains records that are uniformly fixed length, the Vision file can be generated to accommodate only that fixed length. Should **vutil** attempt to load variable length records into a fixed record-size Vision file, an invalid record size error will occur. The error is reported as a generic “parameter error.”

- v This option causes **vutil** to treat the source file as a file with variable-length records. The record length is stored in the record’s header. The length of the header is either two or four bytes, depending on your machine type.

If “-v” is not present, fixed-length records are read.

- x The “-x” option is required when you are working with binary sequential and relative files that contain variable-length records larger than 65,535 bytes. (These files store the record length in two additional bytes in the record header. For “vutil -load” to read these files, it is necessary to indicate that these extra header bytes exist.)

If you are creating this file for the first time, you can either use the “gen” option of **vutil** or write a COBOL program to create the empty Vision file. The “load” function can be used to import data from another application.

If an error occurs, an exit status of 255 is returned.

3.3.11 File Size Summary Report

The “size” option of **vutil** gives summary disk usage information for a set of Vision files. This option is useful for quickly determining which files are occupying the most disk space and for spotting files that contain a large amount of unused space. The command is:

```
vutil -size [ -n ] [ -q ] [ files ]
```

If no files are requested, then the standard input is read for the list of files. The printed information includes the total size of the file, the number of records, and the number of deleted records the file contains. Also given is the percentage of records in the file that are not deleted records. This information is useful when you are trying to find candidates to rebuild when disk space gets tight. Non-Vision files are ignored by this command.

- n** This option shows all files, including non-Vision files. Although **vutil** “-size” option normally ignores non-Vision files, sometimes it may be useful to see which files are being ignored. This option provides that capability.

Non-Vision files display as:

```
junkfile: not a vision file
```

- q** This option causes **vutil** to exit (with status 99) if user interaction is required.

3.3.12 Converting RM/COBOL-85 Indexed Files

vutil can convert an indexed file created by RM/COBOL-85 into a Vision file. For a complete description, see section 2.4.4 in *Transitioning to ACUCOBOL-GT*.

3.3.13 Converting C-ISAM Files

vutil can convert a C-ISAM® file into a Vision file. This is useful when you are moving C-ISAM data to an ACUCOBOL-GT application. The command is:


```
vutil -convert [ -a ] [ +c ] [ -2345 ] [ -d dir ]  
              [ -f # ] [ -q ] [ files ]
```

You need not specify that the file is C-ISAM; **vutil** makes that determination.

The “convert” option starts with the same letter as the “check” option. You must use at least two letters of the word “convert” in order to specify this option. If you just use “-c”, **vutil** will assume that you are specifying the “check” option.

The “convert” function will take each named *file* and convert it from a C-ISAM file to a corresponding Vision file. If no *files* are specified, then the standard input is read for a list of files to convert.

Each C-ISAM file actually occupies two files: an index file with the extension “.idx” and a data file with the extension “.dat”. Specify only the *base name* in the list of files (do not include any extension).

Specifying “+c” causes the resulting records to be compressed.

Normally **vutil** warns the user about the impending conversion and asks if the user wants to continue. The “-a” (for “automatic”) option suppresses this warning. This can be useful when you are calling **vutil** from another program.

The “-5” option specifies that you want the resulting file to be in Vision Version 5. The “-4” option specifies a Vision Version 4 file. A “-3” means a Version 3 file, and “-2” specifies a Version 2 file.

The “-d” option specifies that you want the converted files to be placed in a new directory. *Dir* should be the name of a directory on the machine other than the directory containing the files to be converted. The “-d” option implies the “-a” option.

The “-f #” option sets the compression factor to be used when the file is converted. This option does not force the use of compression, it merely sets the compression factor if compression is used. The compression factor, a numeric literal, specifies how much of the space saved by compression is actually to be removed from the record.

The “-q” option causes **vutil** to exit (with status 99) if user interaction is required.

There are a few types of files that cannot be converted due to restrictions in Vision. Any of the following properties will cause **vutil** to print a message and leave the file alone:

1. A record size or block size greater than 32 KB.
2. More than 120 keys.
3. An individual key with more than 250 bytes in it.
4. A single key with more than sixteen segments (Vision Version 4) or more than six segments (Vision Version 2 or 3).
5. A primary key that allows duplicates.

vutil makes a copy of the file while it is converting it. You must have adequate disk space for **vutil** to complete its conversion. Also, C-ISAM files and Vision files differ in the amount of disk space that they use. This difference is fairly unpredictable and can vary quite widely. Sometimes the Vision files are smaller, and sometimes the C-ISAM files are smaller. You should have some spare disk space when you start converting files to accommodate the potential difference.

3.3.14 Converting Micro Focus Files

vutil can convert a Micro Focus file into a Vision file. This is useful when you are moving Micro Focus data to an ACUCOBOL-GT application. The command is:

```
vutil -convert [ -ac ] [ +c ] [ -f # ] [ -2345 ]  
              [ -d dir ] [ -q ] [mf-files]
```

You need not specify that the file is a Micro Focus file; **vutil** makes that determination on its own.

The “convert” option starts with the same letter as the “check” option described earlier. You must use at least two letters of the word “convert” in order to specify this option. If you just use “-c”, **vutil** will assume that you are specifying the “check” option.

The “convert” function takes each named *mf-file* and converts it from a Micro Focus indexed file to a corresponding Vision file. If no *mf-files* are specified, then the standard input is read for a list of files to convert.

Each Micro Focus file actually occupies two files: an index file with the extension “.idx” and a data file. The resulting Vision file has the same name as the data file, with the extension “.vis”. Specify only the *base name* in the list of files (do not include any extension).

Normally **vutil** warns you about the impending conversion and asks if you want to continue. The “-a” (for “automatic”) option suppresses this warning. This can be useful when you are calling **vutil** from another program.

Specifying the “-c” option causes the resulting file to have uncompressed records regardless of the original file; using “+c” causes the resulting records to be compressed.

The “-f” option sets the compression factor used when the file is converted. This option does not force the use of compression, it merely sets the compression factor if compression is used. The compression factor, a numeric literal, specifies how much of the space saved by compression is actually to be removed from the record.

The “-5” option specifies that you want the resulting file to be in Vision Version 5. The “-4” option specifies a Vision Version 4 file. A “-3” means you want a Version 3 file, and “-2” means you want a Version 2 file.

The “-d” option specifies that you want the converted files to be placed in a new directory. *Dir* should be the name of a directory on the machine other than the directory containing the files to be converted. The “-d” option implies the “-a” option.

The “-q” option causes **vutil** to exit (with status 99) if user interaction is required.

vutil makes a copy of the file while it is converting it. You must have adequate disk space for **vutil** to complete the conversion. Also, Micro Focus files and Vision files differ in the amount of disk space that they use. This difference is fairly unpredictable and can vary quite widely. Sometimes the

Vision files are smaller, and sometimes the Micro Focus files are smaller. You should have some spare disk space when you start converting files to accommodate the potential difference.

3.3.15 Changing Record Size

The “augment” option makes it possible to increase the maximum record size of a Vision file. This is useful for adding fields to a record without having to rebuild the entire data file. The new maximum record size and the file name is specified as shown in the examples below. This command format is:

```
vutil -augment [ -q ] new_max_rec_size filename
```

For example:

```
vutil -augment -q 50 myfile.dat
```

or

```
cat vision_filelist | vutil -augment -q 100
```

If the Vision file originally had a fixed-length record size, and if the new maximum record size is larger than the old maximum record size, the file effectively has a variable-length record size after running this command.

You may specify a new maximum record size that is smaller than the current maximum record size, but not smaller than the current minimum record size. This enables you to correct for the case that the maximum was too large. Be careful, however, because if any records were added while the maximum was at the higher level, the file is marked as broken when those records are next read. Vision detects that a record exists that is larger than the current (reduced) maximum record size and raises an error. When you use “vutil -augment” to reduce the maximum record size, **vutil** issues a warning.

Anytime you change the file record size with the “augment” option, you should consider the need to modify existing FDs to reflect the new maximum record size. Changing the characteristics of a file without making changes to existing FDs will cause a mismatch to be detected at runtime when the file is opened, resulting in a file-status error 39 (“Existing file conflicts with the COBOL description of the file”).

Because this operation changes the logical structure of the file, exclusive file access is required. **vutil** reports “File locked” if any other process has the file open.

The “-q” option causes **vutil** to exit (with status 99) if user interaction is required.

3.3.16 Setting the Comment Field

The “note” function allows you to set the comment field in the Vision file header.

The usage is:

```
vutil -note [-q] "comment" [file ...]
```

-q causes **vutil** to exit (with status 99) if user interaction is required

“comment” is limited to 30 characters and is truncated if longer

“vutil -note” sets the *comment* field in the specified Vision file to “comment”. If no file is listed on the command line, the filenames are read from standard input (one per line).

A Vision file’s comment field can be viewed with the “vutil -info” command. If the field is not empty, the comment is displayed, enclosed in parenthesis, on the line immediately following the filename.

3.3.17 Miscellaneous Commands

The “tree” function produces a listing of the internal B-tree in a file called “v_tree”. The command is:

```
vutil -tree [-q] file
```

This is primarily used by Micro Focus staff to help debug suspected problems with Vision. Five columns of information are displayed, with these headings:

Left/Rec Uniq Size Pre Key

The Left/Rec column displays the pointer from the entry to the next tree level or to the actual record itself. The Uniq value is used to distinguish duplicated keys. The Size field is the number of bytes in the key (as stored after key compression). The Pre field is the number of bytes this key shares with the preceding key. The Key field is the actual key value.

The “version” option of **vutil** tells you which version of the utility you are running. The command is:

```
vutil [ -q ] -version
```

-q This option causes **vutil** to exit (with status 99) if user interaction is required.

3.3.18 Default Settings of vutil

vutil uses the following default settings:

V_BUFFERS	128 blocks (1 block = 512 bytes)
V_BULK_MEMORY	1 megabyte

You can modify these settings if desired by placing the new settings in the operating system’s environment.

Note: **vutil** does not use the runtime’s configuration file. Settings made there have no effect on **vutil**.

3.4 File Transfer Utility — vio

vio is a file transfer utility similar to the UNIX program **cpio**. **vio** allows you to collect a group of files together into archives, and allows you to extract some or all of these files from these archives. Typically, an archive is some external media such as a tape or a diskette, but the archive may also be another disk file. **vio** is typically used to back up a set of files or to move files from one machine to another.

vio is particularly well suited for moving files to a different machine, because:

1. **vio** is available on a wide-range of operating systems, including Windows, UNIX, Linux, and VMS.
2. **vio** automatically adjusts for certain machine-dependent aspects such as byte-swapping.
3. **vio** handles multiple volumes gracefully.
4. On any system where Vision is supported, **vio** can automatically convert ACUCOBOL-GT indexed data files to the appropriate format for the target machine.

vio runs in two modes, the input mode (-i) and the output mode (-o). The syntax for each mode, with all possible options, is shown here:

```
vio -o [ -b ] [ -f file ] [ -u ] [ -g ] [ -h headerfile ]
      [ -k ] [ -l listfile ] [ -pr ] [ -s blocks ] [ -v ]

vio -i [ -cd ] [ -f file ] [ -g ] [ -h bytes ]
      [ -kmnstv2345 ] [ files ]
```

The input mode reads **vio** archives to extract files. The output mode creates new **vio** archives.

In the output mode, **vio** reads its standard input for a list of files to place in the archive. The archive is written to its standard output.

In the input mode, **vio** reads the archive from its standard input and extracts all the files. The extracted files have the same names, permissions, and owners that they had when the archive was created.

If *files* are specified, then only the named files are extracted. Note that each file must exactly match the name of a file in the archive; no wild card characters are allowed.

When **vio** encounters an ACUCOBOL-GT indexed data file, it treats that file specially. When it's running in output mode, it extracts each data record from the file and writes that record to the archive along with some formatting information. When that file is later read in the input mode, a new indexed

data file is created with the proper format, and each data record is loaded into the file using the host's indexed file system. Using this technique, **vio** is able to transfer an indexed file so that it is ready for use on the target machine.

When it's archiving files other than indexed files or ACUCOBOL-GT object files, **vio** assumes that the files are text files. It performs any conversions necessary to match the text file conventions on the host machine. For example, if a file is transferred from a UNIX system to a Windows system, new-line characters are translated into carriage-return, line-feed sequences. The "-p" option described below can cause these files to be treated as binary files instead, in which case no translation occurs. (If you are transferring multiple files at one time, some ASCII and some binary, do not use the "-p" option. Instead, add a space followed by a "b" or a "B" after the name of each binary file in the filename list. The "<space> b" prevents translation from occurring on an individual file.)

Note: ACUCOBOL-GT object files are automatically detected and written out to an archive as binary files, even if you fail to specify "-p".

3.4.1 **vio** Options

The following options can be used with the **vio** utility:

- b** Causes the archive to be blocked with 10 input records per output record. Each input record is normally 512 bytes. Blocking is specified only during output; **vio** automatically determines whether or not an archive is blocked when it is doing input.
- c** Forces all files read from the archive to be placed in the current directory. Any directory information in each file is removed and the file is placed in the current directory using just its base name. This is useful if someone sends you a file with a full directory specification that does not match your machine.
- d** Allows **vio** to create directories as needed to read a file.

-f Allows you to specify the archive file directly. The next separate command-line argument is the name of the archive file. This is particularly useful when you are writing a multi-volume archive, because **vio** will not need to prompt you for the name of the archive when it has to change volumes.

-g Rings the bell when a new volume is needed.

-h In the input mode, you may specify a number of bytes to skip from the beginning of each archive volume. This value is specified as the next separate argument. This is used to skip headers on media that some machines produce.

In the output mode, the next separate argument should specify the name of a file. This file is exactly copied to the beginning of each archive volume. This is used to simulate a media header required by a target machine.

For example, an AT&T 7300 diskette contains a header in the first 8 sectors. If you write a diskette on another type of machine, the 7300 will not recognize it. To get around this problem, take a diskette written on a 7300 and extract the header using this UNIX command: “dd if=disk-device of=73header count=4096”. You can then specify “-h 73header” as part of a **vio** command to have this header placed on each diskette of the archive. The 7300 will then be able to read these diskettes. If you are coming from a 7300, you can use “-h 4096” to cause **vio** to skip the first 8 sectors of each diskette.

-k Changes **vio**'s notion of a record size from 512 bytes to 1024 bytes. All I/O is done using the record size, or a multiple of the record size. This option is occasionally useful on some machines that require 1K transfers to devices. If you use this option on output, you must then also use it when reading the created archive. This option should be used only if required. You can improve performance better by using the “-b” option instead.

- l** Allows you to specify a list of files to output as the next command-line argument. **vio** will then read this list instead of using its standard input. Note that this list must reside in a file, one line per entry.

An optional flag (“<space> b” or “<space> B”) may be placed after the filename. This specifies that the file should be written to the archive without translation (same as the “-p” option, except “-p” applies to all non-indexed files in the list).

For example, if the file “list” contains two lines “file1” and “file2 b”, then specifying “-l list” will cause “file1” and “file2” to be written to the archive, with “file2” written as a binary file. This option is useful on machines that do not allow standard input to be redirected.
- m** Causes **vio** to restore the file’s modification time from the archive along with the other file attributes.
- n** Causes **vio** to assign a new owner (the current user) to extracted files. This is particularly useful when you are transferring files to another machine, because the original user ID is probably not meaningful in this case.
- p** Causes non-indexed files to be treated in a pure (binary) form. This prevents any text file conversion from occurring. **vio** stores its archives in a standardized format (similar to a UNIX text file). When it’s creating archives, it converts any non-indexed file to this format unless this option is specified. (This option applies to all non-indexed files in the list and thus behaves as if you had specified “<space> b” for every non-indexed file in the list, even if you did not.)
- r** Causes indexed files to be treated as raw data files. No conversion is done when the file is written to the archive. This should be done only if the archive is going to be read by a binary-compatible indexed file system. Note that all Vision Version 5, 4, and 3 files are binary-compatible, so you can use this option to move Vision 5, 4, and 3 files. Specifying this option will speed up **vio**, so it is a reasonable option to use if you are doing backups.

-s In the output mode, this allows you to specify the size of the media. This is useful on a few machines that do not detect end-of-media correctly. The size is specified as the next command-line argument. This should be the number of blocks to place on the media. Normally, a block is 512 bytes, but the “-k” option causes blocks to be 1024 bytes in size. **vio** will not place more than this many blocks on the output media before changing volumes. For example:

```
-s 2400
```

could be used to store 2400 blocks per diskette.

In the input mode, this option allows you to skip volumes. This is useful if **vio** dies due to a media error and you want to recover files on successive volumes. This option causes **vio** to start with whichever volume it finds physically mounted.

-t Causes **vio** to print the titles of the files in the archive rather than extracting the files. If this is specified with the “-v” option, long information is printed about each file.

-u Causes **vio** not to do a translation of filename directory separators.

vio by default changes all filenames to use forward slashes as directory separators. This is done to avoid problems in cases when an archive is made on a Windows machine with filenames that use backslashes (\), and then extracted on a Unix machine. The files extracted would not be stored in directories, but would instead be created with the backslashes in the names, causing problems for the user who had to work with these files.

For example,

```
vio -ovbulf listfile archive.vio
```

causes **vio** to not translate any backslashes in filenames listed in listfile to forward slashes. Similarly,

```
vio -ivnduf archive.vio
```

causes **vio** to not translate any backslashes in filenames in the archive to forward slashes.

Windows versions of **vio** handle forward slashes just fine; you do not need to use the “-u” switch on those systems to have your filenames interpreted correctly. The main purpose of providing this switch is backwards compatibility.

- v Causes **vio** to be verbose about its progress. Note that when it’s extracting files from an archive, **vio** prints each name as it starts to work each file. If **vio** dies for some reason, the last name printed will not have been completely extracted.
- 2 Specify this option when you are reading an archive and want to produce an indexed file in Vision Version 2 format, rather than a Version 5 file (the default).
- 3 Specify this option when you are reading an archive and want to produce an indexed file in Vision Version 3 format, rather than Version 5 file (the default).
- 4 Specify this option when you are reading an archive and want to produce a Vision Version 4 file.
- 5 Specify this option when you are reading an archive and want to produce an indexed file in Vision Version 5 format (the default).

vio recognizes UNIX-style names on non-UNIX environments. For example, if you specify the name “../demo/compfile” on a VAX system, **vio** will treat this name as “[-.DEMO]COMPFILE.” For this reason, you should use UNIX-style names if you want to move directory structures between machines with different operating systems.

3.4.2 Windows Considerations

When you are using **vio** on an Windows machine, you can specify a diskette drive with the “-f” option. If you do this, you must specify the type of diskette you want to write. Specify one of the following letters immediately after the drive-name’s colon:

- H 1.2 MB High-density 5.25”
- 3 1.44 MB High-density 3.5”

- 9 720 KB, 9-sector, low density 3.5"
- 8 320 KB, 8-sector, low density 5.25"

If you leave this letter off, **vio** will assume a low-density, 360 KB diskette (which can be either 3.5" or 5.25").

You may not specify a diskette drive with redirection. If you write directly to a drive, all pre-existing files on that drive are lost. In addition, all directory information is lost. In addition, the diskette will not be usable by Windows until it is reformatted.

3.4.3 *vio* Examples

Suppose that you have a list of files that you want to move to another machine using some compatible media. You could use the following **vio** command to create the media:

```
vio -ovblf listfile device
```

For each line inside the "listfile" there cannot be any spaces before or after the file name. The correct form for this file is:

```
filename(newline)  
filename(newline)  
filename(newline)
```

Do *not* include lines with spaces (initial or in the middle), such as:

```
filename (newline)
```

or with leading spaces, such as:

```
filename(newline)
```

On the target machine, you can read the archive you just created with:

```
vio -ivndf device
```

Assuming that this archive was on a 1.2 MB floppy, you could read this on a Windows machine with:

```
vio -ivndf a:h
```

Now let's assume that you want to move a set of Vision indexed files to another machine, but you do not have any common media. You plan to use a network or modem-transfer to get the files to the target machine, but you have a problem because the indexed file format on the two machines is different. You can use **vio** to help you in this case by writing the archive to a disk file with this command:

```
vio -ovblf listfile diskfile
```

The "listfile" must not have spaces before or after the file names.

Then you move "diskfile" to the target machine and use **vio** to create new indexed files in the correct format with this command:

```
vio -ivndf diskfile
```

3.4.4 Known Limitations

If you attempt to write to a write-protected diskette on a Windows system, **vio** incorrectly believes that 10 records are written to the diskette, and then it prompts for a new diskette. When this happens, the archive is incorrect and you must start over. Reading from write-protected diskettes works correctly.

vio will transfer indexed data files to/from VMS, but it will not convert them. If you must do this, you will have to unload and reload the records yourself.

On VMS, the "-n" option is always implied.

Be careful when using full path names. Some operating systems do not translate them in the way you might expect. You should always use relative path names when transferring files to a different operating system. Always make sure you have permissions to create files, and subdirectories if necessary, when you are transferring archives.

When using the "-s" option, you can suggest up to a maximum of 99999 blocks. This number corresponds to 50 MB if the block size is 512 bytes, and 100 MB if the block size is 1024 bytes.

3.5 Indexed File Record Editor (alfred)

As of Version 8.0, the Indexed File Record Editor (alfred) is provided as a sample program and is located in the “sample” folder under “AcuGT”. You can download detailed information on using alfred in PDF format from the **Support > Examples & Utilities > Acucorp Technical Articles and Tips** section of the Micro Focus website (www.microfocus.com).

3.6 logutil

You can use the utility program **logutil** to examine and edit an ACUCOBOL-GT transaction log file. This utility is used only with log files built for the Vision file system. You can run **logutil** from the operating system command line with the following usage:

3.6.1 Syntax and Options

Syntax

```
logutil[ -filv ] [ -d begin_date [ end_date ] ]
[ -t begin_time [ end_time ] ] [ -u user ]
[ -r begin_location [ end_location ] ] [ -h num ]
[ -e new_log_file_name ] log_file_name
```

Options

- f** full listing, lists selection on standard output (implies -i).
- i** prints summary information at end of listing.
- l** report location information.
- v** verbose option, includes record images (implies -f).
- d** limits selection by date.
- t** limits selection by time. Uses 24-hour clock.
- u** selects transactions for a particular user only.
- r** selects transactions within the two locations you provide

- h** num is the frequency with which header lines will be printed.
- e** extracts selected section into a new log file.

If you do not specify any options on the command line, **logutil** acts as if “-i” were specified and prints only summary information.

-v option

If the “-v” option is used, record images are displayed in a format similar to the following:

Record Image:

```
0015 0001 2ce2 dffc 0000 55dd0000 646f ...,.....U...do
7669 6400 6163 7563 6f62 6f6c 00 vid.acucobol.
```

-i option

logutil may be used to monitor transaction log activity. If you run it with only the “-i” option, or with no options, it sends a summary report to standard output. This report contains statistics, version information, and warning messages. One or both of the two warning messages below may appear, as shown in the following report:

```
logutil corruptlog
Log File           : corruptlog
WARNING: COMMIT BEGIN WITHOUT MATCHING COMMIT END IN LOG FILE
WARNING: START TRANSACTIONS WITHOUT MATCHING ROLLBACKS
      OR COMMITS
Total Size         : 2366 bytes
Number of Records  : 79
Mean Record Size   : 29 bytes
Number of Transactions : 17
Mean Transaction Size : 139 bytes
Record Version(s)  : 1
```

The warning “COMMIT BEGIN WITHOUT MATCHING COMMIT END IN LOG FILE” means that the last record in the log file is not a type CE (Commit End) record. This means that, during a commit:

- the log file was being updated at the time **logutil** was run
- or a process was killed with an uncatchable signal

- or a system failure occurred

In the case of a killed process with an uncatchable signal, or a system failure, the next START TRANSACTION using the corrupted log file will return TRANSACTION-STATUS 12.

The warning “2 START TRANSACTIONS WITHOUT MATCHING ROLLBACKS OR COMMITS” means that there are two transactions that have yet to be committed or rolled back. This could indicate a problem if there are no runtime processes currently using the log file.

-d option

The logutil utility date filter, “-d” command-line option, requires you to specify years in the 4-digit format. If you enter a year value less than “1900”, **logutil** reports “logutil: use 4 digit year specification”.

logutil example #1

To list all records for a user named “randy” that were written on November 11th between 4:50 P.M. and 5:00 P.M. to a log file called “mylog”, you would use the following command:

```
logutil -fu randy -d 11/11 -t 16:50 17:00 mylog
```

You will see something similar to the following report:

<u>TY</u>	<u>PID</u>	<u>Term</u>	<u>Client</u>	<u>User</u>	<u>Date/Time</u>	<u>File ID</u>	<u>Filename</u>
ST	21981	tty0	acucobol	randy	11/11 16:50:12		
CB	21981	tty0	acucobol	randy	11/11 16:50:12		
MA	21981	tty0	acucobol	randy	11/11 16:50:12		test.dat
OP	21981	tty0	acucobol	randy	11/11 16:50:12	07700001	test.dat
CE	21981	tty0	acucobol	randy	11/11 16:50:12		
ST	21981	tty0	acucobol	randy	11/11 16:56:40		
CB	21981	tty0	acucobol	randy	11/11 16:56:40		
WR	21981	tty0	acucobol	randy	11/11 16:56:40	07700001	
DE	21981	tty0	acucobol	randy	11/11 16:56:40	07700001	
CE	21981	tty0	acucobol	randy	11/11 16:56:41		
ST	21981	tty0	acucobol	randy	11/11 16:59:20		
CB	21981	tty0	acucobol	randy	11/11 16:59:20		
WR	21981	tty0	acucobol	randy	11/11 16:59:20	07700001	
RE	21981	tty0	acucobol	randy	11/11 16:59:21	07700001	
CE	21981	tty0	acucobol	randy	11/11 16:59:21		
-							

```
End of log.
Total Size           : 580 bytes
Number of Records    : 15
Mean Record Size     : 38 bytes
Number of Transactions : 3
Mean Transaction Size : 193 bytes
Record Version(s)    : 1
```

In a transaction log report, path and file names are limited to 17 characters without the “-l” option, or 21 characters with the “-l” option. Should the path and file name exceed that limit, the report will attempt to display all of the file name. If room permits, this will be followed by the file’s parent directory, root directory, and subdirectories. Path name components that must be omitted are represented by an ellipsis (...).

logutil example #2

To create a new log file called “newlog” that will contain the records reported above, use the “-e” option as follows:

```
logutil -u randy -d 11/11 -t 16:50 17:00 -e newlog mylog
```

3.6.2 logutil Report Headings

The first column of the standard report has the heading “TY”. Its value is the record type, taken from the following list:

ST	Start Transaction
CB	Commit Begin
CE	Commit End
RO	Rollback Transaction
DE	Delete (record)
RW	Rewrite
WR	Write
OP	Open (Opens an existing file)
MA	Make (Creates or Recreates a file during an OPEN operation)
CL	Close

CP	Copy
RN	Rename
RM	Remove (file)

The other columns are as follows:

PID	ID of process which wrote the record
Term	Name of terminal used by the runtime
User	User name of owner of the runtime
Client	Host name of machine running the runtime, the client machine when using AcuServer
Date/Time	Date and Time the event occurred
File ID	Unique identifier of the file
File Name	Name of file being opened, created, recreated, deleted, renamed, or copied

The PID is usually less than six characters on UNIX machines. On Windows, however, the PID can be a long negative number. In order for the output file to fit within 80 columns, all PID numbers are truncated to show only the right-most six characters.

If the “-l” option is used:

Location	Byte offset of the record in the log file
Length	Length of the log record

3.7 The Profiler

To help you tune application performance, the runtime includes an execution profiling facility. This built-in facility is activated when a properly prepared program is executed with the “-p” flag, prompting the runtime to collect information about I/O operations and CALLs, and to install a timer to track the amount of time spent in different parts of the code. All of this information is placed into an output file called “acumon#.xml”. (The “#” is an

automatically incremented number, starting at 1, appended to the filename to ensure that the profile data is not accidentally overwritten by another execution of the profiler.)

Note: Because the runtime performs a linear search to determine the next available filename to use, if a directory contains a large number of profiler output files, the search can take some time. For this reason, it is a good idea to remove unneeded XML profiles regularly.

The raw data in “acumon#.xml” can be processed by the **acuprof** utility to create a text-based performance report, “acumon.rpt”. In all environments, the report summarizes the amount of processor time used by each program in an application and each paragraph in a program, as well as detailing the file I/O operations performed by each program. When the “acumon#.xml” file is created by a UNIX/Linux runtime, the final report also contains information about the amount of user time spent in each program and paragraph.

3.7.1 Using the Profiler

The profiler is optimized for batch programs, and is especially useful with batch programs that run large numbers of transactions. It is more difficult to get good information from interactive programs. If user wait times are the issue you’re trying to solve, trace files are more likely to return useful information than the profiler.

When you prepare to use the profiler, you should make an effort to run your application as cleanly as possible. This means making sure that your system isn’t overloaded with large numbers of users, heavy system traffic, and so on. The cleaner the run, the more useful the information returned by the profiler.

The following steps describe how to perform profiling using default profiler and **acuprof** behavior. Options for configuring both the profiler and **acuprof** appear in the next section.

1. Compile your COBOL programs for debugging.

You must compile with at least the “-Gy” option (to include at least minimal symbol information in the object file) for the profiler to contain paragraph information. It is preferable to compile for full symbol information (“-Gs”) or for full source debugging (“-Gd”).

2. Execute the program with the “-p” runtime option to create the “acumon1.xml” file.

If a file called “acumon1.xml” already exists in the program directory, the profiler automatically changes the file name to create a file called “acumon2.xml”, “acumon3.xml”, and so on. This is intended to make it easier to compare multiple profiles of the same program.

Note that the automatic naming scheme uses the first unused number when naming the file. This means that if a directory contains files called “acumon1.xml”, “acumon2.xml”, and “acumon6.xml”, the next profile created in that directory is called “acumon3.xml”.

3. Use the command **runcl acuprof -a *pathname*** (where *pathname* is the full path and file name of the XML file created by the profiler) to launch **acuprof** and process the profiler data.

By default, **acuprof** creates a report file called “acumon.rpt” in the execution directory, then displays a message to indicate that the report was created successfully.

4. Click **OK** to end **acuprof** execution, then open the newly created report file in the text editor of your choice.

3.7.2 Configuring the Profiling Tools

Using a combination of runtime flags, **acuprof** flags, and configuration variables, you can customize the behavior of both the profiler and the **acuprof** utility. This section describes the various configuration options.

PROFILE_TYPE runtime configuration variable

This configuration variable provides an optional method of profiling ACUCOBOL-GT on Windows called “COUNTER”. The counter method uses the debugger to perform counting and appears to provide the most accurate results in Windows environments.

Set the `PROFILE_TYPE` configuration variable to either “ASYNCH” or “COUNTER”. When set to the default value of “ASYNCH”, the runtime performs profiling the way it historically has. When set to the value “COUNTER”, the runtime uses this method of profiling. Note that your COBOL programs must be compiled with “-Gd” as well as “-Gs” options to use the counter method.

The counter method is also available on UNIX and can be used if profiling your COBOL results in a message similar to “profile timer expired”. This method doesn’t completely solve that problem, but does substantially mitigate it.

Configuring profiler behavior

In order to reduce file size and processing time for the “acumon#.xml” file, the profiler does not create records for paragraphs that have a zero execution count and zero execution time. If you would like to have these zero count paragraphs recorded, use the “-p0” runtime flag in place of “-p”.

To specify a name other than “acumon#.xml” for the XML output file, use the configuration variable `ACU_MON_FILE`. This variable also takes the following specifiers for adding additional information to the name:

- %p** If the name contains the string “%p”, that string is replaced with the process ID (PID) of the runtime.
- %d** If the name contains the string “%d”, that string is replaced with the current date in the form YYYYMMDD where YYYY is the year, MM month and DD day.
- %t** If the name contains the string “%t”, that string is replaced with the current time in the form HHMMSSTTT where HH is the hour, MM minute, SS second and TTT milliseconds.
- %u** If the name contains the string “%u”, that string is replaced with the username.
- %h** If the name contains the string “%h”, that string is replaced with the hostname.

For example:

```
ACU_MON_FILE profile%p.xml
```

would produce a file called something like “profile314.xml”, where “314” is the runtime process ID.

Configuring acuprof

The **acuprof** utility takes the following flags:

Flag	Description
-a <i>or</i> --call-name	If you have specified a name other than “acumon#.xml” for the profiler output file, use this flag to pass the correct name to the acuprof utility.
-o <i>or</i> --output	To give the report file a name other than the default, “acumon.rpt”, use this flag.
-c <i>or</i> --sort-count	Sort data in the report file by the entry count for each program and paragraph. (By default, the report is sorted by time.)
-n <i>or</i> --sort-name	Sort data in the report file alphabetically by program and paragraph name. (By default, the report is sorted by time.)
-q <i>or</i> --quiet	This flag is used to suppress the “report complete” message used to indicate that acuprof has run successfully.

For example:

```
runcbl acuprof -a profile.114 -o report.114 -n
```

Here, **acuprof** parses a profiler output file called “profile.114” and produces a report called “report.114”, sorted by program and paragraph name.

3.7.3 Understanding the Report

The report is divided into three sections:

1. The first section contains general information about when the program was run, which version of the runtime was used, and general system capabilities.

Profile run on Fri Feb 06 10:05:15 2006, sorted by name
 ACUCOBOL-GT version 8.0.0 (2006-05-10)
 Timer interval = 10.029 milliseconds

Note that the runtime uses the best timer that it can get from the system, which generally means an interval around ten milliseconds (100 “ticks” per second). As a result, it’s best to run the application for at least ten seconds (not counting time waiting in an ACCEPT loop for user interaction) to get a useful number of data points.

2. The second section contains information about the programs executed.

Pct	Secs	Count	I/O	Program
36.7%	8.35	57927	0	PDM0425
32.8%	7.47	1	38950	TRP140
15.2%	3.46	41947	0	TRA050A
14.2%	3.24	57927	0	TRS130B
0.7%	0.15	2	8	PCM1800
0.3%	0.07	14	15	TRZCG01B
0.0%	0.00	1	0	PCM1520

This condensed information gives you an easy way to see which programs to focus your attention on. In general, you will want to start by tuning the programs in which the most time is being spent.

Because this example was generated on a Windows system, it doesn’t show a comparison of system time (time spent performing I/O operations and doing memory management) and user time (time spent in the application, running PERFORMs, etc.). On UNIX systems, this additional information is included and can be used to help you figure out where to focus.

3. The third section (which contains the bulk of the information in the report) has information about the paragraphs executed by each program.

In this section, the paragraph totals are per program, not per application, so the total for all paragraphs in each program should add up to 100%.

TRP140

Opens: 1
 Reads: 38949

Pct	Secs	Count	Total	Paragraph
-----	------	-------	-------	-----------


```

=====
52.1%      3.89      41947      17.1%     Z70-CALL-TRA050A
40.5%      3.03      38949      13.3%     Z10-READ-FTR013A
 5.6%      0.42      38948       1.9%     B20-PROCESS-ECR
 0.5%      0.04       2996       0.2%     B20-PROCESS-EO
 0.4%      0.03         14       0.1%     Z40-CALL-TRZCG01B
 0.3%      0.02          1       0.1%     A00-MAINLINE
 0.3%      0.02          1       0.1%     A10-DEBUT-PROG
 0.3%      0.02          0       0.1%     Z99-END

```

In most COBOL programs, one or two paragraphs use the lion's share of the time. There may be another paragraph or two that takes up a moderate amount of time, but most paragraphs use a very small percentage of the total program time.

Note that you may find very small paragraphs (like the EXIT paragraph) getting a very large number of counts (CALLs). Because the time spent counting each CALL is added to the paragraph time, it may appear that such paragraphs are taking a large amount of time, when in fact the behavior of the timer is artificially inflating the paragraph time.

3.7.4 Understanding the XML Data File

The **acuprof** utility takes the raw data in “acumon#.xml” and combines the individual data points to create useful aggregates in the report file. **acuprof** is an ACUCOBOL-GT program that is located in the “tools” subdirectory.

Because “acumon#.xml” is a straightforward XML file, any tool that can parse XML can parse the raw report. This means that you can bring the report into recent versions of Microsoft Excel, for example, or create your own parsing tool using the C\$XML routine to return the information most useful to you.

This section contains the basic information that you need to understand the data collected in “acumon#.xml”. The “ticks” timer

In the final report, program time is reported in seconds, or fractions of a second. The raw XML file, however, counts user and processor time in “ticks”. The length of a tick is system-dependent, but usually equals about (10-milliseconds). The precise amount of time in each “tick” is reported at the beginning of the XML file (as described below). Each time the timer

starts, the runtime examines the current program location and records a tick for the current program and current paragraph. By looking at how many ticks a program or paragraph accumulates, you get a real-time sampling of where the run spent its time.

If a program is running multiple threads and there is only one timer, when the timer expires, a tick is given to the current program and current paragraph, regardless of which thread is running.

The timer runs in “process time” on machines that support the concept (UNIX). Process time is CPU time spent for the particular process and bears little relationship to real time. On other machines (Windows NT), a real-time timer is used instead. For these machines, it is important to run as few other tasks as possible while collecting profile data.

Structure of the raw report file

The structure of the XML file is similar to that of the final report. It contains general information about a specific execution of the application, followed by information about each program and each paragraph in the program. Because, however, the XML file contains raw data instead of aggregate information, it is more useful to think of the file as divided into “levels” rather than sections. The top level (outermost set of XML tags) contains general execution information. The next level (middle set of XML tags) contains program information. The last level (innermost set of XML tags) contains information about paragraphs in each program.

The runtime level

The following table defines the HTML tags at the runtime level.

<Runtime-Version>	Shows the version number of the runtime used by this particular profiling run. The value is the full version number (including any information seen in “runcbl -v” such as build dates or patch numbers).
<Run-Date>	Marks the date and time of the run that produced the profile data.

<Has-Timer>	Is set to “1” if the runtime has support for profiling timers, “0” if not. If support is not available, the various ticks fields below will all be zero. Currently, timer support is available under Windows NT and UNIX machines that have “setitimer” and “siginterrupt” routines.
<Usecs-Per-Tick>	Contains the number of microseconds represented by each tick of the timer. The runtime normally asks for a 10-millisecond timer (a value of 10000 for this field).
<User-Time-Msecs>	When available, this tag shows the number of milliseconds of CPU time spent processing user code for this run.
<System-Time-Msecs>	When available, this tag shows the number of milliseconds of CPU time spent processing system code on behalf of the profiled process.

The program level

The <Program> tag marks the root of a subtree of information for each program used by the profiled run. Each time a program leaves memory, it produces one of these subtrees. Because of this architecture, a particular program can appear in the “acumon#.xml” file more than once.

If a particular program appears many times in a run, it may be getting canceled too often. This can present performance issues, because each cancel causes the program to be reloaded from disk the next time it is called.

Each program sub-tree contains the following tags:

<Program-Name>	Contains the program’s ID.
<Call-Name>	Shows the name the program was called by. This is useful if more than one program has been given the same program ID.

<Object-Code>	Gives the name that describes the object code instruction set. The name “AcuCode” is used for machine-independent object files. If the object was compiled for native code, the name of the relevant CPU type is given.
<Call-Count>	Indicates the number of times this program was entered.
<Program-Ticks>	Shows the number of times the timer went off in this program. Time spent waiting for the user to respond is counted only on Windows NT systems.
<Has-Symbols>	Is set to “1” if the program was compiled for debugging and had section/paragraph symbols available. When this is “0”, no paragraph data is included for this program.
<File-Opens>	Lists the number of times this program opened any file using the OPEN statement or ISIO. Note that this counts opens, not files (so if you open the same file ten times in the course of a run, that counts as “10” and not “1”). Some machines open files much more slowly than others, so a large number here usually suggests a potential performance issue.
<File-Reads>	Lists the number of record read attempts. READ, READ NEXT, and READ PREVIOUS all count here and are not distinguished.
<File-Writes>	Indicates the number of records written.
<File-Rewrites>	Gives the number of records rewritten by the program.
<File-Deletes>	Shows the number of records deleted by the program.
<File-Starts>	Lists the number of file positions made using the START statement or the ISIO subroutine. Note: Attempted reads, writes, rewrites, deletes, and starts that failed are counted along with the successful file operations.
<File-Commits> and <File-Rollbacks>	Show the number of COMMIT and ROLLBACK statements performed by the program, regardless of outcome.
<Records-Sorted>	Gives the number of records sorted by this program via the SORT statement.

The paragraph level

<Paragraph> indicates the root of a subtree of information about a paragraph contained in the program. If the program has been compiled with debugging, there will be one of these for each Procedure Division section or paragraph in the program.

Each paragraph sub-tree contains the following tags:

<Name>	Gives the name of the paragraph or section. This appears in uppercase, regardless of the case seen in the actual source code.
<Count>	Shows the number of times this paragraph was entered, by any means.
<Ticks>	Indicates the number of times the timer went off while in this paragraph. Time spent waiting for the user to respond is counted only on Windows NT systems. Time spent in between paragraphs or programs may count for either the caller or the called routine, depending on the timing.

3.8 External Sort Utility – AcuSort

The **AcuSort** utility enables you to sort or merge Vision indexed, relative, binary sequential, and line sequential files. An alternative to using the SORT verb, this external sort function is invoked from the command line. **AcuSort** instructions may appear directly on the command line, or they may be included in a separate text file. This section outlines **AcuSort** utility functions. Details on the SORT verb may be found in section 6.6, “Procedure Division Statements,” in Book 3, *ACUCOBOL-GT Reference Manual*.

3.8.1 AcuSort Command Format

You can specify **AcuSort** instructions in one of two ways. You can include them on the **AcuSort** command line, as follows:

```
acusort parameters
```

where *parameters* are the various **AcuSort** utility options that control such operations as SORT and MERGE. This format is appropriate if you want to execute a simple sort with few parameters. Note that if you choose this method, you must ensure that the command line size and contents do not violate any operating system or shell limits.

If you need to execute a sort that is often repeated or one with a large number of options, you may find it easier to store and use sort instructions in a text, or *take*, file. In this case, the **AcuSort** command line format is

```
acusort take filename
```

where *filename* is the file that contains the options to use when **AcuSort** is executed.

The take file would contain all the instructions for your sort or merge process. The text file may also contain comments (indicated by an asterisk at the beginning of a line). An example of a take file appears in **section 3.8.3, “Code Sample.”**

Specifying the “-v” option on the **AcuSort** command line as shown below causes the utility to display version and copyright information.

```
acusort -v
```

3.8.2 AcuSort Instructions

Several options are available for use with the **AcuSort** utility, including various instructions to sort or merge files, specify the name of an input or output file, or define conditions under which certain records are included or excluded from a sort or merge process. An exit status of zero indicates success; non-zero indicates failure.

The following sections provide details on these functions. Refer to **section 3.8.3, “Code Sample,”** for **AcuSort** sample code.

3.8.2.1 CHAR-ASCII and SIGN-ASCII

The CHAR-ASCII instruction tells **AcuSort** that the data should be interpreted as ASCII characters. SIGN-ASCII instructs **AcuSort** to use the ASCII sign convention. These keywords mirror the operation of the

CHAR-EBCDIC and SIGN-EBCDIC keywords (described in the next section) and enable you to switch back and forth between the different modes. This means you can put multiple SORT/MERGE operations in a single “take” file, which use different character sets or sign modes. *AcuSort*’s defaults are CHAR-ASCII and SIGN-ASCII mode.

3.8.2.2 CHAR-EBCDIC and SIGN-EBCDIC instructions

The CHAR-EBCDIC instruction tells **AcuSort** to expect data that is encoded in the EBCDIC character set rather than ASCII. SIGN-EBCDIC tells **AcuSort** that numeric DISPLAY types that include signs should be interpreted according to the EBCDIC convention. The use of CHAR-EBCDIC implies SIGN-EBCDIC.

For example, if your data is EBCDIC, you should use CHAR-EBCDIC. If you have ASCII data with EBCDIC sign encoding, you should use SIGN-EBCDIC. If you have ASCII data with ASCII sign encoding, you would not use either instruction.

CHAR-EBCDIC and SIGN-ASCII are incompatible options. If CHAR-EBCDIC is specified and **AcuSort** is in SIGN-ASCII mode, the sign mode will be forced to SIGN-EBCDIC. If SIGN-ASCII is specified and **AcuSort** is in CHAR-EBCDIC mode, the char mode will be forced to CHAR-ASCII.

3.8.2.3 SORT/MERGE instructions

The SORT and MERGE instructions specify whether to perform a sort or merge operation. These two functions are mutually exclusive. A SORT or MERGE instruction must be followed by a FIELDS phrase that indicates the fields on which a file is to be sorted or merged. You specify the start position, the length, the type, and the order for each sort field. Use a comma to separate field attributes and a comma before starting to describe a new field. A merge operation combines records from files that are already sorted on the specified fields. The syntax for these functions follows:

```
acusort sort fields(start, length, type, order)
```

```
acusort merge fields(start, length, type, order)
```

where

- start* is the offset of the field in the record (in bytes, starting at position 1).
- length* is the size of the field in bytes.
- type* is a two-letter code indicating the type of data in the field (see the data field type descriptions below).
- order* is the order of output, either ascending (A) or descending (D).

The following data field types are supported in **AcuSort**:

BI	Unsigned numeric, USAGE COMP
C5	Unsigned numeric, USAGE COMP-5
C6	Unsigned numeric, USAGE COMP-6
CH	Alphanumeric
CX	Usage COMP-X
FL	Usage floating point
LI	Signed numeric, SIGN IS LEADING
LS	Signed numeric, SIGN IS LEADING SEPARATE
NU	Unsigned numeric
PD	Signed numeric, USAGE COMP-3
SB	Signed numeric, USAGE COMP
S5	Signed numeric, USAGE COMP-5
TS	Signed numeric, SIGN IS TRAILING SEPARATE
TI	Signed numeric, SIGN IS TRAILING

In the following example:

```
acusort sort fields (1, 10, ch, a)
```

the sort operation begins at position 1 in a 10-byte sort field, and the alphanumeric data are sorted in ascending order. Refer to **section 3.8.3, “Code Sample,”** for more sample code.

Note: When using LI or TI data types, you should be aware of the sign storage for your data. The **AcuSort** utility supports both IBM and Micro Focus sign storage. In ACUCOBOL-GT, use the “-Dci” compile option to specify IBM sign storage, or the “-Dcm” option to specify Micro Focus sign storage. If you use IBM sign storage and your data is ASCII, use the SIGN-EBCDIC instruction in your take file of **AcuSort** options. See **Section 2.2.10** in Book 1, *ACUCOBOL-GT User's Guide* for more information about data storage compile options. See **Section 5.7.1.8** in Book 3, *ACUCOBOL-GT Reference Manual*, for details on how signs are stored when the various compile options are used.

3.8.2.4 USE/GIVE instructions

The USE and GIVE instructions specify the name and characteristics of the input file and output file, respectively, of a sort or merge process. Note that you must specify all USE instructions before any GIVE instructions. The input and output file descriptions include ORG, RECORD, and KEY phrases, which define the file's characteristics. The syntax for these instructions is as follows:

```
use input-file
   org file-type
   record format, record-length [, max-length]
   key(key-structure)

give output-file
   org file-type
   record format, record-length [, max-length]
   key(key-structure)
```

where

input-file is the pathname of the input file. For file names containing spaces, surround the filename with double quotes (“ ”). If the filename contains double-quote characters, specify these by doubling the double-quote characters (“” “”).

Examples:

filename: Work File

notation: "Work File"

filename: Embedded"Quote

notation: "Embedded""Quote"

output-file is the pathname of the output file. the same rules regarding input-file names with spaces applies to output-file names.

file-type specifies the type of input or output file: indexed (IX), relative (RL), line sequential (LS), or binary sequential (SQ).

format indicates that the file contains fixed length records (F) or variable length records (V).

record-length specifies the record length for a fixed length record or the minimum record length for a variable length record.

max-length specifies the maximum record length for a variable length record.

key-structure specifies the key structure for an indexed file. Refer to the following section for information about key structure.

Note: If the input and output files have the same organization and/or record information, you need not specify these options for each file. The **AcuSort** utility applies the most recent RECORD and ORG options to subsequent files when these options are not specified for the files.

KEY structure

For each key or key segment, you must specify the start position, the length, and the key type, as defined below. Use a comma to separate field attributes and a comma before starting the description of a new key or key segment.

The following command specifies the key structure for an indexed file:

```
key (start, length, key-type, ...)
```

where

start is the offset of the record key (in bytes, starting at position 1).

length is the size of the key in bytes.

key-type is a code indicating the key type (see the list of key types below).

You can specify one of the following key types in the KEY statement:

- P Primary key
- PD Primary key with duplicates allowed
- A Alternate key
- AD Alternate key with duplicates allowed
- C Key segment belonging to the primary or alternate key previously described

The following sample describes the key structure for an indexed file with three keys:

```
key (502, 98, PD, 1, 18, A, 95, 18, AD, 337, 18, C)
```

In this example, the primary key allows duplicates. Its offset is 502 and its length is 98. The first alternate key has an offset of 1 and a length of 18. The second alternate key allows duplicates, and consists of two segments. The first segment starts at offset 95 and has a length of 18. The second segment starts at offset 337 and has a length of 18.

The **AcuSort** utility always sorts duplicate records in the order in which they are encountered in the input file, a process known as a “stable sort.”

USE/GIVE example

In the following sample code:

```
use c:\acuprod\data\ordrdet1 org ix
record f 143
key (1, 36, p)
give c:\acuprod\data\ordrdet_sorted
```

the input file is “ordrdet1” and the output file is “ordrdet_sorted”. They are both indexed files with a fixed record length of 143. The primary key is 36 bytes long, starting at position 1. Complete sample code can be found in **section 3.8.3, “Code Sample.”**

3.8.2.5 INCLUDE/OMIT instructions

The INCLUDE and OMIT instructions specify conditions under which individual records may be included in or excluded from, respectively, a sort or merge process. As with SORT and MERGE, these instructions are mutually exclusive. Each SORT or MERGE instruction may have a single optional INCLUDE or OMIT conditional (COND) phrase. Syntax for these instructions is as follows:

```
omit cond (start, length, type, comparison expression)

include cond (start, length, type, comparison
expression)
```

where *start*, *length*, and *type* are as defined for the SORT/MERGE instructions, and *comparison expression* sets the conditions for a specified comparison.

Conditional constants ALL and NONE match all or none of the records, respectively. As an example, each of the following statements would result in the inclusion of all records:

```
include cond = all
```

or

```
omit cond = none
```

To omit all records, you would use one of the following statements:

```
omit cond = all
```

or

```
include cond = none
```

A default record field type may be specified for an INCLUDE/OMIT instruction by setting FORMAT to the desired type. (Refer to the table of data field types in **section 3.8.2.3, “SORT/MERGE instructions.”**) This assignment can appear either before or after the COND phrase. Record field

specifications without the type inherit the default type specified by FORMAT. A warning is issued if the default format is specified but never used.

The INCLUDE or OMIT instruction comparison expression may compare a record field against another record field or against a constant. The size of an expression is not limited. Comparison operators are:

EQ	Equal to
GE	Greater than or equal to
GT	Greater than
LE	Less than or equal to
LT	Less than
NE	Not equal to

INCLUDE/OMIT samples

If you want to include records in which the first four bytes are greater than 1000 when interpreted as USAGE DISPLAY data, you could use one of the following code statements:

```
include cond = 1 4 nu gt 1000
```

or

```
omit cond (1 4 nu le 1000)
```

To include records in which the first four bytes are the same as the second four bytes when interpreted as characters, you could use one of the following statements:

```
include cond = 1 4 ch eq 5 4 ch
```

or

```
omit cond 1 4 ch ne 5 4 ch
```

As another example, if you want to omit any record in which the first four bytes (COMP-6) are greater than the second four bytes (COMP-4), you could use one of the following statements (note the use of the FORMAT phrase in this example):

```
omit format=c6 cond = (1 4 gt 5 4 bi)
```

or

```
include cond 1 4 c6 le 5 4 format bi
```

In addition to the data types already available with SORT/MERGE, the INCLUDE/OMIT instructions may have a substring search (SS) type, which indicates that a search should be performed for the specified character constant. Only the equal to (EQ) or not equal to (NE) operators may be used in conjunction with the SS type. The designated string is either found or not found with this comparison. As an example, each of the following code statements would result in the omission of records in which the first 10 characters contain the substring “data”:

```
omit cond 1 10 ss eq c'data'
```

or

```
include cond = (1 10 ss ne c'data')
```

Constant types can be decimal, hexadecimal, or character. Decimal constants match the pattern

```
[+-]?[0-9]+
```

that is, an optional sign character followed by a number of digits. Decimal constants may be up to 76 digits long.

Hexadecimal constants match the pattern

```
x'([0-9A-F]{2})+'
```

that is, a leading “x” indicating a hex constant and then groups of two hexadecimal digits in single quotation marks. Hexadecimal constants are unsigned and may be up to 64 hexadecimal digits long (32 bytes). For example, either of the following statements results in the inclusion of records in which the first 10 characters (USAGE DISPLAY) equal 0xFFFF:

```
include cond = (1 10 nu eq x'FFFF')
```

or

```
omit cond 1 10 nu ne x'FFFF'
```

Character constants match the pattern

```
c' .+'
```

that is, a leading “c” indicating a character constant and then a number of characters in single quotation marks. Single quotation marks may be represented within the character constant by specifying two single quotation marks in a row.

In general, numeric types may be compared against each other or against a constant. Floating point data may only be compared against other floating point data. Strings may be compared against each other, a string, or a hexadecimal constant. Strings may also be used in substring searches.

Specifically, data types BI, C5, C6, CX, LI, LS, NU, PD, S5, SB, TI, and TS may be compared against each other, or against a hexadecimal or decimal constant. BI and CH may be compared against each other or a string constant. CH may be compared against a hexadecimal constant.

The AND and OR operators may be used to join comparison expressions. The AND operator takes precedence over OR. Note that the characters “&” and “|” may be used to represent AND and OR, respectively. As an example, either of the following statements results in the omission of any record in which the first character does not equal the second or the third character:

```
omit cond = (1 1 ne 2 1) & (1 1 ne 3 1) format ch
```

or

```
include format=ch cond ((1 1 eq 2 1) or (1 1 eq 3 1))
```

Parentheses may be used to determine the evaluation order of an expression. An expression is evaluated only as far as necessary to determine the inclusion or exclusion of the record. For example, if a conditional is a list of expressions joined by AND operators, and the first expression evaluates as false, the remainder of the expressions is not evaluated for this record.

3.8.3 Code Sample

The following sample code describes the SELECT and FD for an “orders-detail” indexed file:

```
SELECT OPTIONAL Orders-Detail  
          ASSIGN          TO DISK "ORDRDETL"
```

```
ORGANIZATION IS INDEXED
ACCESS MODE IS DYNAMIC
LOCK MODE IS AUTOMATIC
FILE STATUS IS ORDERS-DETAIL-STATUS
RECORD KEY IS Prime = ORDERS-DETAIL-PRIMARY-KEY
```

```
FD ORDERS-DETAIL.
01 ORDERS-DETAIL-RECORD.
   05 ORDERS-DETAIL-PRIMARY-KEY.
      10 ORDETL-CUSTOMER PIC X(10).
      10 ORDETL-DATE.
         15 ORDETL-DT-YYYY PIC 9(4).
         15 ORDETL-DT-MM PIC 99.
         15 ORDETL-DT-DD PIC 99.
      10 ORDETL-TIME.
         15 ORDETL-HR PIC 99.
         15 ORDETL-MIN PIC 99.
         15 ORDETL-SEC PIC 99.
         15 ORDETL-TH-SEC PIC 99.
      10 ORDETL-PROD-NO PIC X(10).
   05 ORDETL-DESCRIPTION PIC X(80).
   05 ORDETL-QTY PIC 9(5).
   05 ORDETL-PRICE PIC 9(9)v99.
   05 ORDETL-TOTAL-PRICE PIC 9(9)v99.
```

In this sample, we illustrate the sort of the “orders-detail” file based on three fields: orderl-customer, ordetl-price, and ordetl-description. Each field is to be sorted in ascending order, and the resulting output file includes only records in which ordetl-date is equal to or greater than May 3, 2006.

The **AcuSort** take file “paramfile1” contains the following options:

```
sort fields (1, 10, ch, a, 122, 11, nu, a, 37, 80, ch, a)
use c:\acuprod\data\ordrdet1 org ix
   record f 143
   key (1, 36, P)
give c:\acuprod\data\ordrdet_sorted
   include cond = 11 8 nu ge 20060503
```

If we include our **AcuSort** instructions in a take file named “paramfile1”, our command line would be

```
acusort take paramfile1
```


3.8.4 AcuSort Environment Variables

The following environment variables affect **AcuSort** behavior:

A_TMPDIR, TMPDIR

These variables control the location of any temporary **AcuSort** files. **A_TMPDIR** is checked first, and then **TMPDIR**. Temporary files created by **AcuSort** are placed in this directory. The default value is the current working directory.

ACUSORT_FILE_MEMORY

This variable allows you to set the maximum amount of memory in megabytes to be used for buffering I/O data with the temporary file. The default value is “1”.

The default value should be adequate for most situations. However, if very large records are in use, you may want to increase this value in order to hold several records. The higher setting allows the buffer layer to avoid doing I/O for record comparisons during a **SORT/MERGE** process.

Tip: The number of file buffer blocks is controlled by the **ACUSORT_FILE_MEMORY** environment variable. It is set in units of megabytes. The buffer block size is 4096 bytes; therefore, for each MB, you get 256 buffer blocks. One buffer block will be reserved for each sorted region in the temporary file. The number of sorted regions in the temporary file will depend on the size of the records being operated on and the amount of memory allocated for sorting with the **ACUSORT_MEMORY** environment variable. Buffers remaining after this reservation may be used for read-ahead. Up to eight buffers per region will be used for read ahead.

ACUSORT_MEMORY

This variable sets the number of megabytes of memory allowed for sorting records. Sort performance may improve as more memory is allocated for this purpose. The default value is “2”.

ACUSORT_TRACE

This variable controls the type of information written to the **AcuSort** log file. The following values determine which sets of log messages appear:

1	general program
2	record import/export
4	numeric values comparison
8	numeric values conversion
16	temporary file buffer
32	command structure
64	modes
128	parser
256	lexer

Set **ACUSORT_TRACE** to the sum of the numbers corresponding to the sets of information you want written to the log file.

Please note that much of this information is intended only for diagnostic use. You should not rely on the content of the information written to the log file, as it is subject to change without notice.

Tip: For time-critical SORTs, examine the **AcuSort** trace output (specifically the “buffer” and “import/export” categories) to see various statistics about sorted region counts and buffer usage. Adjust the memory configuration variables as appropriate.

dd_SYSOUT

This configuration variable specifies the name of the **AcuSort** log file. Various information about **AcuSort** functions is written to this file. The default value is “SYSOUT”.

USE_LARGE_FILE_API

On UNIX systems, setting this variable to “1” causes **AcuSort** to use the large file API. The default value of “0” uses the normal file API, which cannot access files larger than 2GB. This variable applies to both the **USE** and **GIVE** files, and the temporary **AcuSort** file. If the

total size of input records is less than 2GB, leave this variable set to the default of “0”. Otherwise, set it to “1”. (Note that the system must support the large file API in order for this variable to have any effect.) Windows versions of **AcuSort** can always access large files, so it is not necessary to set this variable on Windows platforms.

3.9 Remote Preprocessing Utility – Boomerang

The **Boomerang** utility program includes client and server technologies that enable you to automatically transfer files to a remote server, invoke and perform preprocessing on that server, then return the preprocessed files to your client machine where additional compiling can occur. Many proprietary or third party preprocessors have machine-specific functions that require preprocessing to occur in their native environments. **Boomerang** makes accessing these types of preprocessors easier and more efficient.

With **Boomerang** you can:

- Send source files, COPY files, and user INCLUDE files from a Windows or UNIX/Linux client to a UNIX/Linux server.
- Invoke and run popular third-party preprocessors such as those used with Oracle, DB2, UniKix, and IBM TXSeries CICS, or invoke custom-built preprocessors.
- Have preprocessed output files, error files, and status returned to your client machine.
- Use **Boomerang** with the ACUCOBOL-GT compiler's “-Pg” option to perform single or multiple preprocessing steps.

3.9.1 License Requirements and Installation

To use **Boomerang**, the client machine must have an ACUCOBOL-GT development system and corresponding compiler license. On the server, a standard runtime license and server access file is required. You can use

Boomerang to create an access file or you can use an existing access file. For instructions on setting up a server access file, refer to either the AcuConnect or AcuServer User's Guide, section 3.3.2 and 5.4.1 respectively.

The **Boomerang** client and server program (boomerang.exe) requires no special installation steps, and is automatically installed in the same directory as the runtime.

3.9.2 Server Setup and Configuration

Boomerang server setup and configuration involves four main steps:

Step 1: Creating an Alias File that contains aliases for each preprocessor you wish to invoke from your client machine.

Step 2: Creating a Configuration File and setting configuration variables accordingly.

Step 3: Creating an Access File to establish system security and access.

Step 4: Starting the Server.

3.9.2.1 Step 1: Creating an Alias File

To accomplish its preprocessing tasks, **Boomerang** references an alias file that contains preprocessor-specific commands and instructions.

To create an alias file for a preprocessor, perform the following steps on the server:

1. From a command line, navigate to where boomerang.exe is installed, type “boomerang” and press return.

The following usage information appears:

```
Server usage:
  boomerang -alias
  boomerang -access
  boomerang -kill [-n portnum]
  boomerang -start [-c config] [-e error] [-t #]
  [-f] [-n portnum]

Client usage:
  boomerang -server server[:port]
            -alias alias
            -COPY
            -include pattern [pattern ...]
            [-Po preprocessor-output-file]
            [-Pe preprocessor-error-file]
            -Sf source-file
```

2. Access the alias menu by typing the following command:

```
boomerang -alias
```

The following alias menu options appear:

```
Enter the name of the alias file:
[/etc/boomerang_alias.ini] boomerang_alias.ini

Boomerang Alias file options
1 - Add an alias entry
2 - Remove an alias entry
3 - Modify an alias entry
4 - Display alias entries
5 - Exit

Enter choice [4]: 1
```

3. Select option “1” create an alias file. The following menu appears:

```
Add an alias
Enter the alias name:
Enter the name of the precompiler:
Enter precompiler options:
Enter precompiler directives:
Enter required precompiler extension if any:
Press <Return> to continue...
```

The alias creation fields are defined as follows:

Field	Description
Alias name	The name you wish to give your alias.
Name of precompiler	The precompiler that should be used by this alias. You can also specify the name of a shell script to run instead of the precompiler name. This is necessary for some precompilers like DB2 where certain setup instructions are required before precompiling can commence. See the DB2 Alias Example provided later in this section for more details.

Field	Description
Precompiler options	<p>Instructions you wish to give to the preprocessor. Boomerang includes several keywords you can use to specify several basic files:</p> <ul style="list-style-type: none"> • B_INPUT: Input to the preprocessor. This is replaced with the name specified by the “-SF” option from the Boomerang client command. If you are executing Boomerang from the ACUCOBOL-GT compiler, the compiler automatically calls the Boomerang Client with the “-SF” option and the name of the program to be preprocessed. • B_OUTPUT: Output from the preprocessor. This is replaced with the name specified by the “-Po” Boomerang client command. By default, the ACUCOBOL-GT compiler expects the output from the preprocessor to be named “acu__pp1.out” with two underscores. If you do not specify the “-Po” option, Boomerang will replace this keyword with the default name “acu__pp1.out”. • B_ERROR: Error output from the preprocessor. This is replaced with the name specified by the “-Pe” Boomerang client command. If you do not specify the “-Pe” option, Boomerang will replace this keyword with the default name “acu__pp1.std” with two underscores. The ACUCOBOL-GT compiler automatically displays the contents of this file to the screen.
Precompiler directives	<p>Used to specify any keywords that the preprocessor recognizes as directives. Boomerang will automatically insert ACUCOBOL-GT line directives before and after these keywords. In cases involving compilation errors, this makes it easier for you to identify the offending line of code in the source file.</p>

Field	Description
Required precompiler extension	<p>Your COBOL program can have any extension you want, but some precompilers require a specific extension. If your precompiler requires a specific extension on the source file, specify it here rather than changing the extension name of your source files. Boomerang creates a temporary file on the server with the extension you specify so that preprocessing can be performed. Boomerang will then remove this temporary file. If you do not specify an extension here, it will use the extension of the source file. There is a case where the extension of the source file is not used - when you are calling two or more preprocessors. Refer to the ACUCOBOL-GT User's Guide, section 2.13.1.2 for more information on calling two or more preprocessors. In this case, the output of the first preprocessor is called "acu_pp1.out" and is used as input to the second preprocessor. If the second preprocessor requires a specific extension, you can specify the expected extension here. For example, if the precompiler requires a source file extension of ".ccp", you would need to specify ".ccp" as the required precompiler extension. Otherwise, the precompile will fail.</p>

Pro*COBOL Alias Example

```

Add an alias
Enter the alias name: alias-procob
Enter the name of the precompiler: procob
Enter precompiler options: iname=B_INPUT
oname=B_OUTPUT >B_ERROR 2>&1
Enter precompiler directives: EXEC SQL
Enter required precompiler extension if any:
Press <Return> to continue...

```

Since Pro*COBOL has the following options to specify the input and output files:

```

iname=
oname=

```


the name of the Pro*COBOL input file can be specified by the B_INPUT keyword and the name of the output file can be specified by the B_OUTPUT keyword. Since Pro*COBOL does not have an option to specify an error output file, the “>B_ERROR 2>&1” redirects the output that normally would be displayed on the screen to the file associated with the B_ERROR keyword.

“EXEC SQL” is specified as the precompiler directive since Pro*COBOL uses this phrase to begin its Pro*COBOL statements.

CICS Alias Example

```
Add an alias
Enter the alias name: alias-cicstran
Enter the name of the precompiler: cicstran
Enter precompiler options: -l ACUCOB -O B_OUTPUT
B_INPUT >B_ERROR 2>&1
Enter precompiler directives: EXEC CICS
Enter required precompiler extension if any: .ccp
Press <Return> to continue...
```

The “-l ACUCOB” option is required with CICS. This parameter tells cicstran to precompile the source file in a manner that is compatible with ACUCOBOL-GT.

Since CICS uses the “-o” option to specify the output file, it can be specified by the B_OUTPUT keyword. The name of the input file is specified by the B_INPUT keyword. CICS does not have an option to specify an error output file, the “>B_ERROR 2>&1” redirects the error output from the screen to the file associated with the B_ERROR keyword.

“EXEC SQL” is specified as the precompiler directive since CICS uses this phrase to begin its CICS statements.

CICS requires that the source file have an extension of “.ccp” so it is specified as the required precompiler extension.

UniKix Alias Example

```
Add an alias
Enter the alias name: alias-unikix
Enter the name of the precompiler: kixclt
Enter precompiler options: -O B_OUTPUT B_INPUT
>B_ERROR 2>&1
Enter precompiler directives: EXEC CICS
Enter required precompiler extension if any: .cl2
Press <Return> to continue...
```

Since UniKix uses the “-o” option to specify the output file, it can be specified by the B_OUTPUT keyword. The name of the input file is specified by the B_INPUT keyword. UniKix does not have an option to specify an error output file, the “>B_ERROR 2>&1” redirects the error output from the screen to the file associated with the B_ERROR keyword.

“EXEC SQL” is specified as the precompiler directive since UniKix uses this phrase to begin its CICS statements.

UniKix requires that the source file have an extension of “.cl2” so it is specified as the required precompiler extension.

If there are COBOL COPY statements in the source file UniKix requires these files to exist on the server where kixclt is run. You can use the Boomerang “-COPY” command on the client side to instruct Boomerang to copy the COPY files to the server and to use them in the preprocessing phase.

UniKix requires that you have the following three environment variables set:

UNIKIX

PATH

COPYPATH

See your UniKix documentation for information on setting these variables. Boomerang requires setting these variables before starting the Boomerang server. If you use the “-Sf” or “-COPY” client commands to send COPY files from the client to the server, be sure you add the directory where the Boomerang server resides to the server COPYPATH environment variable so

that these COPY files can be found by the preprocessor. The files specified by the “-SP” or “-COPY” commands get copied into the directory where the Boomerang server resides.

DB2 Alias Example

```
Add an alias
Enter the alias name: alias-db2
Enter the name of the precompiler: db2prep.sh
Enter precompiler options: database B_INPUT B_OUTPUT
B_ERROR
Enter precompiler directives: EXEC SQL
Enter required precompiler extension if any: .sqb
Press <Return> to continue...
```

DB2 requires some setup before running its precompiler. You can perform the necessary setup by specifying a shell script file (db2prep.sh in this example) to run instead of specifying the name of the precompiler. The shell script performs the necessary setup and then starts the precompiler. The Boomerang keywords beginning with “B_” are passed to the shell script by specifying them at the precompiler options line. “EXEC SQL” is specified as the precompiler directive since DB2 users “EXEC SQL” to begin its DB2 statements. The DB2 precompiler requires that the input file have an extension of “.sqb” so this is specified at the required precompiler extension line.

Using the example above, when the Boomerang server runs the shell script it looks something like this:

```
db2prep.sh database ACCT01.sqb acu_pp1.out acu_pp1.std
```

Note that the precompiler options must be specified in the order that the shell script expects. The following is an example of a DB2 shell script used by the Boomerang server:

```
#!/bin/ksh
```

```
# db2prep.sh - This script is designed to be called from the
# Boomerang server. For this script the Boomerang server alias
# file would need to have the following precompiler options
# using the Boomerang file keywords:
#
```

```
# Precompiler-Options: database B_INPUT B_OUTPUT B_ERROR
#
# In this script the Boomerang keywords get mapped to the
# following script variables:
#
# database = $1 the name of the database to connect to
# B_INPUT  = $2 the precompiler input file
# B_OUTPUT = $3 the precompiled output file
# B_ERROR  = $4 the precompiler error file

# Execute the DB2 configuration file
. /home/db2inst1/sqllib/db2profile

# DB2 precompile -- takes .sqb as input, outputs .cbl
echo =====
echo Begin output from \"db2 prep\" SQL precompiler:
echo =====

# Connect to the database
db2 connect to $1 >$4 2>&1

# Execute the precompiler on $2 sending any error output to $4
# capture the return code in $returnCode
db2 prep $2 target ansi_cobol >>$4 2>&1
returnCode=$?

# Boomerang expects the precompiled output file to be the name
# specified by $3. The precompiled output file created by DB2
# is the name as the source file but with an extension of .cbl.
# We need to remove the extension from the input file, $2, and
# add a .cbl extension so that we can copy it to $3.
# The following command removes the "." and everything past it
# to create the prefix.
prefix=`echo $2 | sed -e "s/\..*$//"`

# Copy the precompiled output file to the name that Boomerang
# is expecting, $3.
cp $prefix.cbl $3

# the db2 CLP returns 2 for warnings; treat as if a 0 was
returned
if [ $returnCode -eq 2 ]; then
    returnCode=0
fi
if [ $returnCode -eq 0 ]; then
```

```

        db2 connect reset  >>$4 2>&1
        db2 terminate     >>$4 2>&1
    else
        echo \"db2 prep\" failed:  return code:  $returnCode >>$4
    2>&1
    fi
    exit $returnCode

```

3.9.2.2 Step 2: Creating a Configuration File

Using a text editor, create a configuration file named “boomerang.cfg” and include the configuration variables that can be specified for the **Boomerang** server. (Note: you can specify your own filename if desired). A sample file showing these variables and their default settings appears below:

```

# boomerang.cfg
# This file should be owned by root and only
writeable by root:
# chown root boomerang.cfg
# chmod 644 boomerang.cfg

# Default port is 7770
BOOMERANG_PORT 7770

# Default alias file is /etc/boomerang_alias.ini
BOOMERANG_ALIAS_FILE boomerang_alias.ini

#Default AcuAccess file is /etc/AcuAccess
ACCESS_FILE AcuAccess

```

3.9.2.3 Step 3: Creating an Access File

The server access file for **Boomerang** is named and structured the same as the server access file for **AcuServer** and **AcuConnect**. If you are using either of these servers for UNIX, you can use your existing Access file in conjunction with **Boomerang**, or you can set up a separate file for **Boomerang**. For instructions on setting up a server access file refer to either the **AcuConnect** or **AcuServer User’s Guide**, section 3.3.2 and 5.4.1 respectively.

3.9.2.4 Step 4: Starting the Server

To start the server, issue the following command:

```
boomerang -start -c boomerang.cfg -e boomerang.err
```

3.9.3 Server commands

The following table describes the **Boomerang** server commands.

-access	Used to create an access file. Refer to either the AcuConnect or AcuServer User's Guide, section 3.3.2 and 5.4.1 respectively.
-alias	Used to create an alias. See Section 3.9.1, step 3 for details on this command.
-c <configuration-filename>	Specifies the configuration file that should be used by Boomerang . If the "-c" option is not specified, Boomerang will use the file specified by the environment variable, A_BOOMERANGCFG. If neither the -c or A_BOOMERANGCFG are specified, the default "boomerang.cfg" file is used.
-e <error-filename>	Specifies the error file for the Boomerang server.
-f	Runs the Boomerang server in the foreground.
-kill [-n <port>]	Stops the Boomerang server. You can optionally specify the port.
-start [-n <port>]	Starts the Boomerang server. You can optionally specify the port.

-t #	<p>Turns on the tracing function. When combined with the “-e” option, trace information is placed in the named error file. The “#” represents the type of tracing or logging to be performed.</p> <p>“1” provides information about access file match attempts. The trace information buffer is flushed to the error file when the buffer is filled or Boomerang terminates.</p> <p>“2” provides information about client requests. The buffer is flushed to the error file when the buffer is filled or Boomerang terminates.</p> <p>“3” provides the information described for “1” and “2”.</p> <p>“5” is equivalent to “1”, but the tracing buffer is flushed to the error file each time an access file match is requested. (File trace flushing can also be controlled with the FILE_TRACE_FLUSH server configuration variable.)</p> <p>“6” is equivalent to “2”, but the tracing buffer is flushed to the error file each time a client connection is requested.</p> <p>“7” provides the information described for “5” and “6”.</p>
------	--

3.9.4 Client-side Operation – Remote Precompiling

The **Boomerang** client does not require any special setup or configuration. Once you have **set up and configured** your **Boomerang** server, use the **Boomerang** client to enter and carry out your remote preprocessing commands. **Boomerang** sends the specified source file to the server and invokes the preprocessor using the alias file that you created on the server. The preprocessed output file and status are then returned to the ACUCOBOL-GT compiler. If the precompile was successful, normal compiling occurs. If the precompile was not successful, the compiler will display the status.

Remote precompiling

Boomerang operates as either a standalone program, or as a preprocessor to the ACUCOBOL-GT compiler. To perform remote preprocessing, invoke **Boomerang** from the ACUCOBOL-GT compiler using the “-Pg” option. For example:

```
ccbl32 -Pg boomerang -server <myserver>[:<port>] -alias
<alias-name> source-filename
```

A list of all available commands appears below.

Note: **Boomerang** is also integrated with AcuBench. Refer to Section 9.5.3 of the AcuBench User’s Guide for information on precompiling with **Boomerang** from AcuBench. Refer to the table below for a description of all available client-side commands.

3.9.5 Client Commands

The following table describes the **Boomerang** client commands and arguments.

-alias <alias-name>	Tells Boomerang to pass this alias name to the server, which the server will then use to look up preprocessor-specific instructions.
-include <pattern>	Instructs Boomerang to copy INCLUDE files to the server and to use them in the preprocessing phase. Refer to section 3.9.6 for details on using this command.
-Pe <preprocessor-error-filename>	Writes preprocessor error messages to the specified filename.
-Po <output-filename>	Writes the preprocessed output to the specified filename.
-server < myserver>[:<port>]	Tells Boomerang which server to connect to, and if specified, which port.

-Sf <source-filename>	<p>Instructs Boomerang to copy the specified file to the server if needed by the preprocessor on the server. If no pathname is specified the file is expected to be in the current directory. If you are using AcuBench, the current directory is the directory above the Copylib and Source directories. This is a way to move files to the server that would not normally get moved by the “-include” or “-COPY” options. For example, if the preprocessor expands an EXEC statement into a COBOL COPY statement and the COPY file is on the client but not on the server, you can use this option to move the file to the server so it can be found by the preprocessor. These files are copied into the directory where the Boomerang server resides and are removed after preprocessing. Some preprocessors require that you add the Boomerang server directory to a server environment variable like COPYPATH so that it can locate these COPY files.</p>
-COPY	<p>Some preprocessors require COBOL COPY files to reside on the server. This option instructs Boomerang to copy the COPY files specified by COBOL COPY statements to the server and to use them in the preprocessing phase. If you execute Boomerang from the ACUCOBOL-GT compiler, use the compiler “-Sp” option to tell Boomerang where to find the COPY files on the client. If the “-Sp” option is not specified, Boomerang will look for the COPY files in the current directory. If you are using AcuBench, the current directory is the directory above the Copylib and Source directories. These files are copied into the directory where the Boomerang server resides and are removed after preprocessing. Some preprocessors require that you add the Boomerang server directory to a server environment variable like COPYPATH so that it can locate the COPY file.</p>

3.9.6 Working with INCLUDE files

With **Boomerang**, you can specify the “-include” option to tell **Boomerang** to send any preprocessor INCLUDE files that exist on the client to the server in case they are needed during the precompile. Do this by specifying the following command:

```
-include <pattern> <pattern...>
```

Since each preprocessor may have different syntax for specifying a preprocessor INCLUDE file, “pattern” is a sequence of case-insensitive strings that precede the name of the preprocessor INCLUDE file. The name of the INCLUDE file in the source file does not have to be enclosed in quotes, but if it is, it may be enclosed in single or double quotes.

For example, Pro*COBOL has the following syntax for a preprocessor INCLUDE file:

```
EXEC SQL  
  INCLUDE SQLCA  
END-EXEC.
```

You specify the following **Boomerang** option:

```
-include EXEC SQL INCLUDE
```

4

Terminal Manager

Key Topics

How the Terminal Manager Works	4-2
Getting Your Terminals Ready	4-5
The Keyboard Interface	4-9
The Display Interface	4-42
Restricted Attribute Handling	4-61
The Terminal Database File	4-65

4.1 How the Terminal Manager Works

Terminal Manager is the name we give to the Runtime System module that handles the input from the keyboard and the output to the screen. The Terminal Manager interprets the keys that the user presses, translating each keystroke into a function, such as a backspace. It also manages translation of attributes from your ACUCOBOL-GT application program to the screen.

The Terminal Manager provides a consistent interface between ACUCOBOL-GT programs and the particular machines on which they are running. The manager minimizes any differences among the various machines, operating systems, and terminals for which ACUCOBOL-GT is available.

The Terminal Manager also provides support for the emulation of graphical user interface components such as floating windows (modal *and* modeless) and controls on text-mode systems. For information regarding how to customize the characters used to emulate graphical components, see **section 4.6.7, “Graphical Window and Control Emulation.”**

This chapter describes how the Terminal Manager handles your program’s interaction with terminals, including both the screen display and the keyboard. This chapter also explains how you can configure the Terminal Manager and how it interacts with end users.

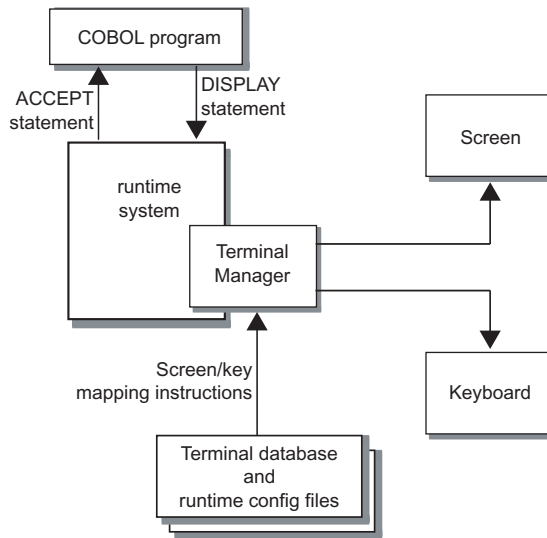
For example, in this chapter you’ll see how to specify what terminal you have, and how to make choices like:

- Designating a special action key
- Changing the on-screen prompt character
- Adding or changing colors
- Controlling data display and entry format
- Sounding error alarms

Sometimes features built into a COBOL program can override the effects of the values and variables described in this chapter. These situations can be important to application program developers and to end users, and are highlighted by notes at the appropriate places in this chapter.

4.1.1 Terminal Manager Functions

This diagram depicts how the Terminal Manager relates to hardware and other software in your system:



Many Terminal Manager functions depend on the data in two files:

- The **terminal database file**, a text file that maps screen and keyboard hardware signals of different terminals to common codes. The file contains signal-to-code sets for many popular terminals. In this chapter, the codes for screen and keyboard signals will generally be called *terminal function codes*. The term *key codes* will be used to refer to the subset of terminal function codes that deals with the keyboard.

Terminal function codes enable the Terminal Manager to handle I/O between application programs and a variety of terminals without any program changes; you only need to tell the Terminal Manager what terminal you will be using. Some of the codes can also be used to customize terminal actions, as described throughout this chapter.

- The **runtime configuration file**, a text file which includes variables that help define how the screen, the keyboard, and the user's keystrokes will be handled. Relevant runtime configuration variables are described in detail later in this chapter. They are often used in conjunction with the key codes and terminal function codes mentioned above.

4.1.2 Alternate Terminal Manager (ATM)

The Alternate Terminal Manager (ATM) runtime is a special 32-bit Windows runtime that allows you to use a 32-bit Windows server in much the same way that many UNIX servers are used. With the ATM runtime, you can telnet to the Windows server (with a third-party telnet service) to execute character-based ACUCOBOL-GT programs in the telnet window. However, the ATM runtime does not support program execution in the console window of the Windows server. The ATM runtime is licensed and installed separately from the standard ACUCOBOL-GT Windows (graphical) runtime.

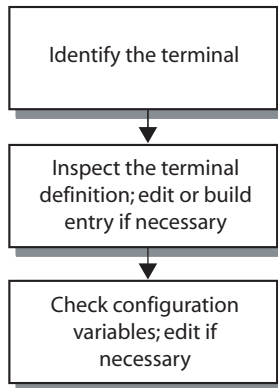
All major runtime functionality, except graphical support, is available with the ATM runtime. Because it is for character-based programs, certain Windows-specific features, such as pop-up dialog boxes, are not supported. Note that the ATM runtime automatically detects and uses Acu4GL DLLs, if present, and it supports calls to other DLLs.

Note: Should you need to relink the ATM runtime, see the instructions in section 6.3.6 of *A Guide to Interoperating with ACUCOBOL-GT*.

4.2 Getting Your Terminals Ready

This chapter describes your options as you prepare to use specific terminals with your application. Your computer's operating system and the ACUCOBOL-GT software will handle communication with most terminals without your doing anything. With some terminals, you will need or want to specify some choices.

Before running a program that uses the Terminal Manager, you may need to identify the type of terminal that will be used, and you may wish to customize the interface. The process of setting up a specific terminal involves these major steps:



4.2.1 Step One: Terminal Identification

The ACUCOBOL-GT runtime opens the terminal database file. Each entry in the file consists of the name of a terminal, followed by its screen and keyboard attributes, definitions and codes. (Runtimes for some systems, such as Windows, typically do not use a terminal database file. Check with your Micro Focus Customer Service representative if your terminal is non-standard, to determine if you require the terminal database file.)

The runtime first looks for the system variable `A_TERM`; if that variable is present, the runtime opens the file named in it as the terminal database file. If the system variable `A_TERM` is *not* present, the runtime opens the file name shown in the table below. The file name varies with the operating system (note that file names on UNIX systems are case-sensitive):

System	Terminal database file
UNIX	/etc/a_termcap
MPE/iX	/etc/a_termcap
VMS	SYSS\$LIBRARY:A_TERMS.DAT

You must tell the Terminal Manager what terminal database file to use with your ACUCOBOL-GT application. Either:

a) use the path and name specified in the table above, and do not set `A_TERM` (this approach works fine in most cases),

or

b) use a path and name of your choosing, and specify that path and name in `A_TERM`.

After the terminal database file is opened, the Terminal Manager needs to know what terminal type is to be used, and where to locate the entry that describes it. One of the system variables `A_TERM` or `TERM` holds the name of the entry that is to be used.

The Terminal Manager looks first for the variable `A_TERM`. If it is present, the Terminal Manager searches the terminal database file for the terminal named in `A_TERM`. If `A_TERM` is *not* present, the Terminal Manager looks for the variable `TERM` and then searches the terminal database file for the terminal named in `TERM`. Setting `TERM` to the correct terminal name will handle most situations; see [section 4.2.2](#) in this chapter for exceptions. If neither `TERM` nor `A_TERM` is present, the Terminal Manager terminates the runtime with an error message.

The various operating systems handle `TERM` and `A_TERM` in different ways:

- On VMS systems, TERM and A_TERM are *symbols*.
- On UNIX systems, they are *environment variables*; most UNIX systems set the TERM variable at login time.
- The Windows console (character-mode) runtime does not use the terminal database file, and so does not need to know the value of TERM.
- Graphical runtimes do not use the terminal database file, and so do not need to know the value of TERM.
- The ATM runtime uses TERM and A_TERM as environment variables, just like UNIX.

The terminal database file shipped with the ACUCOBOL-GT runtime contains definitions of the characteristics of most popular terminals; you will probably find yours listed. If the entry named in A_TERM or TERM describes the terminal you will use with your ACUCOBOL-GT application, then nothing more need be done.

4.2.2 Step Two: Terminal Definition

If the terminal database file entry named in A_TERM or TERM does *not* describe the terminal you will use, you probably will not want to change the value of TERM, because other software may rely on that value. Instead, take these two steps:

1. Locate the terminal database file entry that correctly describes your terminal, or create a new one and give it a new name (see **Section 4.6, “The Terminal Database File,”** for details).
2. Set A_TERM to the name of that entry.

4.2.2.1 Windows special considerations

Neither the ACUCOBOL-GT graphical or console runtimes for Windows use a terminal database file when a standard Windows monitor is used. If you choose to use a character-based terminal (such as the VT-100), you will need both the *alternate terminal manager* runtime and a terminal database file. These can be requested from your Customer Service Representative at Micro

Focus. Be aware that your programs will execute less efficiently with this combination than with the standard Windows runtime and a standard Windows monitor.

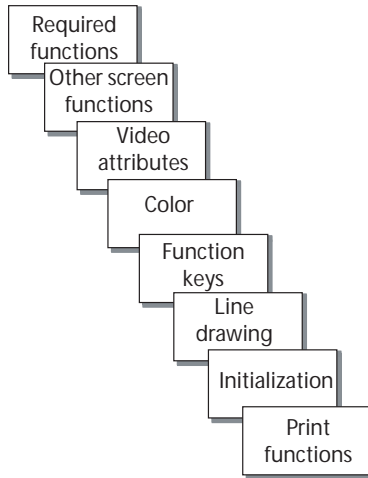
4.2.3 Step Three: Configuration Variables

Some behaviors of the terminal can be controlled by entries (variables) in the runtime configuration file. The default name of this file, like that of the terminal database file, varies according to the host operating system:

System	Runtime Configuration File
Windows	<code>\etc\cblconfi</code>
UNIX and Linux	<code>/etc/cblconfig</code>
MPE/iX	<code>/etc/cblconfig</code>
VMS	<code>SYSS\$LIBRARY:A_CONFIG.DAT</code>

Options on naming and accessing the runtime configuration file, and descriptions of many of the entries in it, are discussed in the “Compiler and Runtime” chapter of this manual. That chapter also discusses the relationship between the runtime configuration file and the host computer’s environment. This chapter discusses entries in the terminal database file and the runtime configuration file which are of particular importance to the Terminal Manager.

Entries in both the terminal database file and the runtime configuration file are described throughout this chapter, grouped according to the functions that they control. These are the basic areas of functionality that you will need to consider in deciding what you need to modify or define:



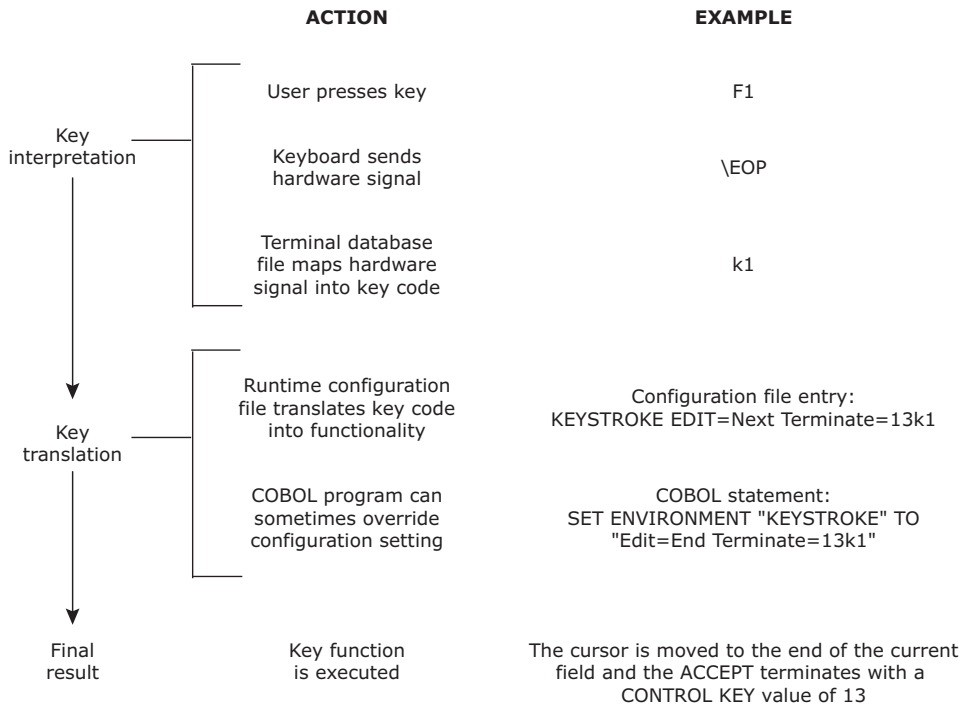
4.3 The Keyboard Interface

The Terminal Manager handles both the screen display and keyboard input. See [section 4.4](#) for more information about the display interface.

This section addresses keyboard-related functions. The Terminal Manager provides certain conventions for entering and editing data; these conventions are described here.

4.3.1 Key Mapping

The mapping of keys to functions is one of the main activities of the Terminal Manager. If you understand what happens when a user presses a key, you'll have a good feel for how you can control the interaction between the keyboard and the COBOL application. The next diagram depicts the overall process, from keystroke to COBOL program.



The following section describes in detail the steps shown above.

4.3.1.1 Key interpretation

When the user presses a key, the keyboard sends a signal to the computer. This signal needs to be interpreted and translated into a value, or functionality, that the COBOL program can understand. This process begins with the terminal database file.

The terminal database file equates hardware signals to logical values. Unless you are running on a system such as Windows, where the key interpretation is built into the runtime, your terminal needs to be listed in the terminal database file. Keystrokes at the terminal generate hardware signals, and each hardware signal must be equated to a logical value if the application program is to respond to the associated keystroke. These logical values, called *key codes*, are listed in the Table of Keys in **section 4.3.2.3**.

We provide the definitions for many popular terminals in the terminal database file that we send you. If the name of your terminal is included, you will not need to change anything unless your particular terminal is different from the standard configuration for its type. If that is the case, you may need to change the entry.

Each entry in the terminal database file consists of the name of the terminal (including all names by which you might typically refer to this terminal type), followed by a series of character strings. Some of these strings are equations that assign hardware signals from the keyboard to key codes that we provide. Some of the strings consist of functional instructions to the terminal.

Terminal database file entries and the syntax rules that govern them are described in **section 4.6, “The Terminal Database File.”**

4.3.1.2 Key translation

After the hardware signal has been equated to a key code, the runtime system checks the runtime configuration file to determine if any special values or functions have been attached to the key code.

This is the point at which statements in a COBOL program can override what is specified in the runtime configuration file.

4.3.1.3 Keyboard configuration

Each terminal has several keys that are available to be used for special purposes. Some of these keys are used as field termination keys, others are used as editing keys. ACUCOBOL-GT supports a large number of special keys, but in the default configuration, only these are used:

Function Keys 1-20	Help
Arrow Keys	Do
Page Up	Page Down
Backspace	Line-Kill
Home	End
Insert	Delete
Clear	Clear to End
Carriage Return	Control Keys

The *Backspace* and *Line-Kill* keys are whichever keys provide these functions for your operating system. The Backspace key is the one that erases individual characters from a command line; the Line-Kill key is the one that can cancel a command line. On most systems, the key that performs the Backspace function is the one labeled either “backspace” or “delete.”

The keyboard interface can be easily configured to meet a variety of needs. The default configuration has the following characteristics:

1. The range of legal input characters is ASCII values 32 through 255. Other characters outside this range are ignored unless covered by one of the cases below.
2. The range of exception characters is ASCII values 1 through 31. If any of these characters is typed, an exception condition exists and input to the field is terminated. The exception key value is identical to the ASCII value of the key. For example, if Control-E is typed, then the exception key value returned would be “5”. This rule does not apply to characters specifically listed in rule 3.
3. The following table outlines the actions of other keys. In this table, **Action** is the special action performed by the key. If a number is present here, then this key terminates the field and returns that number

as its termination key value. If the number is starred (*), then this key also causes an exception condition. If there are both a number and an action, then the key acts as a termination key when the action cannot be applied.

The **Windows** column names the keycap on the IBM-PC keyboard that is used for this key. The **Termcap** column names the terminal database file entry that corresponds to this key for UNIX and VMS systems.

Key	Action	Windows	Termcap
Carriage Return	13	Enter	
Tab	Next Field (9)	Tab	
Host's Backspace	Backspace	BkSp	
Host's Line-Kill	Erase Field		
Backtab	Previous Field	Shft-Tab	kB
Home	First Field	Home	kh
End	Last Field	End	KE
Insert	Auto-Insert Mode	Ins	KI
Delete	Delete Character	Del	KX
Clear	Erase Field	Ctl-Home	KC
Clear-to-End	Erase Remainder	Ctl-End	kE
Left Arrow	Left	Left	kl
Right Arrow	Right	Right	kr
Up Arrow	Previous-All (52*)	Up	ku
Down Arrow	Next-All (53*)	Down	kd
Page Up	Page-Up (67*)	PgUp	kP
Page Down	Page-Down (68*)	PgDn	kN
Do (Command)	40*		KD
Help	90*		K?
F1 - F10	1 - 10*	F1 - F10	k1 - k0
F11 - F20	11 - 20*	Shft F1 - F10	K1 - K0

Note: Keys that terminate input with an exception condition are ignored by ACUCOBOL-GT if the ACCEPT statement does not have an EXCEPTION clause or a CONTROL KEY clause. However, if you use the “-Vx” compile-time option, exception keys *will* be recognized even if the ACCEPT statement does *not* contain an EXCEPTION or CONTROL KEY clause.

When accepting data from the keyboard, the Terminal Manager runs in one of two modes: “standard” mode or “auto” mode. In “standard” mode, the only way to finish input is by typing one of the allowed termination keys; the cursor may not leave the field. In “auto” mode, the cursor can leave the field; when the user fills the field with data, it is immediately accepted and the cursor moves on. The setting of “auto” mode or “standard” mode is determined by the various clauses specified on the ACCEPT statement. For details, see Chapter 6 of the *Reference Manual*, ACCEPT verb.

There are four methods for accepting a field (ACCEPT verb, Format 1), depending on the mode and the presence of either the CONTROL KEY clause or the ON EXCEPTION clause. These methods are:

Standard mode, no CONTROL KEY or ON EXCEPTION clause:

the field can be accepted only by a termination key. In the default keyboard configuration, these are the Carriage Return and Tab keys.

Standard mode, with CONTROL KEY or ON EXCEPTION clause:

the field can be accepted by a termination key or by one of the exception keys.

Auto mode, no CONTROL KEY or ON EXCEPTION clause:

the field can be accepted by a termination key or by filling the field with data.

Auto mode, CONTROL KEY or ON EXCEPTION clause:

the field can be accepted by filling it with data, or by a termination key or an exception key.

The Terminal Manager can control more than one field when the program is doing an ACCEPT that refers to a Screen Section item (ACCEPT verb, Format 2). In the course of this ACCEPT, the user can move between the

fields by using the Tab, Backtab, Left, Right, Up, Down, Home and End keys; the Tab key acts as a terminate key only in the last field. A Format 2 ACCEPT statement does not support the use of the CONTROL KEY clause; the CRT STATUS phrase of the Special-Names paragraph may be substituted. Data entry for a Screen otherwise falls into four categories much like the above.

The termination and exception keys may be changed by runtime configuration options as described in **section 4.3.2, “Redefining the Keyboard.”**

Note to RM/COBOL-85 users:

The ACUCOBOL-GT default keyboard layout is very similar to that used by RM/COBOL-85, but it is not identical. Consider these points:

1. Under MS-DOS, RM/COBOL-85 defines the Command key to be Alt-C. When you are typing this key it is easy to accidentally type Control-C instead, which is the interrupt key in DOS. For this reason, ACUCOBOL-GT uses Alt-D (for “Do”) instead.
2. ACUCOBOL-GT defines more editing keys than RM/COBOL-85 does. In particular, the Home, End, Clear, Clear-to-End and Line-Kill keys return exception values under RM/COBOL-85, while under ACUCOBOL-GT they perform various field-editing functions.
3. ACUCOBOL-GT defines Page Up, Page Down, and Help keys that are not defined under RM/COBOL-85. These keys are used by the ACUCOBOL-GT debugger.
4. The default RM/COBOL-85 keyboard includes the following keys as exception keys: Attention, Home, New Line, Tab Left, Erase Right, Tab Right, Insert Line, Delete Line, and Send. Under ACUCOBOL-GT, these keys either act as editing keys or are ignored. Because these keys are generally not available on most keyboards (or, in the case of the Tab Right and New Line keys, are ambiguous with control keys), most applications do not use them. If you need to use any of these keys, you can alter the ACUCOBOL-GT keyboard configuration as described in **section 4.3.2, “Redefining the Keyboard.”**

5. The RM/COBOL-85 layout varies from machine to machine. In the interest of portability, the default ACUCOBOL-GT keyboard interface is the same for all machines.

4.3.2 Redefining the Keyboard

The ACUCOBOL-GT keyboard interface may be modified via two variables in the runtime configuration file. The `KEYBOARD` variable defines global keyboard attributes. The `KEYSTROKE` variable defines the interpretation of a particular key or key combination. These two variables enable you to tailor the keyboard interface to your application.

4.3.2.1 The `KEYBOARD` variable

You can specify one or more `KEYBOARD` variables. Attributes that you can set are identified by one or more sets of *keywords* and associated *values*, separated from each other by spaces or tabs. The syntax is:

```
KEYBOARD keyword=value [keyword=value]...
```

Keywords are:

AUTO-RETURN=value

Some `ACCEPT` statements terminate automatically when the input field is filled. When this occurs, the termination key value is the *value* (a decimal number) defined by the `AUTO-RETURN` keyword. This value is returned in the `CONTROL KEY` clause of the `ACCEPT` statement. The default value is zero. You may also specify this option with the configuration variable `KBD_AUTO_RETURN`.

CASE=value

The `CASE` option on the `KEYBOARD` configuration entry allows you to cause all entries to be automatically converted to upper or lowercase. *Value* may be set to “Upper”, “Lower” or “Both”:

Upper: all `ACCEPT` statements convert keystrokes to upper case.

Lower: all `ACCEPT` statements convert keystrokes to lowercase.

Both: (default) causes no translation.

CASE may be overridden with the settings of the “UPPER” and “LOWER” keywords on individual ACCEPT statements. See Book 3, *Reference Guide*, section 6.4.9, “Common Screen Options.”

The configuration variable KBD_CASE is also supported.

Note: The value of KEYBOARD CASE does not affect controls. To effect the same result on controls, use the UPPER or LOWER phrase on the ACCEPT statement.

CHECK-NUMBERS=*value*

Normally, ACUCOBOL-GT requires that numeric data be entered for numeric and numeric-edited fields that have the CONVERT phrase specified for them. If *value* is “No”, then any data can be entered and the runtime system will remove the non-numeric data from the user’s input prior to converting. If *value* is set to “Yes” (the default), then a non-numeric entry will cause an error message to print and will force the user to re-enter the field. If *value* is set to “Validate”, the runtime also checks to make sure that the numbers entered are valid, as described by the PICTURE clause for that field. The configuration variable KBD_CHECK_NUMBERS can also be used to set this value.

CURSOR-PAST-END=*value*

By default, ACUCOBOL-GT does not let the cursor leave the field where data is being entered. When the final position is entered, the cursor remains there and further entry is inhibited except for editing keys. Setting *value* to “Yes” allows the cursor to move one character past the end of the field instead. Input is still inhibited. The difference between the two modes is essentially cosmetic and CURSOR-PAST-END can be set to suit the user’s taste. The default *value* is “No”. This value may also be specified with KBD_CURSOR_PAST_END.

DATA-RANGE=*value*

Value defines the range of legal ASCII input values. Any character received that falls outside of this range will not be accepted into the input field, but may define other actions such as field editing or input termination. Two decimal numbers, separated by a comma, express the lower and upper bounds of the range. The maximum range is “1,255”.

The default range is “32,255”. This value may also be set using the configuration variables KBD_DATA_RANGE_HIGH and KBD_DATA_RANGE_LOW.

Note: If the same number(s) is included in both the DATA-RANGE and EXCEPTION-RANGE, then DATA-RANGE takes precedence.

EXCEPTION-RANGE=*value*

This is similar to the DATA-RANGE keyword except that *value* defines the range of characters that generate default exception handling. A key whose ASCII value falls within that range will terminate input with an exception condition value which matches the ASCII value of the key. That value is returned in the EXCEPTION clause or the CONTROL KEY clause of the ACCEPT statement. The default range is “1,31”. A character in this range that is also defined by a KEYSTROKE variable acts as defined by that KEYSTROKE variable, and may or may not terminate the input. The configuration variables KBD_EXCEPTION_RANGE_HIGH and KBD_EXCEPTION_RANGE_LOW may also be used to set this value.

Note: If the same number(s) is included in both the DATA-RANGE and EXCEPTION-RANGE, then DATA-RANGE takes precedence.

IMPLIED-DECIMAL=*value*

If *value* is “Yes”, an implied decimal point is inserted in certain fields when the user does not explicitly type a decimal point. The last *n* digits of the user’s input will be to the right of the decimal point, where *n* is the number of decimal places specified in the receiving field. For example, if the program is accepting a field with two decimal places, and the user types “1535”, the value accepted (and echoed to the screen) will be “15.35”. This is only done for numeric or numeric-edited fields that are input with conversion, either explicit or implicit. It never occurs for floating-point items. The default *value* is “No”. The configuration variable KBD IMPLIED_DECIMAL is also supported.

RM-2-DEFAULT-HANDLING=value

RM/COBOL versions 2.1 and 2.2 have a configuration option that allows for ACCEPT fields that do not receive any input (e.g., the user just types Return) to leave the receiving field unchanged. Normally, the receiving field would be filled with spaces. If the RM-2-DEFAULT-HANDLING *value* is “Yes”, then ACUCOBOL-GT will behave in this alternate fashion. You may also set this value using the variable KBD_RM_2_DEFAULT_HANDLING.

Note: This option is recommended only if you are converting programs written using this feature of RM/COBOL. Note that RM/COBOL-85 does not contain this feature, so only programs written for RM/COBOL version 2 should need to use it.

SCREEN-DEFAULT=value

If *value* is “Yes”, default data is taken from the screen for any ACCEPT statement that does not have a default value specified for it (either explicitly or implicitly). This will also allow for updating of the current screen contents. The default *value* is “No”. This option can also be specified using the configuration variable KBD_SCREEN_DEFAULT.

4.3.2.2 The KEYSTROKE variable

The KEYSTROKE variable defines the actions to be taken for a single keystroke. You need to add one KEYSTROKE line for each key that you wish to redefine. The maximum number of allowed KEYSTROKE entries is 170.

KEYSTROKE entries consist of *keywords* and associated *values* that describe the action to be taken, plus the *key code* (a two-character name) of the key, or key combination, being defined. All definable keys have such a name. The key code is case sensitive, although the rest of the KEYSTROKE line is not. The ASCII value of the key (decimal) may be used instead of the key code. Note that this is the *only* way to assign a value to the DEL key (ASCII value 127). The syntax of the KEYSTROKE line is:

```
KEYSTROKE keyword=value [keyword=value] key-code
```

The *key-code* argument is one of the two-character codes shown in **section 4.3.2.3, “Table of keys.”** Keywords are separated from each other by spaces or tabs. For example:

```
KEYSTROKE      EDIT=Next      TERMINATE=13      ^M
```

The following keywords may be used:

AT-END=value

If *value* is “Yes”, the key becomes a termination key that also causes the AT END condition. This keyword may not be specified along with either the TERMINATE or EXCEPTION keywords. The AT END condition always returns a termination value of “-1” in the CONTROL KEY clause. AT-END keystrokes are always allowed, but will cause no action unless an AT END, EXCEPTION, or CONTROL KEY clause is present in the ACCEPT verb. The default keyboard defines no AT-END keys. See also the AT END phrase of the ACCEPT verb in the ACUCOBOL-GT *Reference Manual*, Procedure Division.

DATA=value

This keyword is used to assign special characters to keys. DATA associates a decimal ASCII *value* with a key; the Terminal Manager will return this value to the COBOL program when the key is pressed. If the DATA keyword is used, no other keywords may be specified for this key.

EDIT=value

EDIT is used to define an editing action for a key. It must be assigned one of the following *values*:

Alt	Left
Auto-Insert	Menu
Backspace	Next
Default-Entry	Next-All
Default-Next	Next-Line
Delete	Numeric-Default
Down	Numeric-Next
End	Page-Down

Erase-All	Page-Up
Erase-EOS	Previous
Erase Field	Previous-All
Erase-Next	Previous-Line
Erase-to-End	Right
First	Switch-Window
Home	System-Menu
Insert-Off	Toggle-Edit-Mode
Insert-On	Toggle-Insert
Insert-Space	Up
Last	

The EDIT keyword values specify various editing functions, described below. EDIT keys may also be designated as termination keys. When they are, the EDIT is applied and then the input is terminated. This rule is slightly changed for the actions that move the cursor. With these actions, the field terminates only if the cursor cannot be moved farther in the requested direction. This is detailed in the descriptions of each of the EDIT values.

In the following descriptions, the order of fields is the order in which they appear in the Screen Section. Thus, the “next” field may not necessarily be the next one on the physical display. This feature can be used to design special purpose screens.

EDIT keyword values can be:

Alt	<p>When edit=alt is defined, the user must press the Alt key and the key letter in order to activate the desired control.</p> <p>The character-based version of the runtime supports the use of key letters for operating controls. By default, if the program has key letters defined for its controls and if the program is currently accepting a control that doesn't allow user input, such as a push-button, checkbox or radio button, the user can move to the control by simply pressing its key letter key. When this happens the accept will be terminated. This default behavior can be changed by setting the runtime configuration variable NO_BARE_KEY_LETTERS to "TRUE".</p>
Auto-Insert	<p>Auto-Insert causes all following characters to be entered in insert mode. Auto insert mode is automatically reset when the input terminates, or when any other editing key is typed. This style of insertion is the RM/COBOL-85 default method.</p>
Backspace	<p>The Backspace function moves the cursor to the left one character and deletes the character found there. If the Backspace function occurs at the left-most field position, it is ignored unless a TERMINATE or EXCEPTION value has been assigned to the key, in which case it is treated as a termination key.</p>
Default-Entry	<p>The Default-Entry action erases the remainder of the field starting at the cursor position <i>provided that</i> the cursor is not in the first position of the field. If the cursor <i>is</i> in the first position, this action does nothing. This editing action is intended to be tied to a termination key (such as the "Return" key or the "Tab" key), and to be used as a reasonable method of handling fields that contain default values. If the default is correct, then this key is typed (which does nothing to the field). If the default is wrong, then the correct value is entered and this key is typed (erasing the part of the old field after the new input).</p>
Default-Next	<p>Default-Next combines the Default-Entry action and the Next (described below) action.</p>

Delete	Deletes the character that the cursor is on (if any).
Down	If there are fields below the current cursor location, the cursor moves to the one on the closest lower line. If there is more than one field on this line, the cursor will move to the one closest to its current horizontal location. The cursor will try to stay in the same column. If there are no fields beneath the current line, then this action does nothing unless an EXCEPTION or TERMINATE value has been assigned to it, in which case it acts as a termination key.
End	The cursor is moved to the end of the current field, excluding any trailing prompt characters. If the cursor is already at the end of the field, then this key is ignored unless it has a TERMINATE or EXCEPTION value, in which case it is treated as a termination key.
Erase-All	<p>All fields controlled by the ACCEPT statement are erased and the cursor is moved to the home position of the first field. This key may not be assigned a TERMINATE or EXCEPTION value.</p> <p>Under Windows, the field in which the cursor is currently positioned is erased, instead of all fields controlled by the ACCEPT statement being erased.</p>
Erase-EOS	<p>The current field is erased from the cursor location to the end of the field, and all fields following the current one are erased. The definition of “following field” is based on the order of fields in the Screen Section. This action may not be assigned a TERMINATE or EXCEPTION value.</p> <p>Under Windows, the current field is erased from the cursor location to the end of the field, but the fields following the current one are not erased.</p>
Erase-Field	The field is erased, and the cursor is moved to the first position of the field.
Erase-Next	This action combines the functions of the Erase-to-End action and the Next (described below) action.
Erase-to-End	This function erases the field from the current cursor position to the end of the field.

First	The cursor is moved to the beginning of the first field controlled by the ACCEPT statement. If the cursor is already in the first field, and the key has been assigned a TERMINATE or EXCEPTION value, then the ACCEPT terminates.
Home	The cursor is moved to the beginning of the field. If the cursor is already at the beginning and this key has been assigned a TERMINATE or EXCEPTION value, the ACCEPT terminates.
Insert-Off	If insertion mode is currently in effect, it is turned off; otherwise does nothing.
Insert-On	This causes all following characters to be entered in insert mode. This causes any trailing characters to be moved one space to the right before the added character is printed. Insertion mode stays in effect until explicitly reset by an Insert-Off, an Auto-Insert, or a Toggle-Insert action. Note that insertion mode stays in effect across multiple ACCEPT statements.
Insert-Space	A space character is inserted at the cursor position, moving trailing characters over one position.
Last	<p>The cursor moves to the end of the last field controlled by the ACCEPT statement. Trailing prompt characters in the last field are ignored in determining the end of the field. If this key has been assigned a TERMINATE or EXCEPTION value and the cursor is already in the last field, the ACCEPT terminates.</p> <p>Under Windows, with TERMINATE and EXCEPTION value, if the cursor is at the last field, the ACCEPT does not terminate and the cursor stays at the current field.</p>

Left	<p>The cursor is moved one position to the left; if it is already in the left-most field position, it moves to the end of the previous field. If the cursor is in the left-most position of the first field, the key is ignored unless it also has been assigned a TERMINATE or EXCEPTION value, in which case the ACCEPT terminates.</p> <p>In the case of Windows, the left and right arrow keys move the cursor inside a field but do not act as terminators. Without TERMINATE and EXCEPTION value, if the cursor is in the leftmost position of the first field, the key is not ignored and the cursor is moved to the last field.</p>
Menu	<p>The key is defined as a Menu key. Pressing this key will cause a program-defined menu to appear on the screen.</p>
Next	<p>Under Windows, without TERMINATE and EXCEPTION value, if the cursor is in the last field, it moves to the first field, instead of moving to the end of the field.</p>
Next-All	<p>The cursor moves to the beginning of the next field regardless of whether or not the next field has a Tab-Stop. Thus a key with the “Next” action will skip controls with the NO-TAB style, while a key with the “Next-All” action will not.</p> <p>By default, the Down key is assigned the Next-All action. This makes the Down key behave more like it does in a common Windows program. Assign the Down keyword (described above) for a more traditional, text-mode behavior.</p>
Next-Line	<p>Next-Line functions the same as the Down action, except that the cursor always moves to the beginning of the left-most field on the new line (instead of maintaining the current cursor column).</p>

Numeric-Default	<p>If the field is numeric, then this key acts just like the Default-Entry key. Typing this key at the first character position of a numeric field leaves the field unchanged and accepts the default value. Typing this key when the cursor is <i>not</i> in the first position causes erasure of the field from the cursor position to the end. This key allows the user either to accept the default or type over it without having to worry about blanking out the trailing portion of the field.</p> <p>If the field is alphanumeric, then this action does not affect the field. Numeric-Default is usually made a termination key, so that typing it causes the ACCEPT to finish.</p>
Numeric-Next	<p>If the field is numeric, then this key acts just like the Default-Next key. Typing this key at the first character position of a numeric field leaves the field unchanged, accepts the default value, and advances to the beginning of the next field. Typing this key when the cursor is <i>not</i> in the first position causes erasure of the field from the cursor position to the end. The cursor is then advanced to the beginning of the next field. This key allows the user either to accept the default or type over it without having to worry about blanking out the trailing portion of the field.</p> <p>If the field is alphanumeric, then this key acts just like the Next key. It advances the cursor to the beginning of the next field and does not affect the current field.</p>
Page-Down	<p>This keyword sets the key that pages down a multiline entry field, list box, and combo box.</p>
Page-Left	<p>This keyword sets the key that scrolls left one page.</p>
Page-Right	<p>This keyword sets the key that scrolls right one page.</p>
Page-Up	<p>This keyword sets the key that pages up a multiline entry field, list box, and combo box.</p>
Previous	<p>The cursor moves to the beginning of the previous field. If the cursor is in the first field, it moves to the beginning of the field unless the key has been assigned a TERMINATE or EXCEPTION value, in which case it acts as a termination key instead.</p>

Previous-All	<p>The cursor moves to the beginning of the previous field regardless of whether or not the previous field has a Tab-Stop. Thus a key with the “Previous” action will skip controls with the NO-TAB style, while a key with the “Previous-All” action will not.</p> <p>By default, the Up key is assigned the Previous-All action. This makes the Up key behave more like it does in a common Windows program. Assign the Up keyword (described below) for a more traditional, text-mode behavior.</p> <p>Under Windows, without TERMINATE and EXCEPTION value, if the cursor is in the first field, it moves to the last field, instead of moving to the beginning of the field.</p>
Previous-Line	<p>The cursor moves to the beginning of the left-most field on the next higher line. If there are no fields above the current one, this action does nothing unless it has an EXCEPTION or TERMINATE value, in which case it acts as a termination key.</p>
Right	<p>This function moves the cursor one position to the right. This will not move the cursor onto any trailing prompt characters (exception: if the prompt character is a space and the field is being updated, the cursor will move over the trailing spaces).</p> <p>If the cursor is as far right as it is allowed to go, it will move to the beginning of the next field. If there is no following field, this key is ignored unless a TERMINATE or EXCEPTION value has been assigned, in which case the ACCEPT terminates.</p> <p>In the case of Windows, the left and right arrow keys move the cursor inside a field but do not act as terminators. Without TERMINATE and EXCEPTION value, if the cursor is as far right as it is allowed to go, the key is not ignored and moves to the first field. (See Book 2, <i>User Interface Programming</i>, section 11.4, “Regarding Configuration Variables”)</p>
Scroll-Left	<p>This keyword sets the key that scrolls left one column.</p>
Scroll-Right	<p>This keyword sets the key that scrolls right one column.</p>

Switch-Window	This keyword defines the key that, when pressed, causes the system to enter “switch window mode.” In this mode, the user can press any key to cycle through the modeless windows, with each window border highlighted until the “Return” key is pressed. Window switching order is from top to bottom.
System-Menu	Use the System-Menu function to define the key used to activate a floating window’s <i>system</i> menu on a text-mode system.
Toggle-Edit-Mode	This keyword defines the key that can be used to toggle the presence of the combo box’s drop-down list and the paged list box’s search box.
Toggle-Insert	If insertion mode is currently in effect, it is turned off. Otherwise, insertion mode is turned on.
Up	If there are fields above the current cursor location, the cursor moves to the one on the closest higher line. If there is more than one field on this line, the cursor moves to the field closest to its current location. The cursor will try to stay in its current column. If there are no lines above the current line with active fields, then this key is ignored unless it has a TERMINATE or EXCEPTION value, in which case it acts as a termination key.

EXCEPTION=*value*

The purpose of this keyword is to create an exception key. EXCEPTION assigns a decimal ASCII *value* to a key; the key becomes a termination key that also causes an exception condition. The assigned value is returned in the EXCEPTION clause or the CONTROL KEY clause of the ACCEPT statement. See the TERMINATE keyword below if you want to terminate input *without* causing an exception condition. Note that ACUCOBOL-GT inhibits exception keys when no EXCEPTION or CONTROL KEY clause is present in the ACCEPT statement, unless the program was compiled with the “-Vx” option.

HOT-KEY=*value*

ACUCOBOL-GT offers two methods for assigning hot keys: the KEYSTROKE keyword HOT-KEY described here, and the HOT-KEY runtime configuration variable described in Appendix H. Either or

both may be used, but the results are undefined if you assign the same key using both formats. The total number of hot-key entries defined by both methods cannot exceed 16.

A *hot key* is a key that is associated with a program, so that when the key is pressed, the corresponding program is run. *Value* is the program name, which must be specified in single or double quotes if it is lowercase. The full configuration file entry looks like this:

```
KEYSTROKE HOT-KEY=program-name key-code
```

Pressing the key specified in *key-code* initiates execution of the program just as if it were named in a CALL statement. The *key-code* argument is one of the two-character key codes shown in **section 4.3.2.3, “Table of keys.”**

For example, there is a screen printing sample program named PRNTSCRN provided with ACUCOBOL-GT. If you want to be able to initiate that program just by pressing the keyboard’s “F11” key, add the following line to your configuration file:

```
KEYSTROKE HOT-KEY=PRNTSCRN U1
```

Hot keys are active only during Format 1 and Format 2 ACCEPT statements (these are the forms of the ACCEPT verb that allow the user to enter data at the keyboard). When the user presses a hot key, the current program status is saved, and the program associated with the hot key is run. When the hot-key program exits (via the EXIT PROGRAM statement), control is returned to the program that was running when the hot key was pressed. The hot-key function does not save the original contents of the screen. You can accomplish this by popping up a window in your hot-key program, and then closing the window just before you exit the hot-key program.

A hot-key program is automatically passed two parameters. The first parameter is PIC X(200). It contains an image of the data in the field that was being entered at the time the hot key was pressed. The second parameter is a COMP-1 field that contains the length of the field being entered. You can define the first parameter as a table that depends on the second parameter like this:

```
LINKAGE SECTION.
01 CURRENT-FIELD.
   03 OCCURS 1 TO 200 TIMES
      DEPENDING ON FIELD-SIZE    PIC X.
```

```
01 FIELD-SIZE    PIC S9(4) COMP-1.
```

You are not required to declare or use either of these parameters in your hot-key program--they are provided for convenience.

The hot-key program may modify its first parameter. Any modifications made are reflected in the field that was being entered when the hot-key program was called. You might use this capability to perform a look-up function and then return the value found to the field being entered. If you want to pass additional data to the hot-key program, use EXTERNAL DATA ITEMS.

When a hot-key program is started, the value of the RETURN-CODE special register is saved and then set to zero. The hot-key program may alter this value. When the hot-key program exits, the value of RETURN-CODE is checked. The following table shows the possible values and the action that the calling program will take:

Value	Action
0	continue ACCEPT
>0	generate exception if allowed
-1	activate "next field" logic

If the value of RETURN-CODE is zero, then the calling program continues to a normal completion of the ACCEPT statement that was active when the hot key was pressed.

If the value is greater than zero, then the calling program acts as if an exception key (with that value) was pressed. This will terminate the ACCEPT statement if it is of a format that allows exception keys.

If the value is "-1", then the ACCEPT statement will act as if a "next field" key were pressed by the user. This will cause the ACCEPT statement to proceed to the next field. If there are no more fields (or if there is only one field), then the ACCEPT statement will terminate with a termination value of zero. *The hot-key program should not set RETURN-CODE to any negative value other than "-1". Other negative values are reserved for future use by ACUCOBOL-GT.*

In any case, after the RETURN-CODE value established by the hot-key program has been acted upon by the calling program, RETURN-CODE is restored to the value it held before the hot-key program was called.

If a hot-key program cannot be executed, an error message is displayed to the user, and control returns to the ACCEPT statement.

Up to two hot-key programs per process may be active at once.

INVALID=*value*

If *value* is “Yes”, the key is ignored when it is typed. This keyword may not be specified with any other keywords.

TERMINATE=*value*

This keyword is used to create a termination key. TERMINATE assigns a decimal ASCII *value* to a key; when the key is pressed, the ACCEPT is terminated and the assigned value is returned in the CONTROL KEY clause of the ACCEPT statement. TERMINATE does *not* cause the key to generate an exception condition when pressed; to define an exception key, use the EXCEPTION keyword instead.

4.3.2.3 Table of keys

The following tables list all of the keys that can be redefined. The tables list the key’s full name, its two-character name (called the “key code”), and the corresponding key used on a Windows keyboard (not all keys listed can be redefined under Windows; see Note below). The key code is used in the terminal database file on UNIX and VMS systems to identify the corresponding key-sequence.

Note: On Windows systems, the alt key sequences, PrtScrn, and F10 keys are directly handled by Windows and cannot be referenced in COBOL. A list of additional keys that can be redefined in Windows environments follows the table below.

Key	Key Code (terminal db file)	Windows Keyboard
Host’s Backspace	ZB	BkSp

Key	Key Code (terminal db file)	Windows Keyboard
Host's Line-Kill	ZK	-
Cntrl-A - Cntrl-Z	^A - ^Z	Ctl A-Z
Escape	^[Esc
Control-\	^\	Ctl-\
Control-]	^]	Ctl-]
Control-^	^^	Ctl-^
Control- <u> </u>	^_	Ctl- <u> </u>
DEL	127	Ctl-BkSp
F1 - F10	k1 - k0	F1 - F10
F11 - F20	K1 - K0	Shft F1 - F10
Down Arrow	kd	Down
Home	kh	Home
Left Arrow	kl	Left
Right Arrow	kr	Right
Up Arrow	ku	Up
Insert Line	kA	Ctl-Ins
Tab Left	kB	Shft-Tab
Clear-to-End	kE	Ctl-End
Delete Line	kL	Ctl-Del
Page Down	kN	PgDn
Page Up	kP	PgUp
Cancel	Kc	-
Next Paragraph	Kd	Ctl-Down
Word Left	Kl	Ctl-Left
Word Right	Kr	Ctl-Right
Previous Paragraph	Ku	Ctl-Up
Exit	Kx	-

Key	Key Code (terminal db file)	Windows Keyboard
Attention	KA	-
Bottom	KB	Ctl-PgDn
Clear	KC	Ctl-Home
Command (Do)	KD	-
End	KE	End
Find	KF	-
Insert Character	KI	Ins
Page Left	KL	-
Mark (Select)	KM	-
Print	KP	-
Page Right	KR	-
Send	KS	-
Top	KT	Ctl-PgUp
Save	KV	-
Delete Character	KX	Del
Help	K?	-
User-defined keys 1 - 10 (1-6 on Windows)	U1 - U0 (U1-6 on Windows)	F11-F12; Shft-F11-F12; Ctl-F11-F12
User-defined keys 11-20	A1 - A0	Ctl-1 - Ctl-0

The following table lists mouse-action “keys” that can be referenced by a KEYSTROKE entry; this table has meaning only for graphical systems such as Windows. The table lists the mouse action, the corresponding key code, and the default exception value returned. See Book 2, *User Interface Programming*, Chapter 7, “Using the Mouse,” for details on mouse handling.

Action	Key Code	Exception Value
Mouse moved	Mv	80

Action	Key Code	Exception Value
Left button pushed	Ml	81
Left button released	ML	82
Left button double-clicked	M1	83
Middle button pushed	Mm	84
Middle button released	MM	85
Middle button double-clicked	M2	86
Right button pushed	Mr	87
Right button released	MR	88
Right button double-clicked	M3	89

The Host's Backspace and Line-Kill keys are not identified in the terminal database file. They are defined, instead, at the operating system level. The Backspace key is the key used to back up while you are typing command lines (usually either "backspace" or "delete"). The Line-Kill key is the one that is used to cancel an entire command line.

Control keys (Control plus another key) are not defined in the terminal database file. They are directly mapped by the runtime system to the corresponding control-key ASCII value. They can be referred to by either their ASCII value or by the key code listed. The DEL key does not have a key code; it can be referred to only by its ASCII value (127).

Some keys may have more than one name. When this occurs, the names have the following precedence:

1. Host name
2. Terminal database file name
3. Control-key name (if applicable)

For example, if a terminal whose left arrow key produces a Control-H is being used, and Control-H is the system's backspace key, that key would be treated as a Host's Backspace key (ZB). If the host's backspace were redefined (by operating system command) to be some other key, then this key

would be considered a Left Arrow key (kl). It would be considered a Control-H (^H) only if the terminal database file were edited and the “kI” definition changed or removed.

4.3.2.4 Additional Windows keys

In addition to the keys listed in section 4.3.2.3, these extra keys are available for 32-bit Windows systems:

Key	Key code
Ctl-Ins (Insert Line)	kA
Ctl-Del (Delete Line)	kL

User-defined keys

User-defined keys 1 - 6 (U1 - U6):

Key	Key code
F11	U1
F12	U2
Shft-F11	U3
Shft-F12	U4
Ctl-F11	U5
Ctl-F12	U6

User-defined keys 11 - 20 (A1 - A0):

Key	Key code
Ctl-1	A1
Ctl-2	A2
Ctl-3	A3
Ctl-4	A4
Ctl-5	A5

Key	Key code
Ctl-6	A6
Ctl-7	A7
Ctl-8	A8
Ctl-9	A9
Ctl-0	A0

Several function key combinations can also be redefined. These keys have no specified default action and the key combinations are recognized by the Windows keyboard driver only after they are assigned a definition.

Key	Key code
Alt-F1	a1
Alt-F2	a2
Alt-F3	a3
Alt-F5	a5
Alt-F7	a7
Alt-F8	a8
Alt-F9	a9
Alt-F10	a0
Alt-F11	U7
Alt-F12	U8
Shift-Ctl-F1	S1
Shift-Ctl-F2	S2
Shift-Ctl-F3	S3
Shift-Ctl-F4	S4
Shift-Ctl-F5	S5
Shift-Ctl-F6	S6
Shift-Ctl-F7	S7
Shift-Ctl-F8	S8

Key	Key code
Shift-Ctl-F9	S9
Shift-Ctl-F10	S0
Shift-Ctl-F11	U9
Shift-Ctl-F12	U0

Note: Alt-F4 and Alt-F6 are reserved for use by Windows and are not included in the table. Shift-Ctl-F10 may be used only if the configuration option F10_IS_MENU is set to “false”. When F10_IS_MENU is set to the default of “true”, then Shift-Ctl-F10 activates context menus (for example, a control’s pull-down menu).

Keys that cannot be defined

Alt key sequences (except as noted in the preceding table), PrtSrcn, and F10 are directly handled by 32-bit Windows and cannot be referenced in COBOL. You can free the F10 key to act as a user defined key by using a configuration variable. Setting the F10_IS_MENU variable to “0” inhibits the standard menu activation capability for the F10 key. See Appendix H for more details.

4.3.2.5 Special keys

The following keys deserve special attention.

Arrow keys

The left and right arrow keys can be configured to meet a variety of needs.

1. As **exception keys** only. In this case, typing an arrow key will cause an ACCEPT to terminate immediately with the arrow-key exception value. The program can then take the appropriate action (such as moving a highlight in the requested direction). To configure an arrow in this manner, define an EXCEPTION value for it with the KEYSTROKE runtime configuration variable.

2. As **edit keys** only. In this case, the arrows will move the cursor within the ACCEPT field, but will not move outside the boundaries of the field. In this mode, the arrow key will never terminate the ACCEPT. To configure an arrow in this manner, define the appropriate EDIT value for it with the KEYSTROKE runtime configuration variable.
3. As both **exception keys** and **edit keys**. In this mode, the arrows will act as edit keys within the ACCEPT field, but will act as exception keys when the user tries to move outside the field. This can be useful if you are writing a “fill-in-the-form” style of application. To configure an arrow in this manner, define both an EXCEPTION and an EDIT value for it.

By default, the left and right arrows act as edit keys, and the up and down arrows act as both edit and exception keys. You can change the behavior of the arrows at runtime to switch between different modes if you need to. You do this via the SET ENVIRONMENT verb and the appropriate KEYSTROKE settings. For example, to configure the left arrow to act as an editing key from within a program, use:

```
SET ENVIRONMENT "KEYSTROKE" TO "EDIT=Left k1"
```

Backspace vs. Left Arrow

On some terminals, the Backspace and Left Arrow keys send the same hardware signal. If so, ACUCOBOL-GT's key naming rules will treat both as a (destructive) Backspace, because the host name takes precedence. You can deal with this situation in one of several ways; some possibilities are:

1. If you do not use the Left Arrow key as anything other than an edit key, you can probably just use the defaults. You will not have the Left Arrow capability, but most users prefer to have destructive Backspace instead. Alternatively, if you prefer to have Left Arrow instead of destructive Backspace you can, with a KEYSTROKE variable, define the Backspace key to have the “Left” edit action.
2. If you use the Left Arrow as an exception key, then you can leave the destructive backspace action on the Backspace key and also give it an exception code value. This will cause the Backspace key to act as a destructive backspace while the cursor is in an ACCEPT field. The Left Arrow exception value will be returned when the user backspaces off the left edge of the field.

3. Finally, you can use operating system commands to assign the host's Backspace key to another key. This will then cause the Backspace key to be recognized as a Left Arrow key while the other key will take on the characteristics of the Backspace key. If you wish to do this, a common key to use as the alternate Backspace key is the Rub Out (or DEL) key.

Other combinations exist, but this should give you a general idea of ways to address this issue.

Interrupt key

ACUCOBOL-GT has no way of defining a key to be the asynchronous interrupt key. ACUCOBOL-GT makes use of the host's definition for this key. This has two effects:

1. If you want to define a special asynchronous interrupt key, you must do so at the operating system level.
2. Whichever key is used as the Interrupt key will be unavailable to you as a normal key. This is because the host operating system acts on this key prior to ACUCOBOL-GT's ever receiving it. ACUCOBOL-GT "sees" an interrupt when this key is typed; it never receives a character for it.

4.3.2.6 Default keyboard

The default ACUCOBOL-GT keyboard is defined below in the language of the KEYBOARD and KEYSTROKE runtime configuration variables.

KEYBOARD	Data-range=32,255	
KEYBOARD	Exception-range=1,31	
KEYBOARD	Auto-Return=0 Screen-Default=No	
KEYBOARD	RM-2-Default-Handling=No	
KEYBOARD	Check-Numbers=Yes	
KEYBOARD	Cursor-Past-End=No	
KEYSTROKE	Terminate=13	^M
KEYSTROKE	Edit=Next Terminate=9	^I

KEYSTROKE	Edit=Previous	kB
KEYSTROKE	Edit=Backspace	ZB
KEYSTROKE	Edit=Erase-Field	ZK
KEYSTROKE	Edit=First	kh
KEYSTROKE	Edit=Last	KE
KEYSTROKE	Edit=Auto-Insert	KI
KEYSTROKE	Edit=Delete	KX
KEYSTROKE	Edit=Erase-Field	KC
KEYSTROKE	Edit=Erase-to-End	kE
KEYSTROKE	Edit=Left	kl
KEYSTROKE	Edit=Right	kr
KEYSTROKE	Edit=Up Exception=52	ku
KEYSTROKE	Edit=Down Exception=53	kd
KEYSTROKE	Exception=67	kP
KEYSTROKE	Exception=68	kN
KEYSTROKE	Exception=40	KD
KEYSTROKE	Exception=90	K?
KEYSTROKE	Exception=1	k1
KEYSTROKE	Exception=2	k2
KEYSTROKE	Exception=3	k3
KEYSTROKE	Exception=4	k4
KEYSTROKE	Exception=5	k5
KEYSTROKE	Exception=6	k6
KEYSTROKE	Exception=7	k7
KEYSTROKE	Exception=8	k8
KEYSTROKE	Exception=9	k9
KEYSTROKE	Exception=10	k0
KEYSTROKE	Exception=11	K1
KEYSTROKE	Exception=12	K2

KEYSTROKE	Exception=13	K3
KEYSTROKE	Exception=14	K4
KEYSTROKE	Exception=15	K5
KEYSTROKE	Exception=16	K6
KEYSTROKE	Exception=17	K7
KEYSTROKE	Exception=18	K8
KEYSTROKE	Exception=19	K9
KEYSTROKE	Exception=20	K0

4.3.2.7 Modification examples

Following are examples of some common modifications to the default keyboard settings.

In the default keyboard, the “Tab” key is used to move from one field to the next. The “Return” key is used to terminate the ACCEPT. If you want the “Return” key to move the user to the next field instead of immediately terminating the ACCEPT, the following entry in the runtime configuration file will cause that to happen:

```
KEYSTROKE      EDIT=Next      TERMINATE=13    ^M
```

Alternately, you might want the “Return” key to clear the part of the field that follows the cursor. If you want to do this along with the previous modification, you can use either of these entries:

```
KEYSTROKE      EDIT=Erase-Next    TERMINATE=13    ^M
KEYSTROKE      EDIT=Default-Next  TERMINATE=13    ^M
```

These two lines have slightly different methods of handling how the field is cleared. The first version always erases the field from the current cursor location to the end. The second form does this only if the cursor is not in the home position of the field. You can also use the actions “Erase-to-End” or “Default-Entry” if you do not want the “Return” key to act as a “next field” key.

4.4 The Display Interface

The Terminal Manager's keyboard interface has been discussed above. The display interface can also be configured. Its task is to implement, for a particular terminal, those program instructions that specify display attributes. You can accomplish most desired display options by defining, in the terminal database file, the actions that *terminal function codes* will take. You can specify some other display options by assigning values to special keywords in the ACUCOBOL-GT runtime configuration file.

The steps listed below describe, in a simplified way, the overall process from COBOL statement to screen display:

Function code generation

- The COBOL program sends output to the screen. (For example, the COBOL statement might be: DISPLAY data-item HIGH.)
- The runtime configuration file may specify an attribute for the DISPLAY keyword. (Continuing with the example, the configuration file might include this entry: COLOR-MAP High=Blue.)
- The Terminal Manager maps the COBOL attributes to terminal function codes. (High=HI Blue=C2)

Function code interpretation

- The terminal database file maps the function codes to a hardware signal. (HI=\E[0;1m C2=\E[34m)
- The Terminal Manager sends a hardware signal to the screen. (\E[0;1m \E[34m)

Final result

- The display function is executed. (The data item characters are displayed at high intensity in blue.)

This section and **section 4.5** describe the runtime configuration file options. The terminal database file function codes and values are described in **section 4.6**.

4.4.1 Adding Color

ACUCOBOL-GT allows you to add color, without reprogramming, to programs that were originally written for black-and-white terminals. You accomplish this by assigning color values to the runtime configuration variable COLOR-MAP. The COLOR-MAP keyword is followed by one of the following single attributes:

High, Low, Reverse, Blink, Underline, Default, or Exit;

or by one of the following hyphenated combinations of attributes:

High-Reverse	Low-Reverse
High-Blink	Low-Blink
High-Reverse-Blink	Low-Reverse-Blink
High-Underline	Low-Underline
High-Reverse-Underline	Low-Reverse-Underline
Reverse-Blink	Reverse-Underline

The single attribute, or attribute combination, is then followed by an equals sign and one of the following color names:

Black, Blue, Green, Cyan, Red, Magenta, Brown, White

The named color becomes the foreground color that is displayed whenever the corresponding attribute is used in a DISPLAY statement. For example, if you want fields that are displayed as low-intensity to appear green, use the following configuration file entry:

```
COLOR-MAP Low=Green
```

You can also assign a background color value. It follows the foreground color and is separated from it by a comma. For example, to assign white characters on a blue background for high-intensity fields, you would use the following:

```
COLOR-MAP High=White,Blue
```

Note: No spaces should appear within the assignment.

You may specify more than one attribute in a single COLOR-MAP line. Simply separate the attributes from each other by spaces. For example:

```
COLOR-MAP High=Green Low=Red Reverse=Blue
```

The following points should be noted:

1. The named video attribute is still used by the ACCEPT or DISPLAY. For example, “Reverse=Blue” will result in a reverse-video blue field while “High=Brown” will use high-intensity brown (on some terminals, specifying High=Brown will cause yellow to be generated by any DISPLAY HIGH phrase).
2. If a particular ACCEPT or DISPLAY statement has a COLOR phrase, that phrase will be used instead of the COLOR-MAP attributes in the runtime configuration file. Note, however, that the COLOR-MAP *will* apply to fields that use the FCOLOR or BCOLOR options of the CONTROL phrase.
3. The attributes HIGH, LOW, and REVERSE are treated in a special manner. If a statement uses more than one of the three, then the attribute/color used will be in this order of preference:
 - 1) REVERSE
 - 2) HIGH
 - 3) LOW

For example, a DISPLAY statement specifying both REVERSE and HIGH will use the color associated with REVERSE.

Also, any one of these settings is used for all applicable cases, except where specifically overridden by another setting. For example:

```
COLOR-MAP REVERSE=Red LOW-REVERSE=Blue
```

will use red for all statements that specify REVERSE except for statements that explicitly specify REVERSE,LOW.

In all other cases, the configuration variable must exactly match the COBOL statement. For example, a configuration file attribute HIGH-REVERSE would not apply to a program statement that included HIGH,REVERSE,UNDERLINE.

Note that compiler options “-VI” and “-Vh” cause LOW and HIGH respectively to be implied for program statements; create your COLOR-MAP as though LOW or HIGH were explicitly coded in the program.

4. The DEFAULT attribute works a little differently. It is used to assign the initial default colors for the screen. Its effect is the same as having a DISPLAY WINDOW COLOR statement as the first statement of your program.
5. The EXIT attribute determines which colors ACUCOBOL-GT will set when it terminates. These colors are also set when a call is made to the “SYSTEM” library routine. On some machines these colors are immediately changed by the operating system prompt and thus have no effect.
6. You can assign values to the color map from within a COBOL program through use of the SET ENVIRONMENT verb. You can turn off the color map with the following statement:

```
SET ENVIRONMENT "COLOR-MAP" TO "OFF"
```

7. Note that on a XENIX console, if you use the XENIX command “setcolor” to establish a high-intensity background color, you may get unexpected results from ACUCOBOL-GT. This is because the implementation of high-intensity background colors causes XENIX to treat the “blink” bit as a background intensity bit instead. In addition, because ACUCOBOL-GT can select only 8 background colors, all of the background colors used will be high-intensity, including black (which shows up as a light gray).

For these reasons, we recommend that you avoid using a high-intensity background color if you are using the XENIX console. As an alternative, you may create a shell script to run ACUCOBOL-GT. This script could set a low-intensity background color, run ACUCOBOL-GT, and then reset the desired high-intensity background color.

4.4.2 The SCREEN Option

There is a runtime configuration variable called “SCREEN” that controls many features of the video sub-system. This option works in the same manner as the “KEYBOARD” variable. You can specify one or more

SCREEN variables. Attributes that you can set are identified by one or more sets of *keywords* and associated *values*, separated from each other by spaces or tabs; the syntax is:

```
SCREEN keyword=value [keyword=value]...
```

The following *keywords* are supported:

ALPHA-UPDATES=value	CONVERT-OUTPUT=value
EDITED-UPDATES=value	ERROR-BELL=value
ERROR-BOX=value	ERROR-COLOR=value
ERROR-LINE=value	FORM-FEED=value
INPUT-DISPLAY=value	INPUT-MODE=value
JUSTIFY=value	NUMERIC-UPDATES=value
PROMPT=value	PROMPT-ALL=value
PROMPT-ATTR=value	REFRESH-LINES=value
REFRESH-MODE=value	SHADOW-STYLE=value
SIZE=value	WINDOW=value

ALPHA-UPDATES=value

This option affects how alphanumeric fields with a default value are displayed prior to entry. It works just like the EDITED-UPDATES option (described below) except that it applies to alphanumeric fields instead of numeric edited fields. The only acceptable value is Unchanged.

Placing “Auto-Prompt” immediately after this option, using a comma as a separator, allows the user to decide whether to change or replace the default value. When Auto-Prompt is specified, the default value will be displayed, and then the program will wait for the user to enter a character. If the character entered is a data character, ACUCOBOL-GT will fill the field with prompt characters (erasing what was there) and then accept data as if this were a new field. If the character entered is an editing character (such as an arrow key), then ACUCOBOL-GT allows the user to edit the data normally. Sample syntax is shown here:

```
SCREEN ALPHA-UPDATES=Unchanged, Auto-Prompt
```


This option can also be specified as `SCRN_ALPHA_UPDATES`. The Auto-prompt value can be specified with `SCRN_ALPHA_AUTO_PROMPT`. For example, to set the above syntax using these variables, you would enter:

```
SCRN_ALPHA_UPDATES Unchanged
SCRN_ALPHA_AUTO_PROMPT on
```

CONVERT-OUTPUT=*value*

This option affects only Screen Section `DISPLAY` statements. If this keyword is set to “Yes”, then all output fields will act as if the `WITH CONVERSION` phrase were specified for them. This has two effects. The first is that numeric fields will be converted from the internal storage format to a readable form (including suppression of leading zeros). The second is that the action of the `JUSTIFY` keyword (see below) takes effect. This option is normally set to “No”, but is provided as an alternate method of displaying numeric data in the Screen Section. The configuration variable `SCRN_CONVERT_OUTPUT` is synonymous with this option.

EDITED-UPDATES=*value*

This option affects how numeric edited fields with a default value are displayed prior to the user making an entry. The four possible values are: `Converted`, `Unchanged`, `Left-Adjust`, and `Formatted`.

Converted is the default setting. When this setting is used, the default value is displayed in a standardized format. This format has an optional leading minus sign, followed by the number, with no leading zeros and no internal formatting characters.

Unchanged is an alternate setting. When this setting is used, the default value is displayed without any changes. All of the editing characters appear, and leading spaces are shown. Note that the `LEFT`, `RIGHT`, or `CENTER` phrase will affect the display. After the value has been displayed, the user can edit it normally.

- Left-Adjust** is identical to Unchanged, except that any leading spaces are removed before the value is displayed.
- Formatted** is fundamentally different from the other options in that it affects the way the number is entered, not just the format of the default value. When “Formatted” entry is selected, the number is continuously reformatted by the ACCEPT statement to match the editing specification of the item being entered. This means that the value will always appear to the user in its “final” form. This is similar to the way numbers are entered on most calculators. Selecting this option has many minor affects on the actions of various editing keys. These are not detailed here, but the actions of the editing keys are analogous to their actions on non-formatted fields.

There is one exception to the rule that the number will always be formatted just as described by the PICTURE clause. This is when “Z” or “*” characters are placed *after* the decimal point in the PICTURE. In this case, the entered characters will be treated like “9” characters instead. This is necessary in order to allow the user to enter values between zero and 1 when the default value is zero. If this rule did not exist, then when the user tried to enter the decimal point, the reformatter would keep removing it. The same applies to any zero digits between the decimal point and the first non-zero digit.

When the “Formatted” option is used with left justification, the entry action is also left justified. When it is used with the centering option, the entry occurs as if the field were right justified, and the final result is centered when the user leaves the field.

Place “Auto-Prompt” immediately after this option, using a comma as a separator, to allow the user to decide whether to change or replace the default value. When Auto-Prompt is specified, the default value will be displayed, and then the program will wait for the user to enter a character. If the character entered is a data character, ACUCOBOL-GT will fill the field with prompt characters (erasing what was there) and then accept data as if this were a new field. If the character entered is an editing character (such as an arrow key), then the program allows the user to edit the data normally. Sample syntax is shown here:

```
SCREEN EDITED-UPDATES=Converted, Auto-Prompt
```

This option can also be specified as `SCRN_EDITED_UPDATES`. The Auto-prompt value can be specified with `SCRN_EDITED_AUTO_PROMPT`.

ERROR-BELL=value

This option determines when the error bell will be sounded. Possible values are:

- Yes:** ring the bell on an entry error, but not on field-full. This is the default setting.
- No:** do not ring the bell on entry error or field-full.
- All:** ring the bell whenever the user makes an entry error or attempts to enter data into a full field.

For example, to use the “All” setting, add the following line to your runtime configuration file:

```
SCREEN ERROR-BELL=All
```

You may also use the configuration variable `SCRN_ERROR_BELL` to set these values. The variable `SCRN_WARN` is synonymous with `SCREEN ERROR-BELL=All`.

ERROR-BOX=value

This option affects whether an error box appears when an entry error has occurred. Examples of entry errors are entering a letter in a numeric field or entering a number in the wrong format. When value is set to “yes” (the default), the error message is displayed in a box. If value is set to “no”, the error is reported based on the entry in the `SCREEN ERROR-LINE` variable (below). The configuration variable `SCRN_ERROR_BOX` may also be specified.

ERROR-COLOR=value

This keyword is given a numeric value that represents the colors used in error messages generated by the runtime system. Value is the arithmetic sum of the numbers representing the colors and other attributes used in error messages generated by the runtime system. The following color values are accepted:

Color	Foreground	Background
Black	1	32
Blue	2	64
Green	3	96

Color	Foreground	Background
Cyan	4	128
Red	5	160
Magenta	6	192
Brown	7	224
White	8	256

You may specify other video attributes by adding the following values:

Reverse video	1024
Low intensity	2048
High intensity	4096
Underline	8192
Blink	16384
Protected	32768

Only one foreground color and one background color may be specified. If either is missing, the corresponding default for the current terminal window is used. High intensity and low intensity may not both be specified. If neither is specified, the default intensity is used.

For example, to get a blinking white foreground on a blue background, you would specify:

```
SCREEN ERROR-COLOR=16456  
(16456 = 8+64+16384)
```

The default value is “4096”, which causes the error messages to use the current colors with a high-intensity foreground. The configuration variable `SCRN_ERROR_COLOR` is also supported.

ERROR-LINE=value

Value is the line number you wish error messages to appear on. The runtime system pops up a one-line window on this line to display the message, and then removes it after the user responds. If this is set to a negative value, then the line used will be that many lines up from the bottom of the screen. For example, “Error-Line=-2” implies that the

next-to-last line should be used. The default value is “-1”. You may also specify the configuration variable `SCRN_ERROR_LINE` to set this value.

FORM-FEED=value

This option lets you use “Control-L” for a form feed. Setting this variable to “yes” and putting “Ctl-L” in a `DISPLAY` statement allows a form feed to occur. In effect, this clears the screen and puts the cursor at screen position (0,0). Setting this variable to “no” disallows a form feed. The default value is “no”. This can also be specified as `SCRN_FORM_FEED` instead of `SCREEN FORM-FEED`.

INPUT-DISPLAY=value

This option determines what happens when the `DISPLAY` verb operates on an input field described in a Screen Section entry. There are four choices: “None”, “Value”, “Spaces”, and “Prompt”.

None: The field is not displayed.

Prompt: The field is displayed with the field’s prompt character (usually underscore).

Spaces: The field is displayed as spaces. This is the default value.

Value: The current value of the field is displayed. This will be zero for numeric and numeric-edited fields, and spaces for other fields.

The configuration variable `SCRN_INPUT_DISPLAY` is also supported.

INPUT-MODE=value

This option affects pre-display of data in a Screen Section `ACCEPT`. The options are “Predisplay”, “Update”, and “Normal”.

Predisplay: A Screen Section `ACCEPT` statement will cause the current value of each input and update field to be displayed. (Whatever is present in the Screen Section is displayed; this is not necessarily the same as the contents of Working-Storage). Each field is then entered as an update field (i.e., the value can be edited).

Update: Each input field is treated as an update field. This causes the field's current value to echo on the screen when the field is visited.

Normal: Causes no echoing of input-only fields.

You may also specify the configuration variable `SCRN_INPUT_MODE`.

JUSTIFY=value

The JUSTIFY setting determines the default justification of converted numeric and numeric-edited fields. If "Left" is chosen, then leading spaces are removed from these fields when they are displayed. If "Right" is chosen, then the leading spaces are retained. Finally, if "Auto" is chosen (the default), then left justification is used if the program was compiled in RM/COBOL compatibility mode, otherwise right justification is used. Note that justification affects only fields that have the CONVERT phrase specified or implied for them. The configuration variable `SCRN_JUSTIFY` is also supported.

NUMERIC-UPDATES=value

This option affects how numeric fields with a default value are displayed prior to entry. This option works just like the "EDITED-UPDATES" option described above except that it applies to numeric fields instead of numeric edited fields. The possible values are Converted and Unchanged.

Place the phrase Auto-Prompt immediately after this option, using a comma as a separator, to allow the user to decide whether to change or replace the default value. When Auto-Prompt is specified, the default value will be displayed, and then the program will wait for the user to enter a character. If the character entered is a data character, ACUCOBOL-GT will fill the field with prompt characters (erasing what was there) and then accept data as if this were a new field. If the first character entered is an editing character (such as an arrow key), then ACUCOBOL-GT allows the user to edit the data normally. Sample syntax is shown here:

```
SCREEN NUMERIC-UPDATES=Converted, Auto-Prompt
```

This option can also be specified as `SCRN_NUMERIC_UPDATES`. The Auto-prompt value can be specified with `SCRN_NUMERIC_AUTO_PROMPT`.

PROMPT=value

The value of the PROMPT setting determines the default prompt character. The default value is underscore. To specify an alternate prompt, place the character immediately after the equals sign. To specify a space as the prompt character, leave the value empty (e.g., “Prompt= ”). You may also specify the configuration variable SCRN_PROMPT to set this value. The variable SCRN_PROMPT_DEFAULT is equivalent to setting SCREEN_PROMPT to the default value.

PROMPT-ALL=value

By default, a prompt character is shown only in the field containing the cursor. If *value* is “Yes”, then the prompt character is shown in every field managed by the ACCEPT statement. The prompt characters are removed when the ACCEPT is terminated. Prompts never appear in SECURE fields. Default is “No”. The configuration variable SCRN_PROMPT_ALL is synonymous with this option.

Note: Setting the SCREEN keyword PROMPT-ALL to the value “Protected” will have the same effect as setting PROMPT-ALL to “Yes”, except that prompt characters will *not* be displayed in protected fields.

PROMPT-ATTR=value

You may specify a prompt attribute. This attribute is used whenever the PROMPT is specified or implied for a Screen Section ACCEPT statement. The PROMPT-ATTR keyword is followed by a single attribute: High, Low, or Reverse. For example:

```
SCREEN PROMPT-ATTR=HIGH
```

The configuration variable SCRN_PROMPT_ATTR is also supported. The usage is:

```
SCRN_PROMPT_ATTR HIGH
```

REFRESH-LINES=value

Value specifies the number of screen lines to redisplay after the user has finished entering data into a field. This option is useful when the terminal or terminal emulator can accept Asian phonetic characters and translate them into ideograms. The entered characters will often overflow the displayed input field, but after translation, the resultant

ideogram(s) will not. This option will “clean up” the screen by redisplaying the affected lines with the ideograms in place. For example:

```
SCREEN REFRESH-LINES=3
```

After accepting input data, the Terminal Manager will redisplay the contents of the input field, the remainder of the line, and the two lines below it.

If the `CODE_SYSTEM` runtime configuration variable (see [section 4.4.4](#)) is non-zero, specifying an Asian double-byte character system, the default value of `REFRESH-LINES` is “1”. If the `CODE_SYSTEM` runtime configuration variable is set to “0”, indicating a single-byte ASCII or EBCDIC character system, the default value of `REFRESH-LINES` is “0”. You may also use the configuration variable `SCRN_REFRESH_LINES` to set these values.

REFRESH-MODE=*value*

This option, like `REFRESH-LINES`, supports double-byte character sets. *Value* specifies when lines should be refreshed after an `ACCEPT`. Setting this variable to a value of “0” means that the lines are never refreshed, “2” indicates that lines are always refreshed. The default value of “1” specifies that lines are refreshed only if double-byte characters are entered. For example:

```
SCREEN REFRESH-MODE=1
```

The configuration variable `SCRN_REFRESH_MODE` is synonymous with this option.

SHADOW-STYLE=*value*

This option determines the way window shadows are displayed. It may have one of the following four values:

- None:** When this setting is used, shadows are not displayed.
- Dim:** This setting displays a one-character border around the right and bottom edges of the window. This border displays the underlying data in low-intensity with a white foreground and a black background; in effect, the border is translucent. This border looks best when the shadowed window and the window it overlays do not both have black backgrounds.

- Black:** This setting displays a black border on the right and bottom edges of the window. On the right edge, this border is one character wide. On the bottom edge, the border is one-half character high. This gives a fairly uniform appearance to the border. The border depends on the existence of an “upper-half” block character on the display device. For machines that use a terminal database file, this character should be specified as the 12th character in the GM code in the terminal database file (GM defines the various graphics characters). Also, we recommend that you specify a “lower-half” character as the 13th GM character. If such characters do not exist, then the bottom border is a full character high. The Black setting is the default shadow style.
- Lines:** This setting causes the right and bottom edges to be shown with a border made from the line drawing set. This setting is not as appealing as the Dim or Black settings when color or reverse-video backgrounds are being used. When the background is black, however, this setting is preferable to the other two.

The configuration variable `SCRN_SHADOW_STYLE` is also supported.

SIZE=*value*

This keyword has meaning only on graphical systems such as Windows. It is used to change the default virtual screen size. *Value* is the desired number of rows and columns, separated by a comma.

For example, to set the initial virtual screen size to 30 rows by 80 columns, you would make the following entry:

```
SCREEN    SIZE=30,80
```

The comma is required.

The size of your virtual screen is independent of the size of the application window or the underlying hardware. In other words, the virtual screen can be larger than the physical screen. You may set any screen size up to a maximum of 100 rows and 200 columns. If you do not specify a size, the default is 25 rows and 80 columns. You may also use the configuration variables `SCRN_SIZE_COLS` and `SCRN_SIZE_ROWS` to set this option.

The SIZE option sets only the initial screen size. After the application begins, the screen size can be changed with the DISPLAY SCREEN SIZE verb.

If the virtual screen is too large to be fully displayed on the physical screen, the user will have to scroll to view all of the rows and columns.

WINDOW=value

This keyword has meaning only on graphical systems such as Windows. Normally, the initial size of an application's window is determined by the host. You can change this initial size with the WINDOW keyword. *Value* is the desired number of rows and columns, separated by a comma.

For example, if you wanted your initial window to contain 10 rows and 70 columns, you would enter:

```
SCREEN WINDOW=10,70
```

The WINDOW configuration option has several special values that it recognizes. If *either* the row or column is set to a negative number, then the initial window is minimized (turned into an icon). If either value is set to "999" or larger, then the initial window is maximized instead. Finally, if either value is zero, then the initial window size is determined by the host system (this is the default).

The application window size may never be bigger than the virtual screen size, nor may the window size be larger than what can be physically displayed on the user's screen. This physical limit will change depending on the resolution of the user's screen and the size of the font you are using. The ACUCOBOL-GT runtime will automatically reduce the requested window size to meet these limits.

You may enter the SIZE and WINDOW options on the same line. For example, if you wanted your application to be able to use 30 lines by 80 columns, and you wanted to start with the window maximized (thus showing the entire virtual screen), you would enter:

```
SCREEN SIZE=30,80 WINDOW=999,999
```

Note: The `SIZE` and `WINDOW` options set only the initial screen and window size. After the application begins, the user is free to change the window size with various system controls, and the application is free to change the screen size with the `DISPLAY SCREEN SIZE` verb.

The configuration variables `SCRN_WINDOW_X` and `SCRN_WINDOW_Y` are also supported for this option.

4.4.2.1 SCREEN examples

The following sample recaps the default settings and provides an example of how to specify `SCREEN` configuration entries.

```
SCREEN      Convert-Output=No
SCREEN      Edited-Updates=Converted, Auto-Prompt
SCREEN      Error-Bell=Yes
SCREEN      Error-Color=4096 Error-Line=-1
SCREEN      Input-Display=Spaces Input-Mode=Normal
SCREEN      Prompt=_ Justify=Auto
SCREEN      Numeric-Updates=Converted, Auto-Prompt
SCREEN      Alpha-Updates=Unchanged
SCREEN      Shadow-Style=Black
```

4.4.3 Additional Configuration Variables

Several miscellaneous runtime configuration variables affect the Terminal Manager. These are described below.

Note that for the variables `AUTO_PROMPT`, `BELL`, `MONOCHROME`, `SCROLL` and `WRAP`, the settings **1**, **on**, **true**, and **yes** are synonymous, as are the values **0**, **off**, **false** and **no**.

AUTO_PROMPT

When set to a non-zero value, the AUTO-PROMPT runtime configuration variable causes every ACCEPT statement without a PROMPT phrase to be treated as if PROMPT SPACES were specified. This has the effect of erasing the field where the data is about to be entered. This is provided primarily for compatibility with ACUCOBOL-85 version 1.1, which behaved this way. The default setting for this variable is zero.

BELL

When set to a zero value, the BELL variable suppresses *all* bells generated by ACCEPT and DISPLAY statements. This will make ACUCOBOL-GT totally quiet even if WITH BELL phrases are used on DISPLAY statements. The default setting is one.

HOT_KEY

This variable associates an exception value or values with a program. When a key with a specified exception value is pressed, the corresponding program is run. This variable is described in detail in Appendix H.

MONOCHROME

When set to a non-zero value, this variable disables color output for Windows machines with graphics video cards.

ACUCOBOL-GT assumes that all Windows machines with graphics video cards have color monitors (because the card has color abilities). If you have a monochrome monitor attached to such a machine, the results can be difficult to see. You can tell ACUCOBOL-GT to disable color output for these monitors through the Monochrome option.

When this is set to a non-zero value, ACUCOBOL-GT will use only black and white. The default value is zero. Note that you may change this in your program by using the SET ENVIRONMENT verb; ACUCOBOL-GT examines the MONOCHROME setting each time it does screen output.

RESTRICTED_VIDEO_MODE

This variable controls the rules ACUCOBOL-GT uses when displaying data on a terminal with “non-hidden” attributes (sometimes called “magic cookies”). See [section 4.5, “Restricted Attribute Handling,”](#) later in this chapter for a discussion.

SCROLL

When set to zero, the SCROLL variable inhibits screen scrolling, except scrolling caused by explicit SCROLL phrases in ACCEPT and DISPLAY statements. If a line wraps on the bottom line of the screen, the screen will not be scrolled if SCROLL is set to zero, but the line wrapping will still occur; it will overwrite the bottom line. Normally, ACUCOBOL-GT will scroll the screen to bring a DISPLAY line onto the screen if its line number is past the bottom edge of the screen. When SCROLL is set to zero, this does not occur and the cursor location becomes undefined (see the Note at the end of this section).

WRAP

The WRAP variable controls whether line wrapping is allowed. Normally, a DISPLAY statement that does not fit onto one line will wrap around to the next line. When WRAP is set to zero, this does not occur and the DISPLAY statement is truncated at the end of the line. Also, ACUCOBOL-GT normally wraps around to bring the column position specified for an ACCEPT or DISPLAY statement onto the screen. If WRAP is set to zero, the cursor location becomes undefined (see the Note at the end of this section).

Note: If WRAP or SCROLL is set to zero, the screen cursor location can be placed into an undefined state. This can occur, for example, if the WRAP setting causes a DISPLAY statement to truncate. This would leave the cursor conceptually just off the right edge of the screen. When this occurs, ACUCOBOL-GT inhibits further DISPLAY statements until the cursor is placed back on the screen via one of the normal positioning rules (ACUCOBOL-GT continues to track the cursor's logical location). Should an ACCEPT statement execute in an undefined location, ACUCOBOL-GT places the ACCEPT field in the home position of the current window.

4.4.4 Double-Byte Character Handling

Asian character sets contain large numbers of ideographic characters that represent an entire or partial word or concept. They may also contain interspersed phonetic characters. They may therefore consist of tens of

thousands of characters. Because one 8-bit byte can hold only 256 unique codes, these languages require at least two bytes to represent each character, in order to accommodate the full range.

Most double-byte characters occupy two full character screen positions (each byte corresponds to one screen position). Such data may be entered into and displayed from USAGE DISPLAY data items. Most COBOL applications can therefore accept and store double-byte data without modification.

Problems can arise when double-byte data is displayed on the screen. For example, during an ACCEPT, one byte of a double-byte character may be deleted or overwritten. When a window is displayed, the edge of the window might cover one byte of a double-byte character. In these circumstances, the pairing of bytes can change, and the resulting codes may represent entirely different characters. On most machines this confuses the operating system's display driver. To overcome these potential problems, the runtime must follow two rules:

1. Always display both bytes of a double-byte character together (never display only part of a double-byte character).
2. Always overwrite, or change the attributes of, both bytes of a double-byte character together (never overwrite, or change the attributes of, only part of a double-byte character).

These rules must be obeyed when an ACCEPT handles cursor movement, cursor placement, text selection, delete, backspace, and character overtyping.

The rules must also be followed when the edges of windows are displayed, to avoid covering parts of double-byte characters.

To implement these rules, the runtime needs to know which of several double-byte character encoding schemes is being used. It gets this information from the value of the configuration variable "CODE-SYSTEM." See Appendix H for a detailed discussion of this variable.

4.5 Restricted Attribute Handling

The ACUCOBOL-GT Terminal Manager assumes that video attributes can be applied individually to each character on the screen. This is the way most personal computers work with ANSI-conforming terminals. Several popular terminals, however, do not behave this way. This section discusses how ACUCOBOL-GT treats these terminals and what restrictions they impose.

Note: The rest of this section does not apply to Windows implementations of ACUCOBOL-GT except those using the alternate runtime with a terminal database file. If you are a Windows user and plan to move your programs to UNIX or VMS systems, you may want to read this section to familiarize yourself with the restrictions these environments impose.

Some terminals implement video attributes by a method that conflicts with the assumptions of the Terminal Manager. These terminals have special characters that show on the screen as a space, but set a display attribute for succeeding characters. That attribute is applied until another attribute-setting character is found. If one of these special characters is overwritten, its attribute will not be set.

UNIX documentation calls these attribute characters “magic cookies.” They are also sometimes called “non-hidden attributes.” Two terminals that use this style of attribute handling are the Televideo 925 and the Wyse 50.

This type of terminal poses special problems. One issue is where to place the attribute character. If it is placed in the first location of the field, the data in the field will be moved over one character position, resulting in a different display than on other types of terminals or personal computers. If it is placed just before the field, it might overwrite some valid data. Combining attribute characters with windows is even more intricate. The next section describes the rules ACUCOBOL-GT follows when accessing this type of terminal.

4.5.1 Restricted Video Modes

The action of ACUCOBOL-GT on a terminal with “non-hidden” attributes is determined by the setting of the `RESTRICTED_VIDEO_MODE` runtime configuration variable. This variable can take several different settings to control the rules ACUCOBOL-GT uses for these terminals.

Note: The following rules do not apply to intensity. These terminals can apply intensity attributes individually to each screen position. The Terminal Manager treats high and low intensity in the normal manner for these types of terminals.

By default, the `RESTRICTED-VIDEO-MODE` value is zero, which causes the Terminal Manager to ignore attributes other than intensity; the application will run correctly, but without any video attributes. This is convenient when you are running a program that has not been written to conform to the following rules.

To use video attributes with these terminals, you must set `RESTRICTED-VIDEO-MODE` to a non-zero value; the syntax is:

```
RESTRICTED-VIDEO-MODE value
```


Optional *values* are:

- 1** When the variable is set to “1”, the Terminal Manager uses rules that tend to emphasize getting the fields in the right location over getting all the attributes correct. These rules are as follows:
 - Every ACCEPT and DISPLAY is preceded by the appropriate attribute-setting character.
 - This character is placed immediately to the left of the beginning of the field. Note that this may overwrite existing data.
 - If the field position is column 1 of the current window, and the attribute is normal white on black, then the attribute-setting character is *not* displayed.
 - If the field position is column 1 of the current window, and the attribute is other than white on black, the field is moved over to column 2 to allow space for the attribute character.
 - The field is then accepted or displayed using the normal rules.
 - If the screen location immediately after the end of the field does not contain an attribute-setting character, a normal white-on-black attribute character is placed there. If this statement is an ACCEPT statement, this is done before the ACCEPT occurs. The current cursor location is then set according to the normal ACUCOBOL-GT rules (this will cause the cursor location to be where this terminating attribute character is located).
- 3** When RESTRICTED-VIDEO-MODE is set to “3”, the Terminal Manager follows all the rules listed under *value* “1” except for rule (c). This causes all ACCEPT and DISPLAY statements that reference column 1 to be placed in column 2. This setting prevents you from placing data in column 1, but causes all fields placed in column 1 to line up vertically regardless of which attributes they use.
- 5** When RESTRICTED-VIDEO-MODE is set to “5”, the Terminal Manager follows all the rules listed under “1” except for rule (b). The attribute character is placed in the first position of the field, and the field is moved to the right one character. This setting will cause all fields to shift to the right by one, but will not overwrite data if two fields are adjacent.
- 7** When RESTRICTED-VIDEO-MODE is set to “7”, the Terminal Manager follows all the rules listed for “1” except for rules (b) and (c). Thus, every ACCEPT and DISPLAY will always be preceded by an attribute character, and this character will always occupy the first field position. This *value* emphasizes getting the attributes correct over getting the fields in the correct screen location.

These rules give a certain amount of flexibility, but also have restrictions. These are discussed in the next section.

4.5.1.1 Restrictions

The following restrictions apply to programs that plan to use “non-hidden attribute” terminals. The restrictions are largely based on physical attributes of these terminals. In essence, by setting RESTRICTED-VIDEO-MODE to a non-zero value, you are declaring to ACUCOBOL-GT that you are willing to work with some restrictions beyond those imposed by other types of terminals. The end user should be aware that moving an application to this type of terminal from a “normal” type may result in unexpected effects.

The following restrictions apply:

1. Under the current version of ACUCOBOL-GT, this style of attribute handling may be applied only at the field and Screen Section levels. If you are using one of these types of terminals, the REVERSED and COLOR phrases of the DISPLAY WINDOW, DISPLAY LINE, and DISPLAY BOX verbs will be ignored.
2. The Terminal Manager makes no attempt to control the screen attributes present when a window is created. If you create a pop-up window over one-half of a reverse-video field, and then you clear that window, the reverse-video field will suddenly extend across the screen when the terminating attribute character is erased. You should keep fields either wholly contained in a window or wholly outside a window.
3. The various RESTRICTED-VIDEO-MODE settings can interact with SCROLL and WRAP settings in unexpected ways. For example, if you have a field wrap-around, the video attribute used for that field will also wrap around for some terminals, but not for others. On the other hand, if you set WRAP to zero and cause a field to be truncated, then the terminating attribute character will not be placed on the screen, and the video attribute may wrap around to the next line on some terminals. Care should be taken with fields that wrap around or scroll the screen.

4. If you position one field within another, you will affect the attributes of the characters that follow the contained field. Keep your fields separate from each other and supply enough space between fields to hold the attribute characters.

These restrictions are relatively easy to work with until you start working extensively with windows. When working with windows, try to keep the use of attributes to a minimum (particularly reverse-video) in order to avoid difficulties. You can use high and low intensity or boxes to organize your screen. Just use reverse-video for special highlighting.

If you intend to use video attributes on these types of terminals, then you should make sure that you fully test your programs on one of them.

4.6 The Terminal Database File

The terminal database file, which is similar to the **termcap** file supplied with many UNIX systems, may be edited to add new terminals to the ones it currently supports. Existing entries in the file may also be edited, if needed, to describe your terminal.

Each line of this file is either blank, a comment (marked by a “#” in column 1), or a definition of a terminal. You can continue a “line” on following lines by ending the line to be continued with a “\” (see below for an example). The “\” character must be the last character on the line.

A terminal definition consists of several fields, separated by colons. The end of the line marks the end of the definition. The first field is always the name of the terminal. Several names can be placed here, separated by a vertical bar (“|”). The rest of the fields consist of codes that describe various terminal functions. Most of these codes are followed by an equals sign and a coded string that describes how to operate that particular function.

Here is a generic representation of a terminal database file entry, where **TN_n** is a terminal name, **tf** is a terminal function code, and **cs** is a coded string to accomplish the function (some terminal function codes are self-defining and do not need a coded string):

```
TN1 | TN2 | TN3 : \
: tf [=cs] : tf [=cs] : tf [=cs] : \
```

:tf [=cs] :tf [=cs] :

The coded string that describes a function is just a representation of the control-sequence (or sequences) that the terminal uses to activate that function. These strings consist of the literal characters used in the control-sequence. Several special forms are recognized to aid in describing the control-sequence. The following abbreviations are supported:

<code>\E</code>	an escape character
<code>\n</code>	a newline (control-J)
<code>\r</code>	a carriage return (control-M)
<code>\t</code>	a tab (control-I)
<code>\b</code>	a backspace (control-H)
<code>\f</code>	a form-feed (control-L)
<code>^X</code>	X is any character, treated as control-X
<code>\nnn</code>	three digits treated as an octal value

The following is a list of all of the supported function codes. The most commonly used codes will be treated in detail in the following sections.

AC	Attributes used by clear screen
AT	Special color for IBM 3164 terminal
B1 - B8	Background color 1-8
BL	Blink
C1 - C8	Foreground color 1-8
DI	De-initialization string
DL	Default intensity is low
DP	Disable print mode
EP	Enable print mode
GA	Graphics on and off are characters
GE	Graphic escape
GF	Graphics off
GM	Graphics map

GO	Graphics on
GX	Graphics movement glitch
HI	High-intensity, normal video
LO	Low-intensity, normal video
NM	Normal Video (only if isgî set)
NS	Screen does not scroll when corner is used
OC	One color can be displayed at a time
RA	Reverse video, alternate intensity
RB	Reverse video, blink
RU	Reverse video, underline
RV	Reverse video
UL	Underline
W3	Set terminal width to 132 columns
W8	Set terminal width to 80 columns
al	Insert (add) line
bc	Backspace cursor (defaults to ^H)
cd	Clear to end-of-screen
ce	Clear to end-of-line
cl	Clear screen
cm	Cursor positioning
co	Number of screen columns (default 80)
dl	Delete line
do	Down one line (defaults to ^J)
is	Initialization string
is1	Additional initialization string
is2	Additional initialization string
li	Number of screen lines (default 24)
nd	Non-destructive space
sg	Standout-mode glitch (uses magic cookies)

tc	Continue description with another entry
up	Cursor up one line
ve	Set cursor to normal
vi	Set cursor to invisible
vs	Set cursor to bright

The following codes are also available to represent various keys. Most terminals have only a subset of this full set.

K1 - K0	Function keys 11 - 20
K?	Help
KA	Attention
KB	Bottom
KC	Clear
KD	Do (command)
KE	End
KF	Find
KI	Insert character
KL	Page left
KM	Mark (select)
KP	Print
KR	Page right
KS	Send
KT	Top
KV	Save
KX	Delete character
Kc	Cancel
Kl	Word left
Kr	Word right
Kx	Exit

k1 - k0	Function keys 1 - 10
kA	Insert line
kB	Tab left
kE	Clear to end
kL	Delete line
kN	Page Down
kP	Page Up
kd	Down arrow
kh	Home
kl	Left arrow
kr	Right arrow
ku	Up arrow
U1 - U0	User defined key 1 - 10
A1 - A0	User defined key 11 - 20

All of the function codes described with lower-case characters are identical to ones found in the UNIX **termcap** file. These sequences can be taken verbatim from **termcap** and included in the ACUCOBOL-GT terminal database file when you are adding a new terminal entry.

To help with this discussion, an example of an entry for a DEC VT-100 will be developed. At each step of the example, the new portion of the entry will be in **bold type**. Initially, we need to assign a set of names that we want to use to refer to the terminal. For example:

```
vt100|vt-100|DEC VT-100:
```

This allows for any of the names “vt100”, “vt-100” or “DEC VT-100” to be used for the TERM or A_TERM variable. By convention, the last name in the list is a long, descriptive name.

4.6.1 Required Functions

In order for the Terminal Manager to run, four functions must be defined for the terminal; all of the remaining functions are optional. These required functions are **Cursor-positioning** (cm), **Clear-screen** (cl), **Clear-to-end-of-line** (ce), and **Clear-to-end-of-screen** (cd). If these functions are not present when the Terminal Manager tries to run, an error will be printed and the program halted.

The Clear-screen function should clear the entire screen and home the cursor. The clear-to-end-of-line function should clear from the cursor position to the end of the current line. The clear-to-end-of-screen function should clear from the cursor position to the end of the screen.

The Terminal Manager starts by establishing a window that is the size of the screen. By default, a screen size of 24 by 80 is assumed. If this is not correct, you can set the Lines (li) and Columns (co) fields to the correct size. These settings are made with a “#” instead of an “=“. For example, if you have a 25-line terminal, the proper setting is “li#25”.

Continuing with our example, the DEC VT-100 clears the screen by sending an “ESC[2J”. Unfortunately, this does not home the cursor. This can be accomplished by sending “ESC[;H”. These can be sent in either order. Clearing to the end of line is done by sending “ESC[K” and clearing to the end of the screen by “ESC[J”. The terminal has the default screen dimensions, so we do not need to add the “co” or “li” options. Our entry now reads:

```
vt100|vt-100|DEC VT-100:\
:cl=\E[;H\E[2J:ce=\E[K:cd=\E[J:
```

Cursor positioning is accomplished by a special encoded form. The program must specify varying information in the control-sequence (the row and column numbers). Special abbreviations are allowed to encode this information. These abbreviations and their meanings are:

- %d** Inserts the row or column number here in ASCII. For example, row 5 would be inserted here as “5”.
- %2** Acts like “%d” except that it always prints as two digits. Row 5 is inserted as “05”.

%3	Acts like %2 except that three digits are used.
%.	Inserts the row or column number here literally. Row 5 would be inserted here as a decimal 5 (ASCII control-E). Note that if this type is used, then Cursor-up (up) and Backspace-cursor (bc) must also be defined.
%+x	Acts like “%.” except that x is added to the row or column number first. If the sequence were “%+ ” (note the trailing space), then row 5 would be inserted here as the sum of the space character and 5, “%” in ASCII. This form is quite common.
%>xy	This does not insert anything in the string. If the row or column number is greater than x, then y is added, otherwise this has no effect.
%r	Normally the row is inserted first, and then the column. This reverses the order.
%i	Normally the row and column numbers are relative to zero. Including this causes them to be relative to 1.
%%	Sends a literal “%”.

For example, the ADM-3A terminal positions the cursor by sending an “ESC=“ followed by the row and column offset by a space character. The code for this is “\E=%+ %+ ” (note spaces).

The VT-100 positions the cursor by sending an “ESC[“ followed by the row, a semicolon, the column and then an “H”. The row and column are sent as ASCII strings and the home position is row 1, column 1. The correct string is “\E[%i%d;%dH”.

```
vt100|vt-100|DEC VT-100 :\
:cl=\E[;H\E[2J:ce=\E[K:cd=\E[J:\
:cm=\E[%i%d;%dH:
```

4.6.2 Additional Screen Functions

Several additional functions are available to manipulate the screen display. These should be included if the terminal supports these features. The functions are: **insert-line** (al), **delete-line** (dl), **non-destructive-space** (nd), **backspace-cursor** (bc), **cursor-down** (do), **cursor-up** (up), **set-width-132** (W3) and **set-width-80** (W8).

The four cursor movement commands are available to optimize cursor motion. The non-destructive-space function should move the cursor to the right one column; the backspace-cursor function should move the cursor left one column. Finally, the cursor-down function should move the cursor down one line and cursor-up should move it up one line. If omitted, cursor-down defaults to a line-feed character, and backspace-cursor defaults to a backspace character. There are no defaults for non-destructive-space and cursor-up.

The insert-line function should insert a blank line at the cursor line, moving the cursor line and all following lines downward. The delete-line function should delete the cursor line, moving all following lines up and inserting a blank line at the bottom of the screen.

NS should be added to the terminal database file entry for a terminal that does not scroll if the lower right corner of the screen is filled. This tells the ACUCOBOL-GT program that it is all right to use this position. NS is the complete sequence (... :NS: ...).

The set-width functions should change the display between 132-column mode and 80-column mode. Both must be specified to use this feature.

You can also specify when the cursor should be visible. These entries should handle cursor modification:

```
ve = set cursor to normal
vs = set cursor to bright
vi = set cursor to invisible
```

After “vi” has rendered the cursor invisible, “ve” is used to make it visible.

If your terminal does not have both a normal and a bright cursor, then set the “ve” entry to turn the cursor on and do not use the “vs” entry.

The VT-100 supports only one of these functions: Non-destructive-space. This is accomplished by sending “ESC[C”. Our current entry is now:

```
vt100|vt-100|DEC VT-100 :\
:cl=\E[;H\E[2J:ce=\E[K:cd=\E[J:\
:cm=\E[%i%d;%dH:\
:nd=\E[C:
```

4.6.3 Video Attributes

To correctly configure attributes for a terminal, you must first determine which style of attribute setting--ANSI or “magic cookie”--it uses. You can do this most easily by typing the sequence to turn on reverse video at your terminal. If the cursor moves one character and a reverse-video bar appears, then you have a “magic cookie” style of terminal. If nothing happens, then type some characters. These should show up in reverse video. If they do, then you have an ANSI style terminal that allows for independent attributes for each screen position. If you do not get reverse-video at all, then you did something wrong.

If you include RV, UL, BL, RU, or RB in your terminal database file entry, then the HI and LO functions must be included. These two functions set the terminal to normal video/high intensity and normal video/low intensity, respectively. If intensity is not being used, then these should both just set normal video.

On all machines except Windows, the runtime system ignores the difference between high-intensity spaces and low-intensity spaces when the background color is black. If your terminal is set up to run with black-on-white characters (reverse video) as its default, you should add the entry VB (visible background) to the description of that terminal. This causes spaces to be handled consistently.

If a “magic cookie” style terminal is being used, then HI and LO should not set normal video, but should just set the appropriate intensity. The function NM should be added to set normal video instead. Also, the function sg must be included to tell the Terminal Manager that this is a “magic cookie” type terminal. The sg setting does not take a value, it just has to be present.

A few “magic cookie” terminals ignore HI and LO, so that reverse video fields appear the same regardless of which intensity is used. If you are experiencing this situation, add RA to the terminal’s description. This sets the terminal into reverse video using the terminal’s alternate intensity (usually low intensity). If RA is used, then RV sets reverse video in the terminal’s default intensity.

The function DL should be included in a definition if the default intensity for the terminal is low-intensity. This function is not set to a value, it is just included in the terminal definition.

On some terminals, a clear screen operation uses the currently selected video attribute. For example, if reverse-video were the current attribute, then a clear screen would cause the entire screen to become reverse-video. If the terminal has this property, then AC should be included to indicate this. ACUCOBOL-GT will use this to optimize certain screen displays.

Continuing the example, the VT-100 allows the independent setting of each attribute. It cannot independently reset the attributes, but that is not required by the Terminal Manager. Low-intensity, normal-video can be set with “ESC[m”. High-intensity can be set with “ESC[1m”. Reverse video is initiated by sending “ESC[7m”, underline with “ESC[4m” and blink with “ESC[5m”. The terminal normally runs in low-intensity, so the DL flag is used. All of these modes can be combined by placing the appropriate attribute numbers together in one command string and separating them with semicolons. Our new entry becomes:

```
vt100|vt-100|DEC VT-100 :\
:cl=\E[;H\E[2J:ce=\E[K:cd=\E[J:\
:cm=\E[%i%d;%dH:nd=\E[C:\
:LO=\E[m:HI=\E[0;1m:RV=\E[7m:\
:UL=\E[4m:BL=\E[5m:RU=\E[4;7m:\
:RB=\E[5;7m:DL:
```

Note the setting of “\E[0;1m” for HI. The initial zero ensures that the terminal is set to normal modes before the high-intensity mode is set.

4.6.4 Color

The Terminal Manager can support terminals that support ANSI-style attribute handling. This style allows for independent setting of the background and foreground colors and does not interfere with the setting of other video attributes such as underlining. Color terminals that meet these criteria can enable color by adding entries to turn on the various foreground and background colors. Use the following table of attribute codes to set the correct color entries; if your terminal has fewer than eight colors, you should still make all eight entries for both foreground and background, repeating colors as necessary:

Foreground	Background	Color
C1	B1	Black

Foreground	Background	Color
C2	B2	Blue
C3	B3	Green
C4	B4	Cyan
C5	B5	Red
C6	B6	Magenta
C7	B7	Brown
C8	B8	White

Terminal definitions for color terminals do not need the RV, RB, and RU entries, because the Terminal Manager sends the appropriate foreground and background color codes instead.

4.6.4.1 One-color terminals

Some terminals can display text in only one color at a time. On these terminals it is impossible to display text with separate foreground and background colors, unless one of the colors is black. The termcap code OC (one color) tells the runtime to use special color handling to accommodate a terminal of this type.

If this code is present in the terminal's database entry, the runtime displays text using the correct color combinations set from COBOL, so long as either the foreground or background color is set to black. If neither the foreground nor the background is set to black, the runtime displays text using the foreground color (the background color is disregarded).

To use this code, add it to the terminal database entry preceded and followed by a colon (:OC:).

4.6.5 Function Keys and Other Keys

Function keys and other special keys are simple to deal with. The various key entries are set to the values that those keys send when they are pressed. All of the key entries are optional. The table at the beginning of this section lists all of the available key codes.

As a minimum, you should define the arrow keys and some function keys. Most programs use these keys.

4.6.5.1 User-defined keys

The User-defined keys (“U1” - “U0”) are available for any keys that are not defined in the table above. These can be used for special purposes.

The VT-100 has the four arrow keys and function keys 1 through 4. The function keys send “**ESC**” followed by a distinguishing character, and the arrow keys send “**ESC**[“ and a distinguishing character. The new entry is:

```
vt100|vt-100|DEC VT-100 :\
:cl=\E[;H\E[2J:ce=\E[K:cd=\E[J:\
:cm=\E[%i%d;%dH:nd=\E[C:\
:LO=\E[m:HI=\E[0;1m:RV=\E[7m:\
:UL=\E[4m:BL=\E[5m:RU=\E[4;7m:\
:RB=\E[5;7m:DL:\
:k1=\EOP:k2=\EOQ:\
:k3=\EOR:k4=\EOS:ku=\E[A:\
:kd=\E[B:kr=\E[C:kl=\E[D:
```

4.6.6 Line Drawing

Some terminals support a line drawing set. This is used by the Terminal Manager when boxes are drawn around windows. The Terminal Manager turns on the “graphics” mode by sending the GO code, then sends normal characters that correspond to the lines, and then sets the terminal back to normal mode with GF.

The GM function lists the normal characters that draw the line segments. This is either a six- or eleven- or thirteen-character string. The characters listed in the GM function correspond, in order, with the following line segments:

1. horizontal line
2. vertical line
3. upper left corner
4. upper right corner
5. lower left corner
6. lower right corner

This is the six-character set. If the terminal has the following line segments, the characters that correspond to them should be included (in order) to make the eleven-character set:

four three-way intersections:

7. missing bottom line
8. missing left line
9. missing top line
10. missing right line
11. the four-way intersection

This is the eleven-character set. If the terminal has the following block characters, the characters that correspond to them should be included (in order) to make the thirteen-character set:

12. upper-half block
13. lower-half block

On a few terminals, the graphics-on and graphics-off sequences are treated as character attributes. In particular, turning off graphics also sets the terminal to its default video attributes. If this is the case, then the code GA (graphics are attributes) should be included in the terminal description. A few terminals also cannot move the cursor while in graphics mode. If this is the case, the code GX (graphics movement glitch) should be included.

Some terminals do not need to send a graphics-on or a graphics-off sequence. For these terminals, the line-drawing characters are available in the default character set. If this applies to your terminal, then just give the GM setting without the GO or GF settings.

4.6.6.1 Multi-character sequences for graphics

Some terminals require more than one character in the escape sequence that draws a graphical line segment. For example, the two-character sequence “E\202” might be required to draw a single horizontal line character.

ACUCOBOL-GT permits up to three characters to be specified in an escape sequence that draws a single line segment. The three characters are stored separately and “assembled” into a single sequence by the Terminal Manager.

When these multiple-character sequences are used, the GO (graphics on) and GF (graphics off) codes serve special purposes. GO is used to store the first character in the sequence, and GF is used (if needed) to store the third character.

You tell the runtime (by including the GE code) that GO should be sent to the terminal *before* each GM graphical character that is sent, and GF should be sent *after* each GM graphical character.

Also you must make sure that the GM character list contains the appropriate characters. To handle the example mentioned above, in which a horizontal line segment requires the two-character sequence “\E\202”, you would add two codes to the terminal database entry: “:GE:” and “:GO=\E:”, and also add “\202” to the GM character list in position one (horizontal line character).

Some VT-100 emulators support line drawing by using alternate character sets. They turn on graphics by sending “ESC(0)” and turn it off by sending “ESC(B)”. The entry is:

```
vt100|vt-100|DEC VT-100 :\
:cl=\E[;H\E[2J:ce=\E[K:cd=\E[J:\
:cm=\E[%i%d;%dH:nd=\E[C:\
:LO=\E[m:HI=\E[0;1m:RV=\E[7m:\
:UL=\E[4m:BL=\E[5m:RU=\E[4;7m:\
:RB=\E[5;7m:DL:k1=\EOP:k2=\EOQ:\
:k3=\EOR:k4=\EOS:ku=\E[A:\
:kd=\E[B:kr=\E[C:k1=\E[D:\
:GO=\E(0:GF=\E(B:GM=qx.lkmjvtwun:
```

4.6.7 Graphical Window and Control Emulation

The character-based version of the runtime emulates graphical windows and controls by displaying characters with particular attributes that approximate the look and feel of a graphical system.

The following standard characters are used by default to represent various graphical components:

“-”, “|”, “+”, “_”, “*”, “.”,

“^”, “v”, “<”, “>”, ““”, “#”

Terminals that support a *line drawing* set or a special *extended* character set, or both, can be configured to use these special characters. The configuration method is similar to the one used for line drawing, described in [section 4.6.6, “Line Drawing.”](#)

To support the substitution of line drawing characters and extended characters, there are two terminal database (“a_termcap”) functions: GO-GUI-MAP and GF-GUI-MAP.

The **GO-GUI-MAP** function uses a list of standard characters that correspond to line segments and other special characters when displayed in the terminal’s *graphics* mode. This is similar to the GM function used for line drawing. The Terminal Manager turns the terminal’s *graphics* mode on by sending the GO code. The GO code is followed by normal characters which are interpreted by the terminal into their corresponding special characters. When all of the special character elements have been displayed, the terminal is set back to *normal* mode with GF.

The **GF-GUI-MAP** function provides a method for specifying substitute *standard* characters for some or all of the graphical components. When these characters are used, they are displayed in normal (not *graphics*) mode. The Terminal Manager gives preference to the characters specified in GO-GUI-MAP. If a character in the list is preceded with “\0” (backslash, zero), the Terminal Manager uses the corresponding character in the GF-GUI-MAP. If the character cannot be determined from the GO-GUI-MAP or GF-GUI-MAP functions (either list may be incomplete or there may be a “\0” in the same position in both lists), the Terminal Manager uses the default character. For more information about defining the list of special characters, see the entry for GUI_CHARS in Book 4, *Appendices*, Appendix H.

The characters listed in the GO-GUI-MAP and GF-GUI-MAP correspond, in order, to the following graphical components. The character in parentheses is the default character:

1. System menu button (*)
2. Floating window title left corner (+)
3. Floating window title right corner (+)
4. Floating window title fill character (=)

- | | | |
|-----|---|-----|
| 5. | Minimizer | (.) |
| 6. | Maximizer | (^) |
| 7. | Scroll bar up button | (^) |
| 8. | Scroll bar down button | (v) |
| 9. | Scroll bar left button | (<) |
| 10. | Scroll bar right button | (>) |
| 11. | Scroll bar page area | () |
| 12. | Scroll bar thumb | (#) |
| 13. | Left entry field box and check box character | (l) |
| 14. | Right entry field box and check box character | (r) |
-

Note: Some of these graphic components may not be used in the current version of ACUCOBOL-GT.

When a program executes, the runtime evaluates the terminal's display capabilities and determines the special display attributes to apply to select control elements. These control elements include the control's *key letter*, push-button text, and a key letter that is part of the push-button text when the user presses the button with the mouse. The runtime applies the first supported capability to each element as follows.

Key letter:

1. Underline
2. Intensity toggle (opposite intensity - for example, if the control is displayed in high intensity, the key letter is displayed in low intensity)
3. Reverse video

Selected push-button text:

1. Reverse video
2. Underline
3. Intensity toggle

Key letter in selected push-button text:

If the selected push-button text attribute is *reverse video*, apply to the key letter:

1. Intensity toggle
2. Underline
3. Reverse video (the key letter is indistinguishable from the other text)

If the selected text attribute is *underline*, apply to the key letter:

1. Intensity toggle
2. Underline (the key letter is indistinguishable from the other text)

If the selected text attribute is *intensity toggle*, apply to the key letter:

Intensity toggle (key letter is indistinguishable from the other text)

Reconstructing the screen

On a character-based system, during program execution when a control is resized, moved, hidden, or removed (destroyed), the runtime applies the following procedure to reconstruct and display the screen:

1. The screen is reconstructed in memory, in a *virtual screen*, before being displayed to the physical screen.
2. The portion of the screen *underneath* the affected control is redrawn to the virtual screen with the attributes and colors of the owning window (this usually results in that area of the screen being filled with the owning window's background color).
3. Any controls that overlap the affected area are redrawn in the order in which they were originally created.
4. The changed portions of the screen (constructed in memory) are displayed to the physical screen.

4.6.8 Mouse Support for X Terminals

The Terminal Manager allows for limited mouse support for X terminals if you are using a curses-compatible mouse. To make mouse events available to your COBOL program, you need to do the following to your termcap file:

- use an escape sequence in the “is” termcap entry to enable mouse events
- use an escape sequence in the “DI” termcap entry to disable mouse events at exit, and
- create a new entry, “km”, which is the lead-in sequence for a mouse event. When the escape sequence for “km” is detected, the next three characters are the event and character position of the mouse at the time of the event.

Currently, the support is limited. In particular, the termcap file will return information about which button was pressed, and where the mouse was at the time the button was pressed. Though it will return information when a button was released, it cannot tell which button was released. The runtime assumes that the button last pressed is the button released. This assumption may, of course, be incorrect. Double-clicks and information about motion are never returned.

The `a_termcap` entry for “xterm-mouse” is:

```
xterm-mouse|xterm emulator with mouse support (X window system):\
:cr=^M:do=^J:nl=^J:bl=^G:le=^H:ho=\E[H:\
:co#80:li#56:c1=\E[H\E[2J:bs:am:cm=\E[%i%d;%dH:nd=\E[C:up=\E[A:\
:ce=\E[K:cd=\E[J:UL=\E[4m:DL:\
:HI=\E[1m:RV=\E[7m:LO=\E[m:\
:ku=\EOA:kd=\EOB:kr=\EOC:kl=\EOD:kb=^H:\
:k1=\E[11~:k2=\E[12~:k3=\E[13~:k4=\E[14~:\
:k5=\E[15~:k6=\E[17~:k7=\E[18~:k8=\E[19~:\
:k9=\E[20~:k0=\E[21~:ta=^I:pt:sf=\n:sr=\EM:\
:a1=\E[L:dl=\E[M:ic=\E[@:dc=\E[P:\
:kh=\EO\000:kN=\E[6~:kP=\E[5~:\
:km=\E[M:\
:w8=\E[?31:w3=\E[?3h:\
:ks=\E[?1h\E=:ke=\E[?1l\E>:\
:is=\E7\E[?47h\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;6l\E[?1h\E=\E[?1000h:\
:DI=\E[2J\E[?47l\E8\E[?1000l:\
:DI=\E[2J\E[?47l\E8:NS:\
:KX=\177:KI=\E[2~:\
:GO=\E(0:GF=\E(B:GM=qx1kmjvtwun:\
```

```
:w8=\E[?31:w3=\E[?3h:\
:hs:ts=\E[?E\E[?%i%dT:fs=\E[?F:es:ds=\E[?E:
```

4.6.9 Initialization

Initialization strings can be sent at the beginning of the session to ensure that the terminal is in the proper state, or to program special function keys. This is primarily used to program function keys and function key labels with application specific information. The codes **is**, **is1**, and **is2** are always sent at the beginning of each session. You can specify a sequence to send at the end of the session with DI.

Numeric mode

With the VT-100, it is useful to set the numeric keypad to numeric mode (as opposed to application mode). This is done by sending “ESC>“. The new entry is then:

```
vt100|vt-100|DEC VT-100:\
:c1=\E[;H\E[2J:ce=\E[K:cd=\E[J:\
:cm=\E[%i%d;%dH:nd=\E[C:\
:LO=\E[m:HI=\E[0;1m:RV=\E[7m:\
:UL=\E[4m:BL=\E[5m:RU=\E[4;7m:\
:RB=\E[5;7m:DL:k1=\EOP:k2=\EOQ:\
:k3=\EOR:k4=\EOS:ku=\E[A:\
:kd=\E[B:kr=\E[C:kl=\E[D:\
:GO=\E(0:GF=\E(B:GM=qx1kmjvtwun:\
:is=\E>:
```

4.6.10 Print Functions

The Terminal Manager also allows for limited support of a printer attached directly to the terminal. You must ensure that communications between the printer and terminal meet all the restrictions that the devices require (some terminals, for example, require that the printer and the terminal run at the same baud rate). The Terminal Manager supports two printer functions. The Enable-print function (EP) causes data sent to the terminal to be also sent to the printer; EP mode remains in effect until turned off by the Disable-print function (DP).

Pass through mode

The VT-100 supports an attached printer. It has several print modes, but the only one that the Terminal Manager supports is the “pass through” mode where data sent to the terminal is passed through to the printer. This is enabled by sending “ESC[5i” and disabled by sending “ESC[4i”. The completed VT-100 entry is:

```
vt100|vt-100|DEC VT-100:\
:cl=\E[;H\E[2J:\
:ce=\E[K:cm=\E[%i%d;%dH:cd=\E[J:\
:nd=\E[C:LO=\E[m:HI=\E[0;1m:RV=\E[7m:\
:UL=\E[4m:BL=\E[5m:RU=\E[4;7m:\
:RB=\E[5;7m:DL:k1=\EOP:k2=\EOQ:\
:k3=\EOR:k4=\EOS:ku=\E[A:\
:kd=\E[B:kr=\E[C:kl=\E[D:GO=\E(0:\
:GF=\E(B:GM=qxlkmjvtwun:is=\E>:\
:EP=\E[5i:DP=\E[4i:
```

4.6.11 Continued Entries

The **tc** function allows you to include, by reference, all the functions from another terminal database file entry. The syntax is **tc=entry**, where *entry* is the name of the database file entry whose functions are to be included.

For purposes of explanation, let us say the terminal database file entry you are working on is called “entryA” and the one you wish to reference is called “entryB”. Then, as the last function in entryA, you would write **tc=entryB**. This will include all the functions from entryB in entryA. If there are conflicts between the functions specified by entryA and entryB, the entryA functions take precedence.

Also, any function in entryB can be “turned off” by naming it, followed by an @ sign, in entryA. For example, if there is an AL function in entryB and you wish to turn it off, simply say :AL@: in entryA.

For example, some VT-100s come with an “Advanced Editing” option that includes, among other things, an add-line and delete-line function. The complete entry for this might be:

```
vt100a|DEC VT-100 w/Advanced Editing:\
:al=\E[L:dl=\E[M:tc=vt100:
```

5

File Processing

Key Topics

Transaction Management	5-2
AcuServer	5-16
XFD Files	5-19
International Character Handling	5-62

This chapter describes several special file processing issues. Included are:

- transaction management
- client/server implementation with remote file access services
- generation of data dictionaries (also known as extended file descriptors or XFDs)
- translation of international character sets between machines that use different codes for the same characters

General file handling capabilities are covered in **section 6.1, “Handling Files.”**

5.1 Transaction Management

ACUCOBOL-GT extends standard COBOL to provide a complete *transaction management* system. A *transaction* is a group of related file operations that are treated as an indivisible unit. The purpose of defining such transactions is to ensure that related files can be restored to a consistent state when errors occur.

The transaction management requirements of business applications vary widely and can be quite complex. ACUCOBOL-GT provides a basic transaction management facility (described in detail in the sections that follow). Specialized transaction management middleware frequently offers sophisticated capabilities not included in ACUCOBOL-GT, such as multi phase commit, nested, concurrent and distributed transactions, transaction queuing, automatic rollback on startup, and dynamic load balancing, to name a few. Before selecting a transaction management facility, you should carefully assess the needs of your business systems.

Note: ACUCOBOL-GT's transaction management facility can sometimes interfere with programs that use a separate transaction management facility, such as an *online transaction processing* (OLTP) facility. To disable ACUCOBOL-GT's transaction management, set the **NO_TRANSACTIONS** configuration variable (see Appendix H of Book 4). For information about using ACUCOBOL-GT with OLTP systems, see Chapter 9 in *A Guide to Interoperating with ACUCOBOL-GT*.

5.1.1 Overview of Transaction Management

Standard COBOL contains a weakness in its handling of file operations in that it does not provide a method of defining transactions. This section describes the special features of ACUCOBOL-GT that allow the security of transaction management.

The benefits of a transaction management system are best illustrated by an example. A COBOL application that handles order entry might perform these steps to accept an order:

1. Write an invoice record.

2. Update a customer record.
3. Write a payroll record for sales commissions.
4. Update an inventory record.

This series of four file operations is a logical unit. If the program were interrupted, and completed only some of the four file operations, then the files would be in an inconsistent state. For example, if the program died after it updated the customer record, but before it updated the inventory record, then a subsequent run might access non-existent inventory.

The solution to this problem is to provide a method where the programmer can define a set of operations that should either all occur or all not occur. Then, if the program encounters an error or dies, the files are left in a consistent state.

Note: Sites using transaction management should ensure that all users are referring to the same files by the same names. For example, all users should map the same network directory to the same drive letter.

5.1.1.1 Transaction logging

ACUCOBOL-GT solves this problem by providing a transaction logging facility. All file operations that are part of a transaction are logged. Once logged, they can either be committed or rolled back (undone) by the program.

If a program dies, or the system fails, the log file can be used to reconstruct complete transactions, thus returning all files to a consistent state.

Transaction logging thus offers these two facilities:

- It provides the programmer with the ability to define transactions, and the ability to commit them or “undo” them (usually in response to an error condition). This “undo” facility is called a “rollback.”
- It provides the ability to reconstruct files into a consistent state after a program dies or system failure occurs. This operation is called “recovery.”

5.1.1.2 File types

In general, transactions apply to relative and indexed files. Transaction operations apply for indexed Vision, C-ISAM, and BTRIEVE files, as well as for Acu4GL database files (which are treated as indexed files).

Each transaction operation is executed on the file systems linked with the ACUCOBOL-GT runtime. Each file system uses its own mechanism to handle the transaction operations, and some have their own log files. The general rules for the Vision file system and for relative files are listed in the ACUCOBOL-GT *Reference Manual* under the COBOL verbs that are used with transaction management. If you use an alternate file system that does not support transaction management, the facility is not available for programs that access those files.

5.1.1.3 Features

Transaction logging provides the following:

- A method of indicating that transaction logging is wanted and identifying which log files to use.
- A means of deciding which file operations to log and which files have rollback capability.
- The ability to start, commit, and rollback transactions in COBOL.
- A utility routine to perform the recovery function.
- A utility program to examine and edit log files.

5.1.2 The Transaction Logging Process

Transaction logging provides the programmer with a COBOL verb that can roll back or “undo” file operations performed during a transaction. A rollback is typically invoked when the program detects a file error or other error condition.

Outside of the application, transaction log files can be used for another purpose: to recover files after a program failure or system crash. The recovery process is performed by an ACUCOBOL-GT library routine that accesses the latest backup and the log files. It is the site's responsibility to clear the log files when a backup is made, so that the logs always contain precisely those transactions that have occurred since the last backup. The logs and the backup can then be used to reconstruct files into a consistent state after a crash.

To make rollback and recovery operations possible, the runtime records information about how to “redo” and “undo” each file update that occurs within a transaction. After the transaction starts, each Vision or relative file update operation is recorded until the transaction is committed or rolled back. At that point:

- If the transaction is committed, the commit operation writes the “redo” information to the log file. The “undo” information is not recorded in the log file.
- If the transaction is rolled back, the rollback operation scans the record of changes in reverse order and performs the operations needed to undo the file updates done during the transaction.

All programs that operate on the same set of data files must also use the same set of transaction log files.

In transaction management, the log files must be synchronized to the backup procedure for the files being managed. When the files are backed up, the log files must be cleared (or deleted). The next time a transaction occurs, the runtime creates new log files (or begins writing to the cleared files). Thus, the logs contain precisely those transactions that have occurred since the last backup.

If the log files and backups are synchronized, when a program dies in mid-transaction, you can reconstruct the files into a consistent state. To do this, you restore the backup, then run a program that calls the C\$RECOVER utility routine (see [section 5.1.10](#)). This routine reads a log file and performs each update operation that was part of a completed transaction. Operations that were not part of a completed transaction are not performed.

By using transaction logging, you ensure logically consistent files. In addition, you never need to do a rebuild operation on a broken data file, because you can always recover files by restoring the backup and running a program that calls C\$RECOVER.

Note: You can use the utility program **logutil** to examine and edit an ACUCOBOL-GT transaction log file. The **logutil** utility supports large (>2GB) transaction log files. The host operating system must allow such large files. For UNIX, the “**USE_LARGE_FILE_API**” configuration variable must be set. For details, see **section 3.6, “logutil.”**

Though a transaction logging system provides a high degree of file security, it does impose extra overhead. Backups are essential, and there is increased disk I/O, more memory use, and more record locks.

Going to your backups and using the recovery procedure usually takes more time than just checking the data files and restarting the program. On the other hand, if the system dies, you do not need to worry about examining the files for problems.

5.1.3 Transaction Management Verbs

There are three verbs used in ACUCOBOL-GT transaction management. These are:

START TRANSACTION

COMMIT TRANSACTION

ROLLBACK TRANSACTION

Note: TRANSACTION, ROLLBACK, START, and COMMIT are reserved words.

START TRANSACTION identifies the beginning of a transaction. The COMMIT TRANSACTION statement indicates the end of a transaction and commits the changes made. The ROLLBACK verb causes the transaction to be rolled back or “canceled.”

There is an implicit COMMIT before a STOP RUN or before the end of the program. However, the runtime performs an implicit ROLLBACK before a STOP RUN if the STOP-RUN-ROLLBACK configuration variable is set. If the runtime system is killed by the user or encounters a fatal error prior to completing a transaction, then an automatic rollback occurs. For more information on **START Statement**, **COMMIT Statement**, and **ROLLBACK Statement**, see their respective entries in Book 3, *Reference Manual*, section 6.6.

5.1.4 Extended Locking Rules

In order to prevent another process from updating the records in a way that might make it impossible to do a rollback, the system locks any records modified during a transaction. This is done only for files that allow rollbacks. ACUCOBOL-GT offers some flexibility in record locking procedures in its transaction management system. You can enable multiple locking rules or single locking rules as the default locking mode for files that allow rollbacks.

To indicate which files will have record locks held, use the following syntax in the LOCK MODE phrase of the FILE-CONTROL entry for each affected file:

```
WITH ROLLBACK
```

When rollback is enabled, record and file locking rules are extended. Every record updated as part of a transaction is locked until that transaction is committed or rolled back. The COMMIT and ROLLBACK verbs remove these locks. Record locks that are applied when the file is read are also kept until the end of the transaction. Multiple record locking rules are in effect by default.

Use of the compiler option, “-FI”, enables single locking rules rather than multiple locking rules as the lock mode default. Normally, “WITH ROLLBACK” causes multiple locking rules to be in effect for a file. When “-FI” is used, the “WITH ROLLBACK” clause does not affect whether single or multiple record locking rules are followed. Single locking becomes the default. You may enable multiple locking either by specifying “WITH LOCK ON MULTIPLE RECORDS” in a file’s SELECT statement or by using “APPLY LOCK-HOLDING ON *file*” in the I-O CONTROL paragraph.

Record locks are held during a transaction in order to prevent another process from updating the records in a way which might make rollback impossible. Note, however, that a record may be deleted during a transaction, and another process is allowed to write a record with the same record key to the file. If this happens, the ROLLBACK will fail with a duplicate key error. See **section 5.1.5, “Logging and Rollback of File Update Operations.”**

Note: During a transaction, the UNLOCK statement affects only files for which rollback is not enabled. In the case where the UNLOCK statement is ineffective because rollback was enabled for the file, the file status will be set to 00 (success).

During a transaction with a Vision or relative file, a CLOSE is postponed if any updates (WRITE, REWRITE, or DELETE) are performed on the file during the transaction, *regardless of whether any records are locked*. This function permits the file to remain open for ROLLBACK.

If the same physical file is OPENed again *within the same transaction*, even if the program is using a different logical file (different SELECT), the postponed CLOSE is canceled. Note that the mode of the original OPEN is retained. (For example, if the file were originally OPEN OUTPUT, and if the CLOSE were canceled, then an OPEN I-O on the same file within the same transaction would *not* enable the program to read the file.) When the second OPEN is encountered, the file position is reset to the beginning so that a READ NEXT would read the first file in the record.

Caution: While a CLOSE action is delayed until the transaction is committed, a DELETE FILE action is performed immediately. If the DELETE FILE statement is contained within a transaction (before the COMMIT) following its associated CLOSE statement, the result will not be what is expected. Because the CLOSE is delayed but the DELETE FILE is executed right away, the program will attempt to delete the open file. The result of the attempted DELETE FILE action is host system dependent (for example, under Microsoft Windows, attempting to delete an open file will cause an error 37, 07 to be returned, while under UNIX, where there is no restriction on deleting an open file, the action will succeed). The safest programming practice is to not code a DELETE FILE action inside a transaction.

5.1.4.1 Special handling of implicit transactions

There are conditions in which transactions are automatically defined by the runtime to ensure that all file operations are logged, such as when the “-Ft” compiler flag is used (see [section 2.2.7, “File Options”](#)) or when an OPEN or CLOSE is performed outside of a defined transaction on a file that has *rollback* enabled. In these situations, the implicit COMMIT does not unlock any records associated with the file. This ensures that the runtime defined transaction is recorded without interfering with the existing program logic (or locking).

5.1.5 Logging and Rollback of File Update Operations

The *record* update operations WRITE, REWRITE, and DELETE can be rolled back. However, *file* operations which open, create, recreate, rename, delete, or close files cannot be rolled back. Therefore, OPEN, CLOSE, DELETE FILE, RENAME, and COPY operations cannot be rolled back.

When a transaction is committed, all of the *file* update operations are written to the default log file defined by the LOG-FILE configuration variable. However, if instead the transaction is rolled back, those file operations that cannot be rolled back are written to the default log file and then committed.

If a file that was opened during a transaction is closed outside of a transaction, the CLOSE will be treated as its own transaction and will be logged. If a file that has *rollback* enabled (using the WITH ROLLBACK syntax; see [section 5.1.4](#)) is opened outside of a transaction, the OPEN will be treated as its own transaction and will be logged.

If, during a transaction, a record is deleted from a file not allowing duplicates on a particular key, the delete may be rolled back only if the record key does not exist in the file at the time of the rollback.

This means that if another process writes a record with the same key to the file after the delete, then the delete will not be rolled back, and the rollback will fail with a duplicate key error. (See [section 5.1.8, “Transaction Error Handling.”](#))

5.1.6 Multiple Log Files

The transaction management facility allows the program to use multiple log files. Different types of transactions can be logged in separate log files, giving you quicker access to a particular type of transaction. Each log file records the information that is necessary to recover a particular file or set of files.

The application administrator or programmer can specify an individual log file to use with any given set of data files. The specified log file is updated whenever a transaction is committed in its associated data files. Log file names are specified with configuration variables of the format:

```
filename-LOG logfilename
```

where *filename* is the base name of the data file, and *logfilename* is the name of the log file. *filename* should not include any directory names, nor should it include a file extension. *logfilename* can include the absolute or relative directory path ending with the name of the log file. If the log file is not found, a new file is created there with the specified name. Note that *logfilename* can have remote name notation. A configuration file entry for multiple log files might look like:

```
#transaction temporary file directory
LOG-DIR /usr/transaction-tmp/
#log file definitions
file1-LOG file1.log
file2-LOG file2.log
file3-LOG file3.log
#default log file definition
LOG-FILE default.log
```

In the example above, during a commit, all of file1's updates are written to "file1.log", all of file2's updates are written to "file2.log", and so forth. If other data files are updated during a transaction, their updates are written to "default.log". Operations performed with the "**RENAME**" or "**CSCOPY**" (Appendices manual, section H) library routines are also written to "default.log".

A default log file must be specified in the LOG-FILE variable. The runtime creates that log file, or opens the existing one, as part of the first START TRANSACTION statement. Log files specified with *filename-LOG* variables are created or opened when the file whose base name is *filename* is

opened OUTPUT or I/O for the first time in the program. Note that this means that any error that can be returned from a START TRANSACTION can also be returned as a secondary code of an error 9E on an OPEN statement. See **section 5.1.8, “Transaction Error Handling.”**

Multiple log files can be used only with Vision indexed files and relative files. Transaction management for other file systems is dependent on the specific file system’s transaction management facility.

5.1.7 Configuration Variables

There are seven configuration file variables used to configure the logging system. For details, see the configuration variable’s listing in Appendix H:

- **filename_LOG** - specify the name of a log file
- **LOGGING** - enable or disable logging
- **LOG_FILE** - specify the name of the default log file
- **LOG_BUFFER_SIZE** - set the log file buffer size
- **LOG_DIR** - specify an alternate directory for temporary files
- **LOG_DEVICE** - enable or disable special device handling for the log file
- **LOG_ENCRYPTION** - enable or disable log encryption

5.1.8 Transaction Error Handling

Error codes associated with transaction management are stored in a special register called TRANSACTION-STATUS. These codes tell you the status of the last transaction and are documented in Appendix E.4. Transaction management errors fall into two categories:

1. For errors that occur during a START TRANSACTION, COMMIT, ROLLBACK, or call to C\$RECOVER (see **section 5.1.10, “Recovery”**), use TRANSACTION-STATUS to determine the type of error that occurred.
2. After the execution of any file operation during a transaction, the file’s FILE-STATUS variable will contain 9E if an error occurred in the transaction system. The exact nature of the error will be shown by the contents of TRANSACTION-STATUS.

A subcategory of these errors are “intermediate” runtime errors that call installed error procedures. They are:

- “File error #”
- “File error # on #”
- “Transaction error #”
- “Transaction error # on #”

where the # signs are replaced at run time by error names, numbers, or other information. See Book 4, Appendix I, “Library Routines,” for detailed discussion of the error and exit procedures.

The TRANSACTION-STATUS variable has the same format as a file’s status variable. It is automatically created by the compiler, and is implicitly shared by all programs of a run unit. TRANSACTION-STATUS is a reserved word.

You can specify procedures for transaction error handling with the USE statement and the reserved word TRANSACTION. The syntax is:

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE ON TRANSACTION  
                                  {ERROR          }
```

If TRANSACTION is specified, the procedure executes when an error occurs during a START TRANSACTION, COMMIT, ROLLBACK, or call to C\$RECOVER. See **section 5.1.10, “Recovery.”**

Note: A transaction error “10” is returned by a START TRANSACTION statement when the LOG-FILE configuration variable (that specifies the default log file) is undefined. The COBOL program may choose to ignore this error in the cases where the *filename*-LOG variables are used.

5.1.9 Compiler File Options

There are three compiler options that can simplify the addition of transaction management facilities to existing programs that use the Vision file system. The “-Ft” option causes implied transactions for every OPEN, CLOSE, WRITE, REWRITE, or DELETE operation that is not part of an explicit transaction. In other words, single file operations that are not part of a transaction are preceded by an implied START TRANSACTION and followed by an implied COMMIT. This makes it easier to convert existing applications to a transaction system.

The “-Fs” option causes an implied START TRANSACTION verb before the first OPEN, CLOSE, WRITE, REWRITE, or DELETE operation and after each COMMIT or ROLLBACK. This, in effect, makes every file operation part of a transaction. If this option is enabled, and the compiler encounters a START TRANSACTION verb, it reports a warning and does not generate any code for the START TRANSACTION. The “-Fs” option provides an alternate way to program transactions and is often useful when you are converting from other COBOL or SQL implementations.

A third compiler file option, “-Fl”, enables single locking rules rather than multiple locking rules as the lock mode default. Normally, “WITH ROLLBACK” causes multiple locking rules to be in effect for a file. When “-Fl” is used, the “WITH ROLLBACK” clause does not affect whether single or multiple record locking rules are followed. Single locking becomes the default. You may enable multiple locking either by specifying “WITH LOCK ON MULTIPLE RECORDS” in a file’s SELECT statement or by using “APPLY LOCK-HOLDING ON *file*” in the I-O CONTROL paragraph.

5.1.10 Recovery

If a program dies, or there is some other system failure, files may become corrupted, destroyed, or left in an inconsistent state. If this occurs, the files may be recovered with the built-in library routine, C\$RECOVER.

Note: For any Vision files or relative files, C\$RECOVER can be successful only if every operation on the files, including file OPEN, was done within a transaction.

To use the recovery function, you need the most recent backup copy of all the files that record their transactions. You also need all of the log files that were started immediately after the backup was created. Here's one way to perform a recovery:

1. Get everyone off the system.
2. Restore backup files to original locations.
3. Run a program that calls the **C\$RECOVER** library routine (for example, CALL "C\$RECOVER"). Use the same configuration file that the applications use. Note that the log files specified with the *filename*-LOG variables are not used.

The C\$RECOVER library routine only recovers the updates recorded in the log file specified by the configuration variable LOG-FILE. To recover updates recorded in multiple log files, specify each log file in the LOG-FILE variable and call C\$RECOVER for each one. For example:

```
SET CONFIGURATION "LOG-FILE" TO "default.log".
CALL "C$RECOVER".
SET CONFIGURATION "LOG-FILE" TO "file1.log".
CALL "C$RECOVER".
SET CONFIGURATION "LOG-FILE" TO "file2.log".
CALL "C$RECOVER".
SET CONFIGURATION "LOG-FILE" TO "file3.log".
CALL "C$RECOVER".
```

To ensure that your data can be recovered, you should:

1. Make sure all transactions are committed before you do a backup.

2. Clear or delete the log files only after your backup is complete.
3. Make sure there is *no file activity* between the time the backup is made and the time you clear or delete the log files.
4. In between backups, you can periodically archive log files, then clear or delete them.
5. If you use archived log files, make sure you recover using the logs in the chronological order in which they were written.
6. Make sure there is *no other activity* on the files during recovery.
7. Make backups and start new log files immediately if any log file gets corrupted or destroyed.
8. Make sure you have good log files.

The backup data files should have the same absolute or relative pathname as when the original programs were run. If there are any cases where relative pathnames are used for the log or data files, run the program that calls the C\$RECOVER routine from the same directory as the original programs. For the program that calls the C\$RECOVER routine, specify the log file in the configuration file or in the environment.

If the recovery process terminates prematurely due to an error, and files are left open, they are automatically closed before the call to C\$RECOVER returns.

If a failure occurs during the recovery process, and no other copies of the backup files are available, then automatic recovery is not possible. This means that you should make sure to have an extra backup copy of your data files.

5.1.10.1 Transaction logging and recovery with AcuServer

Transaction logging and recovery are possible on a network using AcuServer (see **Section 5.2, “AcuServer”**). Transaction logging and recovery of operations on remote files are also supported.

Such recovery is possible only if each client runtime uses the same log file. To accomplish this, specify a host name, according to AcuServer rules, in each client's runtime configuration file. For example, to specify a single log file on a server with the host name "sun", add the following line to each client runtime's configuration file:

```
LOG-FILE @sun:/usr2/users/cobol/mylogfile
```

Each client runtime with this configuration file entry will use the single server log file during its START TRANSACTION, ROLLBACK, and COMMIT processing. Temporary files used for rollback are created in the working directory, or in the directory specified by the LOG-DIR configuration variable of the client runtime.

5.2 AcuServer

AcuServer is an add-on module that provides remote file access services to ACUCOBOL-GT applications. AcuServer is available for applications running on UNIX, Linux, and Windows TCP/IP based networks and executing with ACUCOBOL-GT runtime Version 5.0 or higher.

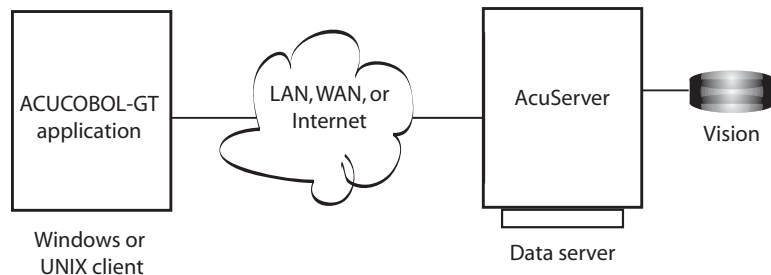
With AcuServer, your applications gain:

- the ability to create and store data files on any UNIX or Windows NT/Windows 2000 server equipped with AcuServer
- full function remote access from UNIX, Linux, and Windows clients to all Vision, relative, sequential and object files stored on an AcuServer server
- full record locking support of all Vision and relative files
- transparent access of remote and local files

AcuServer does not require any changes to your existing application code. (Some applications that contain hard coded paths to files must be modified to use the **FILE_PREFIX** environment variable, or to include the name of the file server in the path.)

AcuServer does not require that you recompile your existing programs. Programs compiled with any version of ACUCOBOL-GT can be executed ACUCOBOL-GT runtime version 5.0 or higher. Contact your Micro Focus **extend** Sales Professional for a current list of supported platforms.

The *AcuServer User's Guide* contains installation instructions and a complete description of each AcuServer feature.



UNIX or Windows NT file server running AcuServer

Above is an illustration of a simple AcuServer network. ACUCOBOL-GT applications running on UNIX and PC client systems access and store data files on a common UNIX or Windows NT/Windows 2000 file server running AcuServer. More complex networks might incorporate multiple AcuServer file servers supporting dozens of client machines running multiple applications.

5.2.1 System Requirements

To use AcuServer:

- Each server machine must be networked to UNIX, Linux, and Windows clients with TCP/IP. (TCP/IP is not sold or supplied by Micro Focus.)
- All servers must have a copy of the AcuServer license management file.
- Windows clients can run any TCP/IP software that uses a WINSOCK compliant WSOCK32.DLL.

- Client machines must have an ACUCOBOL-GT AcuServer-enabled runtime. All Windows runtimes Version 5.0 and later are AcuServer-enabled. To verify that your UNIX runtime is AcuServer-enabled, type “`runcbl -v`” and look for the line:

```
AcuServer client
```

On SCO UNIX and NCR Tower systems, you will also need their optional NFS software package.

5.2.2 Remote Name Notation

Accessing remote files requires that your application refer to them with *remote name notation*. The ACUCOBOL-GT runtime looks for remote name notation to identify requests to AcuServer. Remote name notation has the following format:

```
@server-name:path-name
```

I/O requests to files prepended with “*@server-name:path-name*” are routed to AcuServer on the host specified by *server-name*. AcuServer looks for the named file in the directory specified by *path-name*, for example:

```
@condor:/usr/acct/inventory
```

To add a remote path to **FILE_PREFIX** or **CODE_PREFIX**, use the format:

```
FILE_PREFIX @server-name:path-name  
CODE_PREFIX @server-name:path-name
```

As an alternative to **FILE_PREFIX**, you can define file name aliases in the runtime configuration file. A file name alias is a string that will replace the literal name in the **ASSIGN TO** clause of a **SELECT** statement. For example:

```
input-output section.  
file-control.  
    select idx-file  
    assign to disk "IDXDAT"  
    binary sequential  
    status is idx-status.
```


To define an alias for “IDXDAT” you could add the following line to your runtime configuration file:

```
IDXDAT @condor:/usr/data/index_data
```

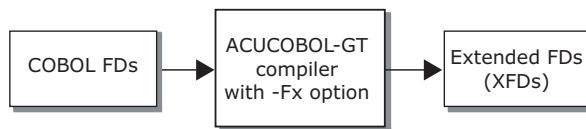
5.3 XFD Files

The ACUCOBOL-GT compiler is capable of generating *data dictionaries* that store a map of COBOL record structures. These dictionaries are also called *eXtended File Descriptors* (XFDs) because they’re based on the standard COBOL file descriptors (FDs).

XFDs are used by Acu4GL to interface to database management systems and by AcuXML to interact with data in XML files. They are also used by the **alfred**, indexed record editor, and by AcuXDBC, database management system for Vision. In addition, they are used to help guide the mapping of international character sets between machines that use different underlying character codes.

The compiler creates an XFD for each indexed file in the compiled program when you specify the “-Fx” compile-time option. The “-Fa” option generates XFD files for all indexed, relative, and sequential files in the compiled program. Ordinarily, XFDs are generated in XML format. Use the “-Fe” option to generate XFD files in flat text format instead. See **Chapter 2, section 2.2.7, “File Options”** for details on these compiler options.

Acu4GL, AcuXDBC, and **alfred** can all read the XFD files in either format. XFDs are fully portable, and thus no recompilation is necessary if you change hardware.



The effects of all compile-time options, COPY REPLACING, and source-code control lines are reflected correctly in the XFDs.

Understanding how the XFD file is formed

Micro Focus's data dictionaries (XFDs) enable the Acu4GL and AcuXML interfaces to create a table (or access an existing one) in a database for each indexed, relative, or sequential file. Each column in the table contains the values for one field. The column names are essentially the field names.

XFDs also allow the **alfred** record editor to display and accept data in an indexed file at the field level, instead of character-by-character. If an XFD is available to **alfred**, it has enough information to display fields on the screen, instead of just groups of characters. This makes the screen easier to read, and helps make editing more efficient.

Each XFD file is *based on the largest record in the COBOL file*, and contains the fields from that record, plus any key fields (key fields are those which are named in KEY IS phrases of SELECT verbs in the File-Control section). This ensures that data from all COBOL records will fit within database tables, and simplifies the storage and retrieval process. If you were to examine the XFD file, only the fields from the largest record, and the key fields, would appear.

Note: If the field named in the KEY IS phrase is a group item, it will not be named as a field in the XFD file. This means it will *not* be used as a column in a database table, if you are using an Acu4GL interface, and it will *not* appear on the screen if you use the **alfred** editor. Instead, all elementary items subordinate to the named group item will be named in the XFD. You can force the group item to be a named field by using the USE GROUP directive (see **Section 5.3.3.11, "USE GROUP directive"**).

With multiple record formats (level 01), not all COBOL fields are named in the XFD file, but all fields *are* storable and retrievable. The data dictionary maps fields from all records of a file to the corresponding locations in the *master* (largest) record of the file. Since Acu4GL and **alfred** have access to the data dictionary, they *know* where the data from a given COBOL record fits. This activity is invisible to the COBOL application.

For example, if your program has one file with the three records shown below, the underlined fields will be included as fields in the XFD file by default (this example assumes that **ar-codes-key** is named in a KEY IS

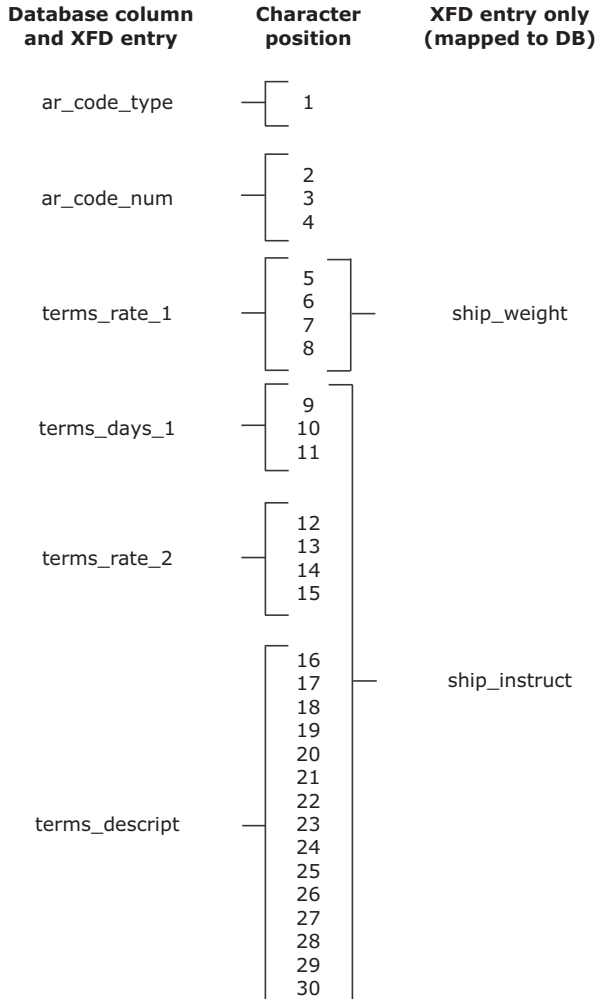
phrase). Some fields will not appear by name in the file, but will be mapped to the *master* field names. The interface thus will eliminate redundancies and give you optimum performance.

```
01 ar-codes-record.
  03 ar-codes-key.
    05 ar-code-type      pic x.
    05 ar-code-num      pic 999.

01 ship-code-record.
  03 filler                pic x(4).
  03 ship-weight          pic s999v9.
  03 ship-instruct       pic x(15).

01 terms-code-record.
  03 filler                pic x(4).
  03 terms-rate-1       pic s9v999.
  03 terms-days-1      pic 9(3).
  03 terms-rate-2       pic s9v999.
  03 terms-descript    pic x(15).
```

The following diagram shows how Acu4GL creates database columns for some of the fields in the COBOL record, while other fields are mapped to those columns by the data dictionary; this means that *all* the fields are accessible to the COBOL program.



The next section describes the rules that the interface follows as it builds the XFD file, and explains how you can override those rules when necessary.

5.3.1 Defaults Used in XFD Files

There are several elements of COBOL that require special handling when data dictionaries are built. These include multiple record definitions, REDEFINES, FILLER, and OCCURS. This section describes how ACUCOBOL-GT handles each of these situations.

Note: In many instances you can override the default behavior described below by placing special comment lines in the FDs of your COBOL code. These comments are called *directives*, and are described in [section 5.3.2](#). For example, the WHEN directive allows you to use multiple definitions for a single set of data by specifying when each definition should be used.

Databases generally do not support the notion of multiple definitions for the same column. (In a similar sense, when you are editing a data file, it makes sense to see and change only one view of the data, rather than multiple views.) As the following paragraphs explain, whenever a COBOL program gives more than one definition for the same data, the compiler makes a choice about which definition to use in the data dictionary. Then it disregards the rest.

5.3.1.1 KEY IS phrase

Fields named in KEY IS phrases of SELECT statements are included as fields in the XFD. Other fields that occupy the same areas as the key fields (by either explicit or implicit redefinition) are not included by name, but are mapped to the key field column names by the data dictionary.

Remember, if the field named in the KEY IS phrase is a group item, it will not become a named field in the XFD unless a USE GROUP directive is used (see “Understanding how the XFD file is formed”).

5.3.1.2 RENAMES clause

You may not use the RENAMES clause when creating data dictionaries. If you do, a file error results and the XFD is not created for that file. Instead of using the RENAMES clause, we recommend that you name the data segments in the usual ACUCOBOL-GT manner. For example:

```
ALTERNATE RECORD KEY K1-CUSTOMER =
```

```
        company-name,  
        company-address2,  
        company-zipcode2  
ALTERNATE RECORD KEY K2-CUSTOMER  
ALTERNATE RECORD KEY K3-CUSTOMER WITH DUPLICATES  
ALTERNATE RECORD KEY K4-CUSTOMER =  
        company-contact,  
        k-customer  
ALTERNATE RECORD KEY K5-CUSTOMER WITH DUPLICATES  
ALTERNATE RECORD KEY K6-CUSTOMER =  
        company-address3,  
        company-zipcode3,  
        company-country  
WITH DUPLICATES
```

5.3.1.3 REDEFINES clause

Fields contained in a redefining item occupy the same positions as the fields being redefined. The compiler needs to select only one of the field definitions to use. The default rule that it follows is to use the fields in the item being redefined as column names; fields that appear subordinate to a REDEFINES clause are mapped to column names by the data dictionary.

5.3.1.4 Multiple record definitions

This same rule extends to multiple record definitions. In COBOL, multiple record definitions are essentially redefinitions of the entire record area. This leads to the same complication that is encountered with REDEFINES: multiple definitions for the same data. So the compiler needs to select one definition to use.

Because the multiple record types can be different sizes, the compiler needs to use the largest one, so that it can cover all of the fields adequately. Thus, the compiler's rule is to use the fields in the largest record defined for the file. If more than one record is of the "largest" size, the compiler uses the first one.

5.3.1.5 Group items

Group items are, by default, never included in a data dictionary for the same reason that REDEFINES are excluded: they result in multiple names for the same data items. You can, however, choose to combine grouped fields into one data item by specifying the USE GROUP directive, described in [section 5.3.3.11, “USE GROUP directive.”](#)

5.3.1.6 FILLER data items

In a COBOL FD, FILLER data items are essentially placeholders. FILLER items are not uniquely named and thus cannot be uniquely referenced. For this reason, they are not placed into the Micro Focus data dictionary. The dictionary maintains the correct mapping of the other fields, and no COBOL record positional information is lost.

Sometimes you need to include a FILLER data item, such as when it occurs as part of a key. In such a case, you could include it under a USE GROUP directive or give it a name of its own with the NAME directive, described in [section 5.3.3.7, “NAME directive.”](#)

5.3.1.7 OCCURS clauses

An OCCURS clause requires special handling, because the Acu4GL runtime system must assign a unique name to each database column. The runtime accomplishes this by appending sequential index numbers to the item named in the OCCURS.

For example, if the following were part of a file’s description:

```
03  employee-table occurs 20 times.
05  employee-number pic 9(3)
```

then these column names would be created in the database table:

```
employee_number_1
employee_number_2
.
.
.
employee_number_10
employee_number_11
.
```

```
employee_number_20
```

Note: The hyphens in the COBOL code are translated to underscores in database field names, and the index number is preceded by an extra underscore.

AcuXDBC users can make use of the SUBTABLE directive to modify how OCCURS clauses are handled. See **Section 5.3.3.10** for information on this directive. The **alfred** record editor shows only the names of the fields, without subscripts or indexes.

5.3.1.8 Summary of dictionary fields

Fields defined with an OCCURS clause are assigned unique sequential names. Fields without names are disregarded.

When multiple fields occupy the same area, the compiler chooses only one of them unless you have a WHEN directive to distinguish them. To choose:

- The compiler preserves fields mentioned in KEY IS phrases.
- It discards group items unless USE GROUP is specified.
- It discards REDEFINES.
- It uses the largest record if there are multiple record definitions.

5.3.1.9 Identical field names

In COBOL you distinguish fields with identical names by qualification. For example, there are two fields named RATE in the following code, but they can be qualified by their group items. Thus, you would reference RATE OF TERMS-CODE and RATE OF AR-CODE in your program:

```
01 record-area.  
  03 terms-code.  
    05 rate          pic s9v999.  
    05 days          pic 9(3).  
    05 descript      pic x(15).  
  03 ar-code.  
    05 rate          pic s9v999.
```



```
05 days      pic 9(3) .  
05 descript  pic x(15) .
```

However, database systems consider duplicate names an error. Thus, if more than one field in a particular file has the same name, the data dictionary will not be generated for that file.

The solution to this situation is to add a NAME directive (see [section 5.3.3.7](#)) that associates an alternate name with one or both of the conflicting fields.

5.3.1.10 Long field names

The compiler determines whether field names are unique within the first 18 characters. This is because some RDBMSs truncate longer field names to 18 characters. (In the case of the OCCURS clause described above, the truncation is to the original name, not the appended index numbers.) If the field names are not unique within 18 characters, the XFD file is generated, but a warning message is issued.

You can, instead of allowing default truncation, use the NAME directive to give a shorter alias to a field with a long name. Note that within the COBOL application you will continue to use the original name. The NAME directive affects only the data dictionary.

5.3.1.11 Naming the XFD

The compiler needs to give a name to each XFD file (data dictionary) that is built. It attempts to build the name from your COBOL code, although there are some instances where the name in the code is nonspecific, and you must provide a name.

Each XFD name is built from a *starting name* that is derived (if possible) from the SELECT statement in your COBOL code. The following paragraphs explain how that occurs.

ASSIGN name is a variable

If the SELECT for the file has a variable ASSIGN name (ASSIGN TO *filename*), then you must specify a starting name for the XFD file via a FILE directive in your code. This process is described in [section 5.3.3.6, “FILE directive.”](#)

ASSIGN name is a constant

If the SELECT for the file has a constant ASSIGN name (such as ASSIGN TO “COMPFILE”), then that name is used as the starting name for the XFD name.

ASSIGN name is generic

If the ASSIGN phrase refers to a generic device (such as ASSIGN TO “DISK”), then the compiler uses the SELECT name as the starting name.

Forming the final XFD name

From the starting name, the final name is formed as follows:

1. The compiler removes any extensions from the starting name.
2. It constructs a “universal” base name by stripping out directory information that fits any of the formats used by the operating systems that run ACUCOBOL-GT.
3. It converts the base name to lower case.
4. It appends the letters “.xfd” to the base name.

Note: The FD can specify the file name using the “VALUE OF FILE_ID IS xxx” clause. (See Book 3, *Reference Manual*, **Chapter 5, section 5.4.8, “VALUE OF FILE-ID Clause”**) This phrase has a similar effect to the ASSIGN phrase of the SELECT statement, but when this phrase is used it overrides the ASSIGN phrase. So all of the above rules that relate to the ASSIGN phrase also relate to the FILE_ID phrase, when that phrase is used. (Technically, the ASSIGN phrase and the FILE_ID phrase are supposed to be identical, but the compiler silently overrides the ASSIGN with the FILE_ID phrase.)

5.3.1.12 Examples of XFD names

COBOL code	File name
ASSIGN TO “usr/ar/customer.dat”	customer.xfd
SELECT TESTFILE, ASSIGN TO DISK	testfile.xfd

COBOL code	File name
ASSIGN TO “-D SYSSLIB:HELP”	help.xfd
ASSIGN TO FILENAME	(you specify)

5.3.2 Using Directives

Directives are optional comments that you can use in your FDs to control how things look in the XFD file. Many applications will not use directives at all. They’re most commonly used when a site intends to do a lot of work with a database management system outside of the COBOL application, and wants to control how the database table is built. They are also used to customize how fields are displayed by the record editor, **alfred**.

Directives are special comments placed into an FD in your COBOL source code. They guide the building of the data dictionaries, which in turn guide the building of the database table (and the display of data in the record editor).

Each directive includes the letters XFD. These three letters indicate to the compiler that the comment is to be used in data dictionary generation.

Directives offer you a great deal of control over how the XFD file is built. Among other things, they enable you to:

- specify a field name to be used in the XFD, in place of a COBOL field name
- map elementary items of a group item together into a single field
- cause numeric COBOL data to be treated as a text string in the XFD
- cause the fields from a specific record in a file to appear in the XFD (rather than just the fields from the largest record)
- mark specific fields as candidates for a secondary database table, if all of the data will not fit into a single table
- cause a fixed-length field to be designated as variable-length

- assign a column the DATE type, so that it will have the built-in functionality that dates have in an RDBMS
- give a name to the data dictionary file itself
- include comments in XFD files

Directives are always placed within a COBOL FD. They do *not* affect Procedure Division I/O statements, and they do *not* change your COBOL fields in any way. Rather, they guide the building of the data dictionaries, giving you a measure of control over the way COBOL data is *mapped* to XFD fields.

5.3.2.1 Important for Acu4GL and AcuXML sites

Each data item in your COBOL FD must correspond to a field in the database. You may have columns in a table for which no data item exists in the COBOL FD, but not vice versa. You may not have data items in your FD with no counterpart in the database.

This correspondence happens automatically when the database is newly generated by the Acu4GL or AcuXML interface. If the database is pre-existing, you should check for the correspondence, and use directives if necessary to make it complete.

Data dictionaries may be built directly from your source code with no directives if the default mapping rules described earlier are sufficient for your situation. If you would like to override the default mapping behavior, or map a field to a different name, then you would add directives to your COBOL code.

5.3.3 Directive Syntax

Place each directive on a line by itself, immediately before the COBOL line to which it pertains.

Introduce each directive with a “\$” in the Indicator Area (column 7 in standard ANSI source format), followed immediately by the letters “XFD”, then the directive itself. There should be no space between the \$ and the XFD. Spaces are permitted elsewhere on the line as separators. For example, the NAME directive looks like this:

```
$XFD NAME = EMP-NUMBER
```

An alternate ANSI-compliant way to introduce a directive is with an asterisk (*) in the Indicator Area. In this case, you begin the directive with the letters “XFD” and enclose the entire comment in double parentheses. For example:

```
*(( XFD NAME = EMP-NUMBER ))
```

There should be no space between the asterisk and the double left parentheses. Spaces are permitted elsewhere on the line as separators.

You may use either form of the directive syntax (or both) in your applications.

Two or more directives that pertain to the same line of COBOL code may be combined on one comment line. The directives should be separated by a space or a comma. For example, to specify both USE GROUP and NUMERIC at the same time, you would add this line:

```
$XFD USE GROUP, NUMERIC
```

or

```
*(( XFD USE GROUP, NUMERIC ))
```

The following pages describe each of the directives, in alphabetical order.

5.3.3.1 ALPHA directive

The ALPHA directive allows you to treat a data item as alphanumeric text in the database, when it is declared as numeric in the COBOL program. Also, ALPHA causes the **alfred** record editor to edit the field as alphanumeric instead of as numeric.

Syntax

```
$XFD ALPHA
```

or

```
* (( XFD ALPHA ))
```

This is especially useful when you have numeric keys in which you occasionally store non-numeric data, such as LOW-VALUES or special codes. In this situation, treating the field as alphanumeric allows you to move any kind of data to it.

Example

Suppose you have specified `KEY IS code-key`. Then assume the following record definition:

```
01 code-record.  
  03 code-key.  
    05 code-num  pic 9(5).
```

In a database, group items are disregarded, so `CODE-NUM` is the actual key field. Suppose you needed to move a non-numeric value to the key:

```
MOVE "C0531" TO CODE-KEY.  
WRITE CODE-RECORD.
```

In this case, the results are not well-defined, because a non-numeric value has been moved into a numeric field. The database might very well reject the record.

One way to solve this problem is to use the `ALPHA` directive. This causes the corresponding database field to accept alphanumeric data:

```
01 code-record.  
  03 code-key.  
$XFD ALPHA  
    05 code-num  pic 9(5).
```

As an alternative, you could specify the `USE GROUP` directive (see [section 5.3.3.11](#)) on the line before “code-key”. The `USE GROUP` directive implies that the field is alphanumeric.

5.3.3.2 BINARY directive

The `BINARY` directive is used to specify that the data in the field could be alphanumeric data of any classification. Absolutely any data is allowed.

The method of storing fields declared as binary is database-specific. For example, with Informix databases, binary data is stored in “char” fields with an extra leading character. This character always contains a space.

Syntax

```
$XFD BINARY
```

or

```
*(( XFD BINARY ))
```

Example

You might use this directive when you need to store a key that contains LOW-VALUES; COBOL allows a numeric field to contain LOW or HIGH values, but these are invalid for a numeric field in the RDBMS:

```
01 code-record.
  03 code-key.
    05 code-indic    pic x.
  *(( XFD BINARY ))
    05 code-num      pic 9(5).
    05 code-suffix   pic x(3).
    .
    .
    .
  move low-values to code-num.
```

5.3.3.3 COBOL-TRIGGER directive

Use this directive to indicate that a COBOL trigger is to be executed.

Syntax

```
$XFD COBOL-TRIGGER=program name
```

or

```
*(( XFD COBOL-TRIGGER=program-name ))
```

This XFD must be immediately before the FD of the file for which the trigger is defined.

Three parameters are passed to the COBL program:

```
01  OPCODE .
    88  READ-BEFORE          VALUE "r" .
    88  READ-AFTER          VALUE "R" .
    88  WRITE-BEFORE         VALUE "w" .
    88  WRITE-AFTER         VALUE "W" .
    88  REWRITE-BEFORE      VALUE "u" .
    88  REWRITE-AFTER      VALUE "U" .
    88  DELETE-BEFORE       VALUE "d" .
    88  DELETE-AFTER       VALUE "D" .

01  FILE-RECORD              PIC X(MAX-RECORD-SIZE) .

01  ERROR-CODE              PIC 99 .
```

You can use **C\$PARAMSIZE** to determine the size of the record passed, in case you have variable-length records. (See Appendix I, “Library Routines,” in Book 4 of the ACUCOBOL-GT documentation set for more information on C\$PARAMSIZE.) In cases where the record size is known, the actual record is passed (READ-AFTER (all), WRITE-BEFORE, WRITE-AFTER, REWRITE-BEFORE, REWRITE-AFTER). In cases where the record size is not known, the maximum record size is passed (READ-BEFORE (all), DELETE-BEFORE, DELETE-AFTER). The contents of this field will be identical to the record area given to the file operation. The BEFORE images will have the value before the file operation, and the AFTER images will have the value after the file operation.

Use the ERROR-CODE parameter to signal to Acu4GL that an error occurred. Setting this in a BEFORE trigger causes the actual file operation to not execute. Setting this in an AFTER trigger causes the runtime to treat the file operation as an error, though in fact the file operation did execute. This means that, for sequential access (next or previous), the file position has changed and a subsequent next or previous will act as if the prior file operation succeeded. Also, setting an error in the REWRITE-AFTER or DELETE-AFTER will not actually return the record to its prior state—the record will be modified or deleted in the database. We suggest using transactions if this behavior is not acceptable.

If the COBOL program can not be called (for any reason), it is treated as having succeeded.

5.3.3.4 COMMENT directive

The COMMENT directive allows you to include comments in an XFD file. Comments enable you to embed information in the XFD file that might be useful to other applications that access the data dictionary. Because the information is embedded in a comment, it does not interfere with processing by the **alfred** record editor or by the Acu4GL database interfaces. A comment is any line in the XFD file that starts with “#” in column 1.

Syntax

```
$XFD COMMENT comment
```

or

```
*(( XFD COMMENT comment ))
```

The text following the directive is copied into the XFD file as a comment. For example:

```
$XFD COMMENT This is a comment
```

5.3.3.5 DATE directive

The DATE directive instructs the runtime to store a field in the database as a date. Because there’s no COBOL syntax that identifies a field as a date, you may want to add this directive to differentiate dates from other numbers, so that they enjoy the properties associated with dates in the RDBMS. Note that the record editor **alfred**, does not validate the data you enter into date fields.

Syntax

```
$XFD DATE=date-format-string
```

or

```
*(( XFD DATE=date-format-string ))
```

The *date-format-string* is a description of the desired date format in the COBOL program, composed of characters from the following list:

M	month (01 - 12)
Y	year (2 or 4-digit)

D	day of month (01 - 31)
J	Julian day (00000000 - 99999999)
E	day of year (001 - 366)
H	hour (00 - 23)
N	minute
S	second
T	hundredths of a second

Any other characters cause the date format string to be invalid, and result in a “corrupt XFD” error or a compile-time warning.

Each character in a date format string can be considered a placeholder that represents the type of information stored at that location. The characters also determine how many digits will be used for each type of data.

For example, although you would typically represent the month with two digits, if you specify MMM as part of your date format, the resulting date will use three digits for the month, left-zero-filling the value. If the month is given as M, the resulting date will use a single digit, and will truncate on the left.

If you do not specify a date-format-string, the default is YYMMDD if the field has six digits, or YYYYMMDD if the field has eight digits.

Julian dates

Because the definition of Julian day varies, the DATE directive offers a great deal of flexibility for representing Julian dates. Many source books define the Julian day as the day of the year, with January 1st being 001, January 2nd being 002, and so forth. If you want to use this definition for Julian day, simply use EEE (day of year) in your date formats.

Other reference books define the Julian day as the number of days since some specific “base date” in the past. This definition is represented in the DATE directive with the letter J (for example, a six-digit date field would be preceded with the directive “\$XFD DATE=JJJJJ”). The default “base date” for this form of Julian date is 12/31/4714 BC.

You may define your own base date for Julian date calculations by setting the runtime configuration variable “4GL-JULIAN-BASE-DATE.” The value of this variable must have the format “YYYYMMDD”. See the *Acu4GL User’s Guide* for more information.

Using group items

You may place the DATE directive in front of a group item, so long as you also use the USE GROUP directive (see Example 2 below).

Example 1

```
$xfd date
  03 date-hired      pic 9(8).
  03 pay-scale       pic x(3).
```

The column `date_hired` will have eight digits and will be type DATE in the database, with a format of YYYYMMDD.

Example 2

```
*(( XFD DATE, USE GROUP ))
  03 date-hired.
    05 yyyy      pic 9(4).
    05 mm        pic 9(2).
    05 dd        pic 9(2).
```

This also will produce a column named `date_hired` with eight digits and type DATE in the database, format YYYYMMDD.

Example 3

```
*(( XFD DATE=EEEEYYY))
  03 date-sold      pic 9(7).
  03 sales-rep     pic x(30).
```

This will produce a column named `date_sold` with seven digits and type DATE in the database. The date will contain the day of the year (for example, February 1st would be 032), followed by the four-digit year, such as 1999.

FY and RY date formats

The FH and RY date format characters have very precise requirements and are used to handle a specific case where eight digit dates are expressed in six characters.

Instead of YYYY or YY, you can specify FY to mean that the first character of the year specifies the decade instead of the “tens” year. The decade can be a character between space (“ ”) and “I” (inclusive). For the characters “0” through “9” to be treated as decades in the 20th century, the decade characters have the following meaning:

	00	10	20	30	40	50	60	70	80	90
1700					spc	!	“	#	\$	%
1800	&	'	()	*	+	,	-	.	/
1900	0	1	2	3	4	5	6	7	8	9
2000	:	;	<	=	>	?	@	A	B	C
2010	D	E	F	G	H	I				

This means that a date of ?70210 is converted to 20570210, or February 10, 2057. The range of valid dates is 00101 (Jan 1, 1740) through I91231 (Dec 31, 2159).

Instead of YYYY or YY, you can also specify RY to mean that the first character of the year specifies the decade and that the entire date is to be 9s complement (to be able to sort dates in reverse order). Note that the entire date and time is treated as a 9s complement number in this case. The decade characters have the following meaning:

	00	10	20	30	40	50	60	70	80	90
1700					I	H	G	F	E	D
1800	C	B	A	@	?	>	=	<	;	:
1900	9	8	7	6	5	4	3	2	1	0
2000	/	.	-	,	+	*)	('	&

	00	10	20	30	40	50	60	70	80	90
2010	%	\$	#	“	!	spc				

The date 20570210 is now specified with *29789. The range of valid dates is 199898 (Jan 1, 1740) through 08768 (Dec 31, 2159), the same valid range as when using F instead of R.

While using F and R before month, day, hour, or any other format specifier does not generate a compile error for the XFD, the results are undefined at runtime.

Note that if you use these format specifiers with Acu4GL, the actual date is written to the database, not the encoded date. That means that the R specifier is not very useful in this scenario (you won't be able to read forward through a file in reverse date order). This format specifier is more useful with AcuXDBC where the data is in Vision format.

5.3.3.6 FILE directive

The FILE directive supplies a *starting name* from which the data dictionary file name is formed. This directive is required only when the file name in the COBOL code is nonspecific. For example, you would use this directive when the SELECT for the file has a variable ASSIGN name (ASSIGN TO *variable_name*). In this situation, the interface cannot form a file name automatically, and you must provide a name.

A starting name is simply a short file name that serves as the basis for the dictionary name.

Syntax

```
$XFD FILE=name
```

or

```
*(( XFD FILE=name ))
```

This directive must appear on the line immediately preceding the file's FD.

Special characters (such as \$) can be included within the filename if the name is surrounded by either single or double quotes. (Example: \$XFD FILE="f\$test") No quotes are required if the filename does not include special characters.

Example

Suppose your SELECT statement has a variable ASSIGN name such as the one shown here:

```
select my-file
  assign to my-variable.
```

Then you would need to add the FILE directive as shown here:

```
select my-file
  assign to my-variable.
.
.
.
$xfd file=payroll
fd my-file
```

Note: This directive must appear immediately before the file's FD.

5.3.3.7 NAME directive

The NAME directive assigns a field name to the field defined on the next line.

Syntax

```
$XFD NAME=fieldname
```

or

```
*(( XFD NAME=fieldname ))
```

This directive has several uses, as shown in the following examples.

Example 1

Within a database file, all field names must be unique. (Multiple database tables may include the same field name, but duplicates may not exist within a single table.) Unique field names are not required in COBOL, because names can be qualified by group items. For example, in COBOL this is done:

```
01 employee-record.
   03 date-hired.
       05 yy pic 99.
       05 mm pic 99.
       05 dd pic 99.
   03 date-last-paid.
       05 yy pic 99.
       05 mm pic 99.
       05 dd pic 99.
```

You need not change the field names in your COBOL program to access a database. Instead, you use the NAME directive to provide unique database names for the fields. For example:

```
01 employee-record.
   03 date-hired.
       05 yy pic 99.
       05 mm pic 99.
       05 dd pic 99.
   03 date-last-paid.
* (( xfd name=year-paid ))
       05 yy pic 99.
$xfd name=month-paid
       05 mm pic 99.
$xfd name=day-paid
       05 dd pic 99.
```

The dates portion of the resulting database table will look like:

yy	mm	dd	year_paid	month_paid	day_paid
88	02	15	94	04	30

Example 2

Some SQL-based databases require that names be no more than 18 characters long. For those systems, the Acu4GL runtime will automatically truncate longer COBOL names after the eighteenth character.

If you have names that are identical within the first 18 characters, or that would not be meaningful to you if shortened to the first 18 characters, you can use the NAME directive to assign them different database field names.

For example, a portion of your database might contain:

```
01 acme-employee-record.  
   03 acme-employee-record-date-hired  
      pic x(6).  
   03 acme-employee-record-date-last-paid  
      pic x(6).
```

You could add two NAME directives to differentiate the two item names by making them meaningful within 18 characters:

```
01 acme-employee-record.  
$xfd name=date-hired  
   03 acme-employee-record-date-hired  
      pic x(6).  
$xfd name=date-last-paid  
   03 acme-employee-record-date-last-paid  
      pic x(6).
```

Note: Your COBOL names have not changed. The new names are used only for the XFD fields.

Each time you compile your program and specify “-Fx” to create data dictionaries, any field names longer than 18 characters will be checked for uniqueness within the first 18. If any field names are identical for the first 18 characters, a compiler warning message will be issued. A warning of this type does not prevent the program from compiling and does not prevent the data dictionary from being generated.

Example 3

You may want to use the NAME directive to assign shorter names than those used in your COBOL programs. This makes the formation of interactive SQL queries easier and quicker. For example:

```
*(( XFD  NAME=EMPNO ))
  03  employee-number  pic x(8).
```

This directive causes the data dictionary to map EMPLOYEE-NUMBER to EMPNO in the database.

Example 4

If your database already exists, and a field name in the database does not match the name used in your COBOL FD, you can use a NAME directive to associate the two names. For example:

```
$xfd  name=employee-no
  03  employee-number  pic x(8).
```

This directive causes the data dictionary to map EMPLOYEE-NUMBER in the COBOL program to EMPLOYEE-NO in the database.

Example 5

If your COBOL program uses field names that begin with a numeric character, use the NAME directive to assign a different name for use with your database. SQL will typically generate a syntax error when it encounters a column name that begins with a numeric character. For example:

```
  03  12-months-sales  pic 9(5)V99.
```

could be renamed this way:

```
$xfd  name=twelve-months-sales
  03  12-months-sales  pic 9(5)V99.
```

5.3.3.8 NUMERIC directive

The NUMERIC directive allows you to treat a data item as an unsigned integer when it is declared as alphanumeric. You might use this when the data stored in the item is always numeric.

Syntax

```
$XFD NUMERIC
```

or

```
*(( XFD NUMERIC ))
```

Example

```
$xfd numeric  
01 student-code pic x(7).
```

5.3.3.9 SECONDARY-TABLE directive

Some RDBMSs such as Sybase permit subordinate tables. When this is the case, you can use the SECONDARY-TABLE directive to indicate that the next data item may be placed into a subordinate table, if more than one table is necessary to accommodate the data.

Up to 16 subordinate tables can be created from a single record description. Each table name is based on the original table name, with a letter from “A” to “P” appended. For example, if the original table were named “my-table,” then subsequent subordinate tables would be given these names:

```
my-tableA  
my-tableB  
my-tableC  
my-tableD
```

When the runtime accesses the data dictionary, it makes an initial pass through the data, taking all eligible data items for which the SECONDARY-TABLE directive is *not* specified.

SECONDARY-TABLE is ignored for certain data items. These include:

- any field that is part of a key
- any field that is referenced in a WHEN directive

If the table size is not exceeded in the first pass, then, in a second pass, the runtime appends to the original table all items marked SECONDARY-TABLE that can be accommodated.

When the first table reaches a limit (either in total number of columns or total number of characters), that table is created, and a new table is begun. Items that did not fit into the previous table are placed into a subordinate table, in the order in which they are encountered.

The process repeats until all items have been accommodated.

It's permissible to place the SECONDARY-TABLE directive just before a level 01 record definition. In this case, it applies to all fields underneath the level 01.

Syntax

```
$XFD SECONDARY-TABLE
```

or

```
*(( XFD SECONDARY-TABLE ))
```

Example

```
$xfd secondary-table
01 description pic x(80).
```

5.3.3.10 SUBTABLE directive (AcuXDBC use only)

This directive modifies the way fields that appear in an OCCURS clause are processed resulting in the creation of subtables (see the [Section 5.3.1.7](#) for more information on the OCCURS clause). This directive instructs the XFD parsing routines not to append the occurrence number to the field name, as would normally take place with OCCURS clauses, but instead store just the base name along with the name of the subtable as written in the XFD file.

Syntax

```
$XFD SUBTABLE=name
```

or

```
*(( XFD SUBTABLE=name ))
```

Example

```
*((XFD SUBTABLE=subtab1))
03 employee-table occurs 10 times.
```

```
05 employee-numberpic 9(3).
```

Note: AcuXDBC is the only product that currently makes use of this directive. Using this directive with other products such as Acu4GL will result in errors.

5.3.3.11 USE GROUP directive

The USE GROUP directive allows you to enter a group item into the XFD as a single field, instead of using the elements contained in the group. This is necessary if the item is stored in a pre-existing database as a group, rather than as individual fields.

Combining fields of data into meaningful groups for database storage also improves I/O efficiency.

Syntax

```
$XFD USE GROUP
```

or

```
*(( XFD USE GROUP ))
```

By default, the USE GROUP directive implies that the consolidated field is alphanumeric. If you want a numeric field, simply add the word NUMERIC at the end of the directive.

Example 1

For example, the directive in the following code combines the functions of the USE GROUP and DATE directives, and indicates that the date should be entered into the XFD as a single date-formatted data item instead of three distinct fields:

```
*(( XFD USE GROUP, DATE ))
03 date-entered.
05 yy pic 99.
05 mm pic 99.
05 dd pic 99.
```

Either a comma or a space (or both) may separate the word DATE from the words USE GROUP.

Example 2

Other fields with which you might use this directive include multi-part account numbers or department numbers, or keys that are referenced as a unit but not by their individual pieces. Here's an example of an item that might be grouped:

```
$xfd use group
01 gl-acct-no.
   03 main-acct      pic 9(4).
   03 sub-acct       pic 9(3).
   03 dept-no        pic 9(3).
```

If you are using a pre-existing database in which certain fields are grouped, they must also be grouped in your COBOL FD.

If the database does not yet exist, keep in mind that combining fields into groups typically improves execution speed. Whether to group fields or not also depends on how you want to process them. Do you always store and use the fields together? Someone who really knows how the data is being used might help to identify groups of fields that can be combined to speed processing.

5.3.3.12 VAR-LENGTH directive

By default, the compiler generates fixed-length fields in the XFD. The VAR-LENGTH directive requests that the data item that immediately follows the directive be assigned a type that implies variable length, if possible. This can save considerable space in your database.

The precise variable type that is assigned to the data item depends on which DBMS is in use.

Possible variable types that might be assigned are VARCHAR and VARBINARY.

Syntax

```
$XFD VAR-LENGTH
```

or

```
* (( XFD VAR-LENGTH ))
```

Example

For example, the directive in the following code indicates that the employee-name field should be entered into a Sybase database as a VARCHAR data item.

```
* (( XFD VAR-LENGTH ))  
03 employee-name pic x(45).
```

5.3.3.13 WHEN directive

The WHEN directive is used when you want to include multiple record definitions or REDEFINES in the XFD file. It's typically used when you want to force certain database columns to be built that would not be built by default (because you want to use them from the RDBMS side). It can also force certain fields to be displayed in the **alfred** record editor.

Note: You cannot use the WHEN directive in an OCCURS clause.

Recall that the key fields and the fields from the *largest* record are automatically included as explicit fields in the XFD file. *All fields* are stored and retrieved, whether they appear as explicit fields or not. So you do not need to use WHEN unless you want to ensure that some additional fields are *explicitly listed by name* in the XFD.

WHEN declares that the field (or subordinate fields, if it is a group item) that immediately follows the directive must appear as a field (or fields) in the XFD. It also states one condition under which the fields are to be used. WHEN thus guarantees you that the fields will be explicitly included in the XFD (as long as they are not FILLER and do not occupy the same area as key fields).

A WHEN directive may also be assigned an optional *tablename*. If you assign a tablename, then the data that immediately follows the WHEN directive and meets the specified condition will be considered as a separate table.

Syntax

```

$XFD WHEN field=value [TABLENAME=name] (equals)
$XFD WHEN field=OTHER [TABLENAME=name] (equals)
$XFD WHEN field<=value [TABLENAME=name] (is less than or equal)
$XFD WHEN field<value [TABLENAME=name] (is less than)
$XFD WHEN field>=value [TABLENAME=name] (is greater than or equal)
$XFD WHEN field>value [TABLENAME=name] (is greater than)
$XFD WHEN field!=value [TABLENAME=name] (is not equal to)

```

or

```
*((XFD WHEN field=value [TABLENAME=name])) (also <, <=, >, >=, !=)
```

The *value* may be an explicit data value (in quotes). The word OTHER, which can be used only with “=”, means “use the field(s) that are subject to the WHEN OTHER condition only if none of the other WHEN conditions listed for this field is met.”

Note: Keep in mind the distinction between a field *subject* to a condition and a field *in* a WHEN condition (i.e., the field for which a condition is listed). A field *subject* to a WHEN condition is one that immediately follows the WHEN directive or is subordinate to a group item that immediately follows the WHEN directive. The field *in* a WHEN condition is the field before the arithmetic operator and is a part of the directive itself.

In other words, a WHEN OTHER condition is true if all *other* WHEN conditions with the same field *in* them are false. If a WHEN condition is false, the fields *subject* to it are not written.

For example:

```

01 ar-code-type.
*(( xfd when ar-code-type = "s" ))
   03 ship-code-record          pic x(4).
*(( xfd when ar-code-type = "b" ))
   03 backorder-code-record redefines
      ship-code-record.
*(( xfd when ar-code-type = other ))
   03 obsolete-code-record redefines
      ship-code-record.

```

OTHER may be used before one record definition, and may be used once at *each nesting level* within each record definition.

Note: A WHEN directive with condition OTHER *must* be used if there is a possibility that the data in the field will not meet any of the explicit equality conditions specified in the other WHEN directives; if this is not done, results are undefined.

Example 1

If the following code were compiled without directives, the underlined fields would appear explicitly in the XFD file. Note that the key fields, and the fields from the longest record, would be included automatically. FILLER would be ignored:

```
01 ar-codes-record.
  03 ar-codes-key.
    05 ar-code-type pic x.
    05 ar-code-num pic 999.
01 ship-code-record.
  03 filler pic x(4).
  03 ship-instruct pic x(15).
01 terms-code-record.
  03 filler pic x(4).
  03 terms-rate-1 pic s9v999.
  03 terms-days-1 pic 9(3).
  03 terms-rate-2 pic s9v999.
  03 terms-descript pic x(15).
```

If you added the WHEN directive as shown below, it would cause the fields from the SHIP-CODE-RECORD to be included in the XFD file, and would determine when specific fields would be used. The underlined fields would appear as columns in a database table if you were using Acu4GL:

```
01 ar-codes-record.
  03 ar-codes-key.
    05 ar-code-type pic x.
    05 ar-code-num pic 999.
$xfd when ar-code-type = "s"
01 ship-code-record.
  03 filler pic x(4).
  03 ship-instruct pic x(15).
$xfd when ar-code-type = "t"
01 terms-code-record.
  03 filler pic x(4).
  03 terms-rate-1 pic s9v999.
```



```

03 terms-days-1      pic 9(3)
03 terms-rate-2     pic s9v999.
03 terms-descript   pic x(15).

```

FILLER data items do not have unique names and are thus not used to form field names in the XFD. You could use the NAME directive to give them a name if you really need to have them in the XFD. However, in this example the FILLER data items implicitly redefine key fields. Thus, they would be disregarded *even if you provided a name for them*.

Example 2

In the following code, in which no WHEN directives are used, the underlined fields will be explicitly named in the XFD. (Key fields have the suffix “key” in their names in this example.)

Note: REDEFINES records simply re-map the same data area and are not explicitly included in the XFD by default.

```

01 archive-record.
   03 filler                pic x(33).
   03 archive-code         pic 9(6).
   03 archive-location     pic 9(2).
   03 filler                pic x(10).
01 master-record.
   03 animal-id-key.
       05 patient-id          pic 9(6).
       05 species-code-type   pic 9(5).
       05 species-name       pic x(6).
   03 service-code-key.
       05 service-code-type   pic 9(6).
       05 service-name       pic x(10).
   03 billing-code.
       05 billing-code-type   pic 9(4).
       05 plan-name           pic x(8).
   03 office-info.
       05 date-in-office     pic 9(8).
       05 served-by-name     pic x(10).
   03 remote-info redefines office-info.
       05 van-id           pic 9(4).
       05 proc-code       pic 9(8).
       05 vet-name        pic x(6).

```

If you added the WHEN directives shown below, you would add several fields to the XFD. The fields that would appear are underlined:

```
*(( xfd when animal-id-key = "0000000000000000" ))
01  archive-record.
    03  filler                pic x(33).
    03  archive-code         pic 9(6).
    03  archive-location    pic 9(2).
    03  filler                pic x(10).
*(( xfd  when animal-id-key =  other ))
01  master-record.
*(( xfd  use group ))
03  animal-id-key.
    05  patient-id           pic 9(6).
    05  species-code-type   pic 9(5).
    05  species-name        pic x(6).
03  service-code-key.
    05  service-code-type   pic 9(6).
    05  service-name        pic x(10).
03  billing-code.
    05  billing-code-type   pic 9(4).
    05  plan-name           pic x(8).
*(( xfd when billing-code-type = "1440" ))
03  office-info.
    05  date-in-office      pic 9(8).
    05  served-by-name     pic x(10).
*(( xfd when billing-code-type =  other ))
03  remote-info redefines office-info.
    05  van-id              pic 9(4).
    05  proc-code          pic 9(8).
    05  vet-name           pic x(6).
```

Example 3

If your application has a REDEFINES whose field names are more meaningful than the fields they redefine, you might consider switching the order of your code, rather than using a WHEN directive. Use the less significant field names in the REDEFINES. For example, you might change this:

```
03  code-info.
    05  filler                pic 9(8).
    05  code-1               pic x(10).
03  patient-info redefines code-info.
    05  patient-id          pic 9(4).
```

```

05 service-code pic 9(8).
05 server-name  pic x(6).

```

to this:

```

03 patient-info.
05 patient-id    pic 9(4).
05 service-code pic 9(8).
05 server-name  pic x(6).
03 code-info redefines patient-info.
05 filler        pic 9(8).
05 code-1        pic x(10).

```

The fields that would appear in the XFD by default are underlined above. This shows how the XFD field names might become more meaningful when the order is reversed. Your application operates the same either way.

Note: When a WHEN directive condition is met, COBOL record definitions or REDEFINES records that are subordinate to other WHEN directives are not used. Database columns in rows which correspond to those records are set to the special database value “NULL”. This means that there is no value provided for those columns. “NULL” is not equivalent to zero, and it has special properties in the RDBMS. For example, you can select all rows for which a given column is NULL.

Example 4

This COBOL code:

```

01 example4-record.
03 col-type                pic x.
$xfid when col-type = "a"
03 def1                    pic x(2).
$xfid when col-type = "b"
03 def2 redefines def1     pic 9(2).

```

will result in this database table:

col_type	def1	def2
a	xx	null
b	null	10

col_type	def1	def2
a	yy	null

5.3.3.14 XSL directive

If you are using the “-Fe” compiler option to generate XML-style XFD files, the XSL directive allows you to include a stylesheet reference in the XML header.

Syntax

```
$XFD XSL=stylesheet
```

or

```
*(( XFD XSL="stylesheet" ))
```

where *stylesheet* is an alphanumeric literal indicating the appropriate stylesheet. The compiler includes the following line in all generated XML-style XFD files:

```
<?xml-stylesheet type="text/xsl" href="stylesheet"?>
```

For example:

```
*(( XFD XSL="myxsl.xsl" ))
```

generates this line:

```
<?xml-stylesheet type="text/xsl" href="myxsl.xsl"?>
```

5.3.4 XFD Format

An XFD file may be created either as a simple text file or as an XML file. In either case, the XFD contains a description of a COBOL indexed, sequential, or relative file based on its fields.

5.3.4.1 Identification section

The identification section describes the data file described by the XFD. It contains information about the version of XFD created and the data file type (indexed, relative, or sequential). It also lists the data file's SELECT and base file name, as well as basic record information (minimum and maximum record size, number of keys).

Consider the following SELECT statement:

```
SELECT clients ASSIGN TO DISK "clients.dat"
ORGANIZATION IS INDEXED
...
RECORD KEY IS cl-client-id
ALTERNATE RECORD KEY IS key01 = cl-first-name, cl-last-name
WITH DUPLICATES.
```

In a simple text XFD, the identification section appears similar to the following:

```
# [Identification Section]
XFD,06,CLIENTS,CLIENTS,12
0000000197,0000000197,002
000,18,046,044,00
```

Here, “XFD,06” specifies that the XFD file was created in version 6 format. The SELECT and base file name are both “CLIENTS”, and the data file is in indexed format (“12”). The other available file formats are relative (“08”), and sequential (“04”).

The minimum and maximum record length for the file described in the XFD are the same (“0000000197”), and there are “002” keys.

The next line of digits are derived from various compile options and program settings that may be used when creating the XFD. The first three-digit sequence (“000”) indicates sign compatibility. The possible values are:

```
FLG_ACU_SIGNS = 000
FLG_IBM_SIGNS = 004
FLG_MF_SIGNS = 008
FLG_NCR_SIGNS = 020
FLG_VAX_SIGNS = 036
FLG_MBP_SIGNS = 072
```

FLG_REA_SIGNS = 128

The above values are set by the “-Dc” compiler options. See **Chapter 2, section 2.2.10, “Data Storage Options”** for details on using these options.

The next two-digit sequence (“18”) indicates the maximum numeric digits. The possible values are “18” and “31”. This is set by the “-Dd31” compiler option.

The next three-digit sequence (“046”) indicates the decimal value of the character used as the program period.

The next three-digit sequence (“044”) indicates the decimal value of the character used as the program comma.

The last two-digit sequence (“00”) is a UNICODE indicator.

In XML format, the same information is presented as follows:

```
<xfd:identification xfd:version="6">
  <xfd:select-name>CLIENTS</xfd:select-name>
  <xfd:table-name>CLIENTS</xfd:table-name>
  <xfd:file-organization>Indexed</xfd:file-organization>
  <xfd:maximum-record-size>197</xfd:maximum-record-size>
  <xfd:minimum-record-size>197</xfd:minimum-record-size>
  <xfd:number-of-keys>2</xfd:number-of-keys>
  <xfd:sign-compatibility>0</xfd:sign-compatibility>
  <xfd:maximum-numeric-digits>18</xfd:maximum-numeric-digits>
  <xfd:period-character>.</xfd:period-character>
  <xfd:comma-character>,</xfd:comma-character>
  <xfd:alphabet>ASCII</xfd:alphabet>
</xfd:identification>
```

5.3.4.2 Key section

The key section contains an entry for each key in the file. The following example refers to the SELECT cited in the previous section, and shows the key information in a simple text format:

```
# [Key Section]
01,0,005,0000000000
01
CL-CLIENT-ID
02,1,015,0000000005,015,0000000020
```

```

02
CL-FIRST-NAME
CL-LAST-NAME

```

For each key, there is a line describing the number of segments in the key, whether or not the key allows duplicates, and the size and offset for each segment in the key. Segment size and offset are repeated as necessary to describe each segment in the key.

In the example given, the primary key (CL-CLIENT-ID), is defined as having one segment and being a unique key (“01,0”). The size of the segment is 5 (“005”), and the offset is 0.

The alternate key is defined as having two segments and allowing alternates (“02,1”). The first segment is described as having a size of 15 and an offset of 5; the second segment also has a size of 15, but its offset is 20.

The next line in each key description contains the number of elementary field definitions defined in the key. For the primary key in the example, there is one elementary field. In the alternate key, there are two.

The remaining lines in the key section give the name of each elementary field contained in the key.

In XML format, the key section appears as follows:

```

<xfd:keys>
  <xfd:key xfd:segment-count="1" xfd:duplicates-allowed="false">
    <xfd:segments>
      <xfd:segment xfd:segment-size="5" xfd:segment-offset="0"/>
    </xfid:segments>
    <xfd:key-columns xfd:key-column-count="1">
      <xfd:key-column xfd:key-column-name="CL-CLIENT-ID"/>
    </xfid:key-columns>
  </xfid:key>
  <xfd:key xfd:segment-count="2" xfd:duplicates-allowed="true">
    <xfd:segments>
      <xfd:segment xfd:segment-size="15" xfd:segment-offset="5"/>
      <xfd:segment xfd:segment-size="15" xfd:segment-offset="20"/>
    </xfid:segments>
    <xfd:key-columns xfd:key-column-count="2">
      <xfd:key-column xfd:key-column-name="CL-FIRST-NAME"/>
      <xfd:key-column xfd:key-column-name="CL-LAST-NAME"/>
    </xfid:key-columns>
  </xfid:key>

```

```
</xfd:keys>
```

5.3.4.3 Condition section

Condition definitions identify fields that are not present in every record. This section of the XFD specifies the number of condition definitions set, followed by the actual definitions. Each definition appears on its own line, and contains the condition number and type. A table of condition types appears at the end of this section.

In a simple text XFD, the condition section appears as follows:

```
# [Condition Section]
001
001,4,CL-CLIENT-ID,200
```

Here, there is one condition (“001”). Condition number “001” is a type “4” (greater than, “>”) condition specifying that the field “CL-CLIENT-ID” is greater than “200”.

In XML format, the same condition section would appear as follows:

```
<xfd:conditions xfd:condition-count="1" >
  <xfd:condition xfd:condition-number="1">
    <xfd:condition-comparison>gt</xfd:condition-comparison>
    <xfd:condition-variable>CL-CLIENT-ID</xfd:condition-variable>
    <xfd:condition-value>200</xfd:condition-value>
  </xfd:condition>
</xfd:conditions>
```

The recognized condition types are:

Type	Description
1	Equal to (=)
2	And
3	Other
4	Greater than (>)
5	Greater than or equal to (>=)
6	Less than (<)
7	Less than or equal to (<=)

Type	Description
8	Not equal to (!=)
9	Or

5.3.4.4 Field section

The final section of the XFD is the field section. This is the longest section of the file, because it describes all of the fields contained in the data file.

The file begins with a line describing the number of elementary fields defined for inclusion in the XFD (four digits), the total number of elementary fields in the file (five digits), the number of group and elementary fields listed in the FD (five digits), and the total number of group and elementary fields listed in the FD (five digits).

These last two items—the number and total number of group and elementary fields—may be different if the FD contains an OCCURS clause. Although the *total* number is increased by the number of times the item OCCURS, the item still has only one listing in the field section of the XFD. Total number is also affected by redefines, group items, and multiple FD descriptions, but it excludes filler fields.

Following the summary line, the field section contains a line for each field definition, listing:

- The field offset, in bytes (10 digits)
- The field size, in bytes (10 digits)
- The field type, as described in the following table:

Type	Description
Values	
0	Numeric edited
1	Unsigned numeric
2	Signed numeric (trailing separate)
3	Signed numeric (training combined)

Type Values	Description
4	Signed numeric (leading separate)
5	Signed numeric (leading combined)
6	Signed computational
7	Unsigned computational
8	Positive packed-decimal
9	Signed packed-decimal
10	Computational-6
11	Signed binary
12	Unsigned binary
13	Signed native-order binary
14	Unsigned native-order binary
16	Alphanumeric
17	Alphanumeric (justified)
18	Alphabetic
19	Alphabetic (justified)
20	Alphanumeric edited
22	Group
23	Float or double
24	National
25	National (justified)
26	National edited
27	Wide
28	Wide (justified)
29	Wide edited

- For numeric fields, this holds the number of digits in the field. For non-numeric fields, the number shown is the same as the size of the field (10 digits).

- The scale of the field (+ or - and a two-digit number), expressed as a power of 10. The scale for non-numeric fields is “0”.
- The field’s “user type”. This is “000” for no user type, “001” for date fields, “002” for binary fields, and “003” for variable-length character. If “secondary table” is used, 16 is added to the value. So “017” would be the entry for a date in the secondary table.
- The condition number to apply to this field. This is “000” if the field appears in every record, or “999” if the field is a group item.
- The field level number (2 digits)
- The field name (30 characters)
- An optional date format if the field is a date (only included if you put a *date_format_string* in your COBOL program)

When a field uses the OCCURS phrase, two special entries appear in the list of field definitions. These entries are designated by offsets of “90001” and “90002”. The START-OCCURS entry is defined as follows:

```
90001,num-occurs,size,START-OCCURS
```

The END-OCCURS entry is defined as follows:

```
90002,END-OCCURS
```

The fields contained in the OCCURS clause are listed between these markers using normal field notation.

Fields that are not defined for actual use but that occur in the FD are indicated by the special condition marker “999”.

A simple text field section would appear as follows:

```
# [Field Section]
0010,00010,00011,00011
0000000000,0000000197,16,0000000197,+00,000,999,01,CLIENT-RECORD
0000000000,0000000005,01,0000000005,+00,000,000,05,CL-CLIENT-ID
0000000005,0000000015,16,0000000015,+00,000,000,05,CL-FIRST-NAME
0000000020,0000000015,16,0000000015,+00,000,000,05,CL-LAST-NAME
0000000035,0000000040,16,0000000040,+00,000,000,05,CL-STREET-ADDR
0000000075,0000000030,16,0000000030,+00,000,000,05,CL-CITY
0000000105,0000000020,16,0000000020,+00,000,000,05,CL-STATE-PROV
0000000125,0000000006,16,0000000006,+00,000,000,05,CL-POST-CODE
```

```
0000000131,0000000013,16,0000000013,+00,000,000,05,CL-COUNTRY
0000000144,0000000013,16,0000000013,+00,000,000,05,CL-TEL
0000000157,0000000040,16,0000000040,+00,000,000,05,CL-EMAIL
```

In an XML-format XFD, the field section description appears as follows:

```
<xfd:fields
  xfd:elementary-items="10"
  xfd:elementary-items-with-occurs="10"
  xfd:total-items="11"
  xfd:total-items-with-occurs="11">
  <xfd:field
    xfd:field-offset="0"
    xfd:field-bytes="197"
    xfd:field-type="16"
    xfd:field-length="197"
    xfd:field-scale="0"
    xfd:field-user-flags="0"
    xfd:field-condition="999"
    xfd:field-level="1"
    xfd:field-name="CLIENT-RECORD"
  />
  <xfd:field
    xfd:field-offset="0"
    xfd:field-bytes="5"
    xfd:field-type="1"
    xfd:field-length="5"
    xfd:field-scale="0"
    xfd:field-user-flags="0"
    xfd:field-condition="0"
    xfd:field-level="5"
    xfd:field-name="CL-CLIENT-ID"
  />
  .
  .
  .
</xfid:fields>
```

5.4 International Character Handling

This section explains how to exchange data between different types of hardware that use different character code sets. Not all machines use the same codes for the characters that are outside of the standard ASCII character set (those whose underlying decimal values are 128 or larger). But you can overcome these differences by setting up a simple character map file and then

pointing to your map file with a runtime configuration variable. Character mapping is triggered by the presence (on the client machine) of a configuration variable that points to the map file. Only single-byte alphanumeric characters are mapped.

Mapping might be essential for sites using the AcuServer product to retrieve and store data on a remote server. Sites that use AcuConnect to launch programs on a remote server might need character mapping to ensure that parameter values returned for their CALL statements are correctly translated. Sites using AcuODBC to access Vision data files on a remote machine may need to provide a map file for the client.

The map file is a simple text file that you create with any text editor. You can choose any name for it that you like. It should contain two values on each line. The first value on the line indicates the decimal or hexadecimal value of the character on the client machine. The second value on the line indicates the decimal or hexadecimal value of the corresponding character on the server machine. Hexadecimal values use the standard “0x” notation. For example:

```
0x90 0xC9
```

maps “E” (acute) in the IBM PC character set to “E” (acute) in the ISO8859-1 character set.

Only those character codes that differ between the two machines need to be included in the map file. This might include characters with a grave accent, acute accent, circumflex, tilde, umlaut, and so forth.

You can check the values of specific characters by using the Windows Character Map accessory in the PC environment, or by referring to your UNIX man pages in the UNIX environment.

As data moves from the client machine to the server, each alphanumeric character that appears in the first position of a map file entry is mapped to the corresponding second character in that entry. A character coming from the server to the client is mapped in reverse: from the second value listed to the first value listed. Each line in the map file thus sets up a one-to-one mapping.

The map is used to translate only single-byte alphanumeric fields. All group items are treated as alphanumeric, so you may want to restructure them to eliminate numeric fields within the group. Data items that are subject to a REDEFINES clause need to be examined with care. For example, if the data looks like this:

```
03 customer-info          pic x(150).
03 customer-detail redefines customer-info.
    05 customer-name      pic x(30).
    05 customer-code      pic 9(20).
    05 customer-address1  pic x(25).
    05 customer-address2  pic x(25).
    05 customer-address3  pic x(25).
    05 customer-address4  pic x(25).
```

then you might need to reorganize or restructure the data definition to ensure that numeric fields are not translated.

5.4.1 Files Required for Translation

International character mapping makes use of three files: your map file of corresponding character codes (explained above), an XFD file that the compiler creates to describe the structure of data files, and a runtime configuration file that points to the location of both the map file and the XFD file. This section provides more details about each of these files. Note that file *names* are not translated, so remote files should use only standard ASCII characters in their file names.

Map File: Your map file of corresponding character codes may be stored on either the client or the server machine. One advantage of storing it on a server is that many clients can access the same map file. If it needs to be changed, a single update benefits all clients. To make a remote map file accessible to a client, you need to run the AcuServer remote file access product on the server.

XFD File: In order to determine whether a data item is alphanumeric, the mapping process for AcuServer and AcuODBC makes use of a data dictionary (also known as an XFD file). (AcuConnect does not require this file.) The XFD file stores the structure of the COBOL records in a format that the runtime can access. An XFD file must be created at compile-time for each program that will use the mapping process. It is created with the “-Fx”

or “-Fa” compile-time option. The XFD files may be stored on either the client machine or the server machine. If they are located on the server, you must have the AcuServer product on that server, to enable client access. If an XFD file cannot be found, no error message is generated, but no character translation occurs.

Runtime Configuration File: The runtime configuration file on the client machine must identify the location of the character map file and the XFD files. Remote name notation can be used to specify remote locations.

Set **XFD_DIRECTORY** to specify the directory for the XFD files (the default is the current directory). AcuConnect does not require XFD files.

Set **DEFAULT_MAP_FILE** to specify the name and location of the character map file.

If you have more than one map file (because you have different types of servers), you can associate a map file with a particular server by using the configuration variable **server_MAP_FILE** to point to a map file on a particular server machine. For example, if you are using AcuServer to access remote files on a machine named sun3, you would use remote name notation to specify the directory that contains the data files. It might look like this:

```
@sun3:/user/acct/inventory
```

Then, use this configuration variable to point to the map file:

```
sun3-map-file @sun3:/user/acct/inventory/map.txt
```

If the map file is local, your value might look like this:

```
sun3-map-file C:\user\utility\map.txt
```

Note: The server specified in the configuration variable name must match the server specified in the remote file name. The map file may be stored on either the client machine or the server machine. If it is located on the server, you must have the AcuServer product on that server, to enable client access.

The runtime looks first for `<server>-MAP-FILE`. If that variable is not set, the runtime looks for `DEFAULT-MAP-FILE`. If that variable is also not set, no translation is done.

6

Programmer's Guide

Key Topics

Handling Files	6-2
Terminal I/O	6-34
Memory Management	6-38
Memory Testing and Error Handling	6-42
Screen Section	6-46
Data Validation	6-55
Exiting From ACUCOBOL-GT Programs	6-56
Multiple Execution Threads	6-57
Working with External Sort Modules (UNIX)	6-70

6.1 Handling Files

Managing files forms the central focus of most COBOL applications. This section discusses the implementation and special features of the three types of files: sequential, relative, and indexed.

ACUCOBOL-GT supports variable-length records in accordance with ANSI standards for *all* file types. A file's records are variable-length whenever any one of these conditions is true:

- The RECORD CONTAINS clause contains the VARYING phrase.
- The RECORD CONTAINS clause contains both a minimum and maximum size.
- There is no RECORD CONTAINS clause but the file's FD specifies more than one record, and those records have different sizes.

A file's records are fixed-length whenever:

- The RECORD CONTAINS clause specifies only a maximum record size.
- There is no RECORD CONTAINS clause, and the file's FD does not specify multiple records having different sizes.

Note: ACUCOBOL-GT automatically closes all of its files if it is killed by the user. However, a power failure, turning off the computer, or issuing a "kill -9" from the console are catastrophic exits. In these cases, ACUCOBOL-GT cannot close its files.

6.1.1 Sequential Files

ACUCOBOL-GT treats sequential files in one of two ways. One form is called *binary sequential*; the other form is called *line sequential*. Note that RECORD SEQUENTIAL is synonymous with BINARY SEQUENTIAL.

Binary sequential files are designed to contain non-ASCII information and are easy to move to foreign systems. A binary sequential file consists of either fixed-length or variable-length records grouped together into blocks.

Variable-length binary sequential records occupy only as much disk space as necessary. If the maximum record size is equal to or less than 65,535 bytes, two bytes indicating record size (VLI) are placed in front of each variable-length record when it is written to disk. If the record size is larger than 65,535 bytes, four bytes are placed in front of each variable-length record. *This two- or four-byte field is not specified in your COBOL program, and non-COBOL programs that access the records need to be aware of the extra bytes.*

All I/O on the physical device is done using the block size, except for the last block. Only that portion of the last block that contains records is read. The default block size is one record. You can change block size with the BLOCK CONTAINS clause.

On VMS systems, binary sequential files are sequential RMS files with fixed-length or variable-length records.

Line sequential files contain variable-sized records. These files are designed to be printed and to be used with other programs, such as editors. The exact form of these files depends on the host system, and thus they should not generally be treated as portable files. On many machines, these files consist of variable-length lines delimited by carriage-control characters. These files should contain only USAGE DISPLAY data so as not to inadvertently introduce stray carriage-control characters.

On Windows systems, a line sequential file contains each data record followed by a carriage-return/line-feed pair. On UNIX systems, these files use just a line feed as the delimiter. On VMS systems, a line sequential file is a sequential RMS file with variable-length records.

ACUCOBOL-GT performs all of its I/O on line sequential files according to the following rules. If the file is blocked (i.e., it has a BLOCK CONTAINS clause), then all I/O is done by blocks. Otherwise, all output is done one record at a time. Input, on the other hand, is internally blocked by **runctl**. This provides an efficient interface while still providing line-by-line control over a print device (which may be needed to do form alignment, for example).

Line sequential files further break into two types, *print files* and *variable-length files*. Print files are similar to variable-length files except that different line-delimiting characters are used. These characters are chosen so that the resulting file will print correctly. Print files may not be opened for INPUT or I-O. Print files are designed to be efficiently printed.

When a line sequential file is read, the default behavior is that any carriage-return, line-feed, or form-feed characters in the record are removed before the record is made available to the program. To change this default behavior, see **section 2.8, “Runtime Configuration.”**

Some line sequential files have automatic blank stripping. This causes records written to the file to have any trailing spaces in the record removed before it is written to the file. All print files have automatic blank stripping. You can specify blank stripping for other line sequential files by choosing one of the following device types in the file's ASSIGN phrase: CARD-PUNCH, CARD-READER, CASSETTE, INPUT, INPUT-OUTPUT, MAGNETIC-TAPE or OUTPUT. See the ACUCOBOL-GT *Reference Manual* for details on the ASSIGN phrase.

When reading from a variable-length sequential file, **runcbl** treats short records in one of two ways. If blank-stripping is specified for the file, then the short records are padded with spaces to fill out to the maximum record size. If blank-stripping is not specified, then the trailing portion of the record is left unchanged (from the most recent contents of the record area).

The runtime system allows a sequential file to be opened for input when the user does not have write-access to the file.

6.1.2 Relative Files

Relative files are generally used to store data where low overhead is required. Records are available by record number. On machines that directly support relative files (such as VMS), relative files are used.

On other machines (Windows, UNIX, and MPE/iX systems), relative files are treated the same as binary sequential files. Each record is physically located at its record number, and writing record 1 and record 1000 causes 999 blank records to be written. Block sizes are ignored, and each I/O uses the record size to determine how many bytes to transfer.

Records may be either fixed-length or variable-length. With variable-length records, if the maximum record size is equal to or less than 65,535 bytes, two bytes indicating record size (VLI) are placed in front of each variable-length record when it is written to disk. If the record size is larger than 65,535 bytes, four bytes are placed in front of each variable-length record. *This two- or four-byte field is not specified in your COBOL program, but other non-COBOL programs that access the records need to be aware of the extra bytes.* Each record is padded to the maximum size so that records can be retrieved randomly in an efficient fashion.

When a record is deleted, the physical record is filled with up to 256 null characters (LOW-VALUES in ACUCOBOL-GT). (This deleted record marker can be changed through the use of the REL_DELETED_VALUE configuration variable. See Book 4, Appendix H.) Note that ACUCOBOL-GT checks only the first 256 characters of the record to determine if it is deleted.

Relative files may be assigned only to disk files.

On many (but not all) systems, the runtime system allows a relative file to be opened for input when the user does not have write-access to the file.

6.1.3 Indexed Files - Vision

Vision is ACUCOBOL-GT's native indexed sequential file system. It provides fast, flexible, and reliable data storage to ACUCOBOL-GT applications in all environments except VMS where ACUCOBOL-GT uses the native RMS file system. Vision handles small or large amounts of data with equal efficiency and provides rapid access to data via optimized indexes, sequential reading, or a combination of indexed START and sequential READ statements. Vision handles all ANSI standard COBOL data types. The Vision file system is described here.

Vision Version 5 and 4

Vision Version 5 and 4 use a dual file format. One file, or *segment*, holds the data records and another segment holds the key information. When a segment approaches the file size limit (configurable up to nearly 2 GB), Vision creates a new data or index segment in which to store more information. The dual file format provides several key benefits:

1. The maximum size of Vision 5 and 4 files is virtually unlimited. Individual data and index segments can be up to 2 GB in size.
2. Separating data and key information provides a higher degree of reliability when files must be rebuilt.
3. Permanent data loss due to memory-related problems is less likely to impact two files.

Vision Version 5 is a superset of Version 4. The rules that apply to Version 4 segment file naming also apply to Version 5.

ACUCOBOL-GT uses Vision Version 5 by default. Vision Version 5 is the most recent and advanced Vision version. Version 5 supports the following capabilities that Version 4 does not:

- records up to 64 megabytes in size
- block sizes up to 8192 bytes
- very large pre-allocate and extension factors
- the ability to recover deleted records (that have not yet been overwritten by subsequent writes)

Vision Version 3

Vision Version 3 files are generated in a single file that contains both the data records and the key information. A separate linked list is maintained and used to rebuild corrupted files. A linked list of deleted records is also maintained so that new records first reuse space held by deleted records before new space is consumed. The single file format supports a maximum file size of 2 GB.

Note: On many (but not all) systems, the runtime system allows an indexed file to be opened for input when the user does not have write-access to the file.

Indexed files may be assigned only to disk files.

6.1.3.1 Segment naming of Vision 4 and 5 files

File segments generated for Vision Version 5 and 4 files can be given customized extensions.

The naming rules described below do not apply to the first data segment. The name of the first segment is set in the same way that the name of any other data file is set. In this discussion, the name of the first segment is referred to as the “regular” name.

Tip: Turning on level 3 or higher file tracing in the debugger (“tf 3”) causes the output of file diagnostics that show the names used for active segments of a Vision Version 5 or 4 file. Also, “vutil -info” displays the names of all segments of a Vision Version 5 or 4 file.

Vision Versions 5 and 4 use two methods to determine the file names of additional index and data segments (after the first segment).

- The first method allows the user to specify a “format” from which Vision generates the names of additional segments.
- The second method (the default) automatically generates the names of additional segments using the base name as a template.

Both methods allow specific generated file names to be overridden.

The first method takes precedence over the second. That is, if circumstances allow Vision to use the first method, it is used. If not, the second method is used.

In both methods, environment variables are used to indicate how Vision is to determine the names of additional segments. You form the names of these environment variables by taking the regular name of the file, converting all the alphabetic characters to upper case, leaving numeric characters alone, and converting all other characters to underscores (“_”). Thus, “gl42.dat” becomes “GL42_DAT”. This name is typically suffixed with another string, depending upon which method you are using, as described below. The runtime looks for these variables in the runtime configuration file. Utilities such as **vio** and **vutil** look for them in the operating system’s environment.

6.1.3.2 Method one: The format method

This method allows the user to specify a format that Vision will use to determine the file names of additional segments. Two formats must be specified: a format for data file extensions and a format for index file extensions. The resulting variables have this general look: *filename_DATA_FMT* and *filename_INDEX_FMT*. Each of these variables must be equated with a format code that includes an escape sequence. The valid escape sequences are defined below.

Example for method one

Suppose the regular name of your COBOL file is “/usr1/gl.dat”. The variables you would use to set the segment naming formats for this file are *GL_DAT_DATA_FMT* and *GL_DAT_INDEX_FMT*.

Each of these variables must be set equal to a pattern that shows how to create the segment names. The pattern shows how to form the base name and how to form the extension for each segment. Part of this pattern is a special character (such as %d) that specifies how the segment number should be represented. Choices include %d (decimal segment numbers), %x (lower case hexadecimal numbers), %X (uppercase hexadecimal numbers), and %o (octal numbers).

For example, setting environment variables *GL_DAT_DATA_FMT=gl%d.dat* and *GL_DAT_INDEX_FMT=gl%d.idx* would result in data segments named /usr1/gl.dat (remember that the first data segment is not affected), /usr1/gl1.dat, /usr1/gl2.dat, and so forth. The index segments would be named /usr1/gl0.idx, /usr1/gl1.idx, /usr1/gl2.idx, and so forth.

Escape sequence definitions

The %d in the values of the *filename-DATA-FMT* and *filename-INDEX-FMT* variables above is a printf-style escape sequence. Most reference books on the C language contain an in-depth explanation of these escape sequences, and UNIX systems typically have a man page (“man printf”) that explains them in detail. Here are the basics:

- “%d” expands into the decimal representation of the segment number.
- “%x” expands into the hexadecimal representation (with lower case a-f) of the segment number.
- “%X” expands into the hexadecimal representation (with upper case A-F) of the segment number.
- “%o” expands into the octal representation of the segment number.
- You can add leading zeros to the number (to keep all the file names the same length) by placing a zero and a length digit between the percent sign and the following character. “%02d” would result in “00”, “01”, “02”, and so forth when expanded.
- To embed a literal “%” in the file name, use “%%”.

The escape sequence can be positioned anywhere in the file name, including the extension.

Note: The runtime checks for these segment naming variables in the runtime configuration file. Utilities such as **vutil** and **vio** check in the operating system’s environment.

6.1.3.3 Method two: The default method

The default method uses the regular file name to determine the file names of additional segments. This method stores the segment number in the extension of the file name. If you use the extension of the file name to distinguish files that are otherwise named the same, you should not use this method.

This method takes the regular file name and removes the extension (if any) from that name. It then adds “.vix” to generate the name for the first index segment. Subsequent index segments are named with “.v01” through “.vff” (hexadecimal representation) for the first 255, and “.v0100” through “.vfff” for segments 256 through 65536. The data segments are named using a similar numbering scheme, but use “d” instead of “v” before the segment number. You should avoid using file name extensions that can be considered as “d” or “v” followed by a hexadecimal number. For example, the extension “.dae” is not safe, because that is the name of the 175th data segment. The common extension “.dat” is safe because “t” is not a hexadecimal digit.

6.1.3.4 Overriding individual segment names

You can override an individual generated segment file name by setting an environment variable named by the generated file name of the segment (converted as described above) to the full path of the desired file name. As an example, suppose the regular name of your file is /usr1/gl.dat, and you have `GL_DAT_DATA_FMT=gl%d.dat` set, but you want to place the second data segment on /usr2. Setting `GL1_DAT=/usr2/gl1.dat` will override the originally determined name. This feature works with both methods. Using the file names generated by the default method as environment variables (converted as described above) works, too.

6.1.3.5 Selecting the Vision version

You can control what Vision file format is applied to new Vision files through the `V_VERSION` and `filename_VERSION` configuration variables. The value of `V_VERSION` specifies the default format for new Vision files. The `filename_VERSION` variable allows you to set the file format on a file-by-file basis. *Filename* is replaced by the base name of the file (the filename minus directory and extension). These variables take an integer value that corresponds to the Vision Version that you want applied (5, 4, 3, 2). For more information about these configuration variables, see Appendix H of Book 4, *Appendices*.

6.1.3.6 Keys

Vision files allow up to 120 keys for sorting. One key is the primary key, all others are alternates. In a record, a key may not occupy physical space that exceeds the minimum record length. Thus, a 10-byte key cannot occupy positions 20-29 in a record with a minimum record length of 28.

You can specify whether duplicates are allowed for the primary key and for each alternate key. If duplicates are allowed for a particular key, it is possible to write a record whose key fields contain exactly the same data as the key fields of an existing record. In this case, the records are stored in chronological order.

Note: It is generally recommended that you not allow duplicate primary keys unless the file is processed only in ACCESS SEQUENTIAL mode. Allowing duplicate primary keys in files that are processed in DYNAMIC and RANDOM access modes could generate unanticipated results. The rules that govern the treatment of duplicate primary key values are presented in General Rule 16 of **section 4.3.1, “File-Control Paragraph”** in Book 3, *Reference Manual*.

Every alternate key causes significant additional overhead. (Keys have the least amount of overhead when duplicates are allowed to occur.) The keys for a data file are automatically stored in a compressed form.

A key may be a contiguous part of the records, or it may be split into as many as 16 segments. If you are compiling for compatibility with versions earlier than Vision Version 4, you can have no more than six segments. You must use Vision Version 4, or later, if you want more than six segments.

Suppose you have the following record structure in a file called AJAX-SUPPLIES:

```

01 CUSTOMER-RECORD
   03 CUSTOMER-ID          EC96.
   03 CUSTOMER-NAME       EC99V9.
   03 CUSTOMER-NAME       ECX(3).
   03 CUSTOMER-CONTACT   ECX(3).

```

To use CUSTOMER-NAME as the primary key, you would use the syntax shown in the last line below:

```
FILE-CONTROL.  
  SELECT AJAX-SUPPLIES  
  ASSIGN TO DISK "INDEX.DAT"  
  RECORD KEY IS CUSTOMER-NAME.
```

If data elements are contiguous and defined in the order that would be used for sorting, they may be grouped and defined together as a key. For example, suppose you wanted to use CUSTOMER-BALANCE, CUSTOMER-NAME as an alternate key. Because these two fields are contiguous and are defined in the same sequence they will be used for sorting, the most efficient way to define the alternate key is to establish a group item that includes both fields. For example:

```
01 CUSTOMER RECORD  
  03 CUSTOMER-NO                PIC 9(6) .  
  03 CUSTOMER-BALNAME .  
    05 CUSTOMER-BALANCE        PIC S9(9)V99 .  
    05 CUSTOMER-NAME          PIC X(30) .  
  03 CUSTOMER-CONTACT          PIC X(30) .
```

Then, to define CUSTOMER-BALNAME as an alternate key, you would use the syntax shown in the last line below:

```
FILE-CONTROL.  
  SELECT AJAX-SUPPLIES  
  ASSIGN TO DISK "INDEX.DAT"  
  RECORD KEY IS CUSTOMER-NAME  
  ALTERNATE KEY IS CUSTOMER-BALNAME.
```

Suppose now that you want to define a sort sequence that uses fields that are *not* contiguous, or are defined in a *different order* from the sorting order. In this case, you could either:

- move the fields around, or duplicate them, so that they are contiguous and are in the same sequence in which they will be used for sorting
- define a *split key*

Split keys allow you to specify up to *16 segments of data elements* as the components of a key. (Note that if you compile for compatibility with versions earlier than Vision Version 4, you can have no more than six segments.) The data segments need not be contiguous and need not be listed

in the order they appear within the record. The composite length of a split key cannot exceed 250 bytes, and no key can be defined beyond the minimum record length.

For example, to define an alternate key consisting of CUSTOMER-BALANCE, CUSTOMER-NAME, and CUSTOMER-NO, use the syntax shown in the last two lines below:

```
FILE-CONTROL.
  SELECT AJAX-SUPPLIES
  ASSIGN TO DISK "INDEX.DAT"
  RECORD KEY IS CUSTOMER-NAME
  ALTERNATE RECORD KEY IS CUSTOMER-BALNAME
  ALTERNATE RECORD KEY IS BAL2-KEY =
    CUSTOMER-BALANCE, CUSTOMER-NAME, CUSTOMER-NO.
```

In this example, BAL2-KEY is a user-defined word and is the name you would use in your READ and START statements. Note that BAL2-KEY is *not* defined in Working-Storage. This is the *only* definition of the key.

6.1.3.7 Other Vision features

Block size

Vision files have a block size (physical record size) of between 512 and 8192 bytes (in 512-byte increments). The block size specifies how large a node is in the Vision index tree. The block size of Vision Version 2, 3, or 4 files is restricted to 1024 bytes. The block size is specified by the BLOCK CONTAINS clause (see [section 5.4.4, “BLOCK CONTAINS Clause”](#) in Book 3, *Reference Manual*). If the BLOCK CONTAINS clause is omitted, the file will have 512-byte blocks.

Each index node holds a number of key entries and pointers to other nodes or record data. The larger the block size, the more key entries each node can store. Also, larger block sizes produce shorter index trees. However, larger index nodes can cause Vision to spend more time performing linear searches of each node to find the desired key entry. Therefore, it is important to determine the optimal block size for your particular key and data structure. If you have very large keys, a larger block size can help keep the tree height manageable. If you have smaller keys, a large block size will produce a short tree, but performance may be lost in the time it takes Vision to scan each of the large nodes.

Some experimentation will help you determine the optimal block size for your application and hardware. In the past, Vision has used 512- and 1024-byte block sizes. Users have reported good results with these values. If your keys are small or medium size, try using the 512 or 1024 block sizes. If you have a file with large keys and you are using Vision 5 files, you might want to experiment with a larger block size.

Compression and encryption

Vision can optionally compress and/or encrypt records. Record compression uses a simple run-length compression algorithm. Encryption uses a byte transformation algorithm that is unique to every byte in the file. Encrypted files may not have records extracted by the Vision utility program **vutil**. Records are stored internally in the least amount of space required. Furthermore, they are packed together and span block boundaries, so no disk space is wasted. Compression is discussed in more detail in section 6.1.6.1, “Compression.”

User count

Vision maintains a *user count* for each file. This count is normally zero. When a file is opened for update, the user count is incremented; when the file is closed, the user count is decremented. The user count is thus the number of currently updating processes for the file. If a program dies catastrophically, however, the user count will not get decremented. (For a definition of “catastrophic,” see **section 6.7, “Exiting From ACUCOBOL-GT Programs.”**) This results in the count never reaching zero. Thus, if the value is non-zero when there are no active users, it indicates a catastrophic program failure and suggests that corrective action may need to be taken. At the very least, the file should be checked for integrity, but depending on the program that died, perhaps more significant action should be taken. Basically a non-zero user count indicates that someone knowledgeable about the system should intervene and ensure that everything is okay. This can be used as an early warning system to head off some problems. Note that a non-zero user count is not a fatal error to Vision. It is used only as an indicator of potential problems.

6.1.4 Record Locking

Record locking for indexed and relative files is used for all machines. Currently, record locking is not supported for sequential files. Only files opened in the I/O mode will actually lock records; input-only files will not lock records they read. Record locking occurs in one of two modes: *automatic* (the default) and *manual*. In automatic mode, a record is locked when it is read (unless the WITH NO LOCK or ALLOWING UPDATES phrase of the READ statement is used). In manual mode, records are locked only when WITH LOCK is specified on the read statement. Both modes support single and multiple record locks. When single record locks are used (the default), the currently locked record is released whenever any I/O statement is executed for the file. When multiple record locks are used, records become unlocked only when an UNLOCK or CLOSE statement is executed for the file (or a COMMIT is executed for a transaction file).

It is possible to get a record locked condition when you are doing a READ, REWRITE, or DELETE on a file.

On most machines, a READ performed without locking will not get a record-locked condition even if another process has the record locked. This allows report programs to run without hindrance. Note, however, that on some older UNIX machines, non-locking READs *will* receive the record-locked condition if another process has the record locked. This is because on these machines, UNIX enforces the record locks. Most UNIX systems do not enforce record locks (the current system V standard does not allow for it) so this should not continue to be a problem for long. PC-based environments (i.e., LANs, Windows) also enforce record locks, introducing a portability issue. For unenforced locks in a PC-based environment, you should use indexed files rather than relative files.

runcbl behaves in one of two fashions when a locked record is encountered. Normally, the return is immediate and the file status variable is set to the locked condition. When this happens, the I/O fails. Unlike most errors, the current record pointer is not changed by a locked condition. This allows multiple READ NEXTs to be performed in a loop until the lock condition disappears, without losing track of the current record position.

In RM/COBOL compatibility mode, however, this rule changes if the file does not have a Declarative associated with it. In this case, locked records on REWRITE and DELETE statements are handled as described above. On

READ statements, however, **runcbl** automatically waits until the record becomes unlocked. This is identical to the behavior of RM/COBOL version 2. The programmer should beware when using this technique, however. Deadlocks can occur if two programs end up waiting for each other to unlock a desired record. For this reason, we recommend against using this technique.

When ROLLBACK is enabled in a file's FILE-CONTROL entry, record and file locking rules are extended for that file. Every record updated as part of a transaction is locked until that transaction is committed or rolled back. The COMMIT and ROLLBACK verbs remove these locks. For more information, see **section 5.1.4, "Extended Locking Rules."**

6.1.5 Device Locking Under UNIX

On UNIX machines, handling devices poses a special problem. The difficulty is that UNIX does not in any way limit access to a device, so that it is possible for two programs to intermix reports on the same printer. This is usually not a problem with printers, because the output is usually directed to the UNIX spooler. However, for special-form printing, or writing to other types of devices, it is necessary to write directly to the device.

On UNIX systems, **runcbl** can control concurrent access to a device by automatically performing file locking for all files identified as devices. You must enable this feature if you want to use it. When it is enabled, **runcbl** will create a *lock file* whenever a device file is opened. If the lock file exists when the open is made, then the open will fail and a file-locked condition will be returned to the program. When an open device file is closed, its lock file will be removed.

You identify a device to **runcbl** by using the "-D" flag in its name. For information about this, see **section 2.9, "File Name Interpretation."** The lock file will be called "LOCK.name" where *name* is the base file name of the UNIX device (for example, "lp" or "mt0").

You enable automatic device locking by specifying a lock directory. Set the configuration variable LOCK_DIR to the desired directory.

For example, if you wanted to enable device locking and place the lock files in the directory “/usr/locks”, you could use the following configuration variable:

```
LOCK-DIR  /usr/locks
```

Note: If a program that has established a device lock dies in a catastrophic fashion (e.g., a power failure), the lock file will be left around and should be removed manually.

Also note that enabling device locking will not prevent the UNIX spooler from printing to the device. When you are using a printer as a special-form printer, the spooler should first be disabled (see the “disable” UNIX command) and then the program should be run that will print the special forms. It will prevent other special-form printing programs from using that printer. Programs that print through the spooler, meanwhile, will continue to run, because the UNIX spooler accepts queued reports when disabled. After the special forms are printed, the spooler can be enabled and the queued reports will start printing.

6.1.6 Indexed File Considerations

This section describes the impact of ACUCOBOL-GT’s special indexed file features. It covers how they work and when they should (and should not) be used.

6.1.6.1 Compression

File compression can be used on indexed files to save disk space. The Vision file system supports compression, but not all file systems do. Compression is enabled by specifying the WITH COMPRESSION phrase in the ASSIGN clause of a file’s SELECT statement. Compression must be specified when the file is initially created to have any effect. However, the **vutil** “-rebuild” option allows you to apply or remove compression during the file rebuilding process. See **section 3.3.3, “Rebuilding Files.”**

File compression uses a simple run-length compression scheme. This replaces “runs” of identical bytes with a shorter sequence. Files using compression may contain any type of data.

Some files will compress better than others. Generally speaking, files that contain text compress the best due to repeated space characters. Results can vary significantly, however. Experimentation is the best way to tell how much space may be saved.

Each compressed record usually retains some extra, unused space for future expansion. This is advisable especially if the records are frequently changed. You can specify via a *compression factor* how much of the space saved by compression should be retained to allow for future growth. When no compression factor is specified, WITH COMPRESSION uses the default compression factor (70). The following paragraphs explain how the factor is used.

A compression factor other than the default may be specified via the COMPRESSION CONTROL VALUE IS clause in the SELECT statement. The factor must be a numeric literal within the range zero (no compression) to 100 (maximum compression). A factor of one (1) causes Vision to examine the **COMPRESS_FACTOR** configuration variable. If COMPRESS_FACTOR is not set, the default compression factor is used (70).

For factors from two through 100, the factor is considered to be a percentage. It specifies how much of the space saved by compression is actually removed from the record. For example, suppose an 80-byte record is compressed to 30 bytes. Then the compression factor is used to determine how much of the 50 bytes of saved space is to be removed from the record. A compression factor of 70 would mean that 70% of the 50 bytes (35 bytes total) will be removed. This leaves 15 bytes for future expansion, and results in a compressed record size of 45 bytes (30 compressed size plus 15 extra for growth). The larger the compression factor, the more of the saved space is removed. A compression factor of 100 removes all saved space and is advisable only if the file is rarely updated.

An alternate way to specify the compression factor is to set the COMPRESS_FACTOR configuration variable. COMPRESS_FACTOR is used when the COMPRESSION CONTROL VALUE IS clause is either omitted or set to a value of one. See the entry for **COMPRESS_FACTOR** in Book 4, Appendix H. As noted earlier, the compression factor for a file is established when the file is created. Subsequent changes to COMPRESS_FACTOR do not affect existing files.

The selection of the compression factor should be based on the amount of updating that the file undergoes. If rewrites and deletes are rarely or never done on the file, then a high compression factor is most efficient. We recommend 100 for files that are rarely updated, 70 for average files, and 50 (or less) for files that are frequently updated.

6.1.6.2 Mass update

The MASS-UPDATE option of the OPEN verb can provide significant performance benefits under some circumstances. Several issues come into play, however, when you are deciding whether or not to use MASS-UPDATE. Currently, the MASS-UPDATE clause affects only the systems that use Vision.

Normally, when Vision updates a file, it immediately writes all of the changed information to disk. This is done for two reasons: to allow current information to be accessed by other concurrent processes, and to ensure that the file will be accurate should the program die suddenly without closing the file (e.g., when a machine's power goes out or the operating system crashes). Note that a Vision file is really only at risk of being damaged during an update to the file.

The MASS-UPDATE option changes this strategy. It allows Vision to retain information in memory until the file is closed. This allows Vision to be much more efficient, particularly on Windows systems. However, using this option means that the file is at risk from the time the first update is made until the time the file is closed. Should the machine die during this period, the file will almost certainly be corrupt. To mitigate this issue, Vision writes enough information to disk to ensure that the file can be rebuilt using **vutil**.

The MASS-UPDATE option also requires that the entire physical file be locked against other updaters, because the disk version of the file is not always accurate. This somewhat limits the opportunities in which MASS-UPDATE can be used.

Generally, programs might use MASS-UPDATE if they heavily update a file. For many such programs, the fact that the file is at greater risk is not really an issue. For example, many posting programs cannot recover from an incomplete run. This is because the program cannot tell where it left off in the process. This is particularly true for programs that update several files at once, because it is usually not clear which file got updated last. For these

programs, it is usually necessary to go to a backup of the affected files when the program dies. These programs are obvious candidates for MASS-UPDATE because it does not matter if the files are corrupt after a program failure, since they are just going to be restored from backup. Furthermore, these programs benefit the most from MASS-UPDATE because they do a lot of updating.

Interactive programs, however, make poor candidates for MASS-UPDATE. Usually the volume of updates is low (at least for the time frame the program runs in). Furthermore, interactive programs are often killed or left running overnight by their operators, thus increasing both the risk to the file and the inconvenience of the file lock that MASS-UPDATE implies.

To summarize, MASS-UPDATE is appropriate for programs where the implied file lock is useful, the volume of updates is large, and where a system failure would usually require special attention for recovery (either restoring from backup or rebuilding the files).

Note: For convenience when you are converting programs written with other COBOL compilers, ACUCOBOL-GT can treat files opened WITH LOCK as if they were opened with MASS-UPDATE. This is controlled by the **MASS_UPDATE** runtime configuration variable. Configuration variables are described in Appendix H.

6.1.6.3 Bulk addition mode for Vision

When Vision writes a record to a file, its normal algorithm is to first add the record, then add the primary key, then add the first alternate key, and so on until all the keys have been added. This is the algorithm one expects and is generally required to correctly support the WRITE verb.

Vision also has another technique that it can use to add records to a file. This technique does not write the keys to the file when it adds the record. Instead, it adds many records to the file first, then it gathers all the primary keys for those records and adds them, then it gathers all the first alternate keys and adds them, and so on. By adding many keys at once, Vision can be much more efficient in its handling. This has two benefits:

1. The time it takes to add a large number of records using this technique is generally much smaller than using the normal technique; and

2. The resulting file is more efficiently organized, with all of the blocks associated with a particular key near each other on disk.

This technique of adding records to a file is called “bulk addition” mode. It is available only for Vision files. It may be used with any format Vision file.

You can use bulk addition mode in your programs. This is most useful in programs that add a large number of records to a file at once. It is less useful in programs that do operations other than WRITE on a file, or programs that do not write many records. Some typical applications for bulk addition mode are:

- Programs that load externally supplied data into a file. For example, a program that adds to an indexed file from the data contained in a sequential file.
- File conversion programs, such as programs that change a record format. These programs read from one file and write to a new one.
- Archiving programs.
- Programs that post transactions to permanent log files.

While bulk addition mode has fairly specialized uses, its benefits are high in these cases.

Bulk addition mode can provide substantial performance improvements over other techniques, including MASS-UPDATE mode. These improvements become more noticeable as the file grows and in files with a large number of alternate keys. For files with few records, bulk addition mode can be slightly less efficient than the normal WRITE technique.

Using bulk addition affects some of the standard COBOL file handling rules. These are described in a separate section below.

Using bulk addition

To use bulk addition on a file, open that file with the BULK-ADDITION phrase. For example:

```
OPEN OUTPUT MY-FILE FOR BULK-ADDITION
```

A file opened for bulk addition is locked for exclusive use by the program. This is necessary because the file becomes internally inconsistent when the first WRITE occurs and does not become consistent again until the file is closed.

You may use BULK-ADDITION with the OUTPUT, EXTEND and I-O open modes.

If you specify BULK-ADDITION for a file other than a Vision file, the effect is the same as if you had specified MASS-UPDATE instead.

Vision allocates a memory buffer for each file opened for bulk addition. The size of this buffer is controlled by the **V_BULK_MEMORY** configuration variable. The default size of this buffer is 1 MB. The default size is fairly large because it is assumed that only a few files will be open for bulk addition on a system at any one time. If this buffer cannot be allocated, the OPEN fails with a status indicating inadequate memory.

When a WRITE occurs for a file opened for bulk addition, only the record is actually written to the file. The keys for the record are not written out until later. When an operation occurs on the file other than a WRITE, that operation is suspended and all of the records with missing keys have their keys added. After this completes, the suspended file operation resumes.

The process of adding keys uses the memory buffer allocated when the file was opened. The algorithm is to fill the buffer with keys, sort them, and then add them to the file. This is repeated until all the missing keys have been added. The larger the memory buffer, the more efficient this process is.

When adding records to the file, Vision always places the records at the end of the file when using bulk addition. It does not re-use deleted record areas in this case. It does this to make the process of gathering the missing keys efficient. If you need to recover the deleted record disk space, you can rebuild the file with **vutil** to do this.

Note: Any operation on the file other than WRITE will trigger the addition of the keys. Ideally, this operation will be the final CLOSE of the file. In this case, all of the newly added records are keyed at once and efficiency is maximized. In the worst case, each WRITE is followed by some other operation (such as a READ). In this case, each record is keyed individually. This is less efficient than just updating the file normally.

This process of delaying the keying of the records has several effects besides improving performance. It affects the rules of COBOL, especially the handling of duplicate keys. In addition, it makes it harder to report the program's progress to the user, since much of the program's time may be spent in a single COBOL statement (the file's CLOSE, for example, may take the majority of the program's running time as it adds all the missing keys). These issues are discussed in the following sections.

Effect on COBOL rules

When you open a file for bulk addition, the regular rules of COBOL file handling are changed for that file, until that file is closed. The following changes apply:

1. File status "02" (record written contains an allowed duplicate key value) is never returned by WRITE.
2. File status "22" (record not written because it contained a disallowed duplicate key value) is not returned by WRITE. See the next section for a discussion of how illegal duplicate keys are handled.
3. File status "24" (disk full) may occur on file verbs that normally cannot produce this status. This occurs because the verb (for example, READ) triggered writing of the keys to the file and the disk became full while doing this.
4. Records may be rejected as having illegal duplicate keys that would not normally be rejected. The circumstance under which this occurs is described in the next section.

Duplicate key handling

Because keys are not written at the time a new record is written, the WRITE statement never gets a duplicate key error status (status 22). When you are using bulk addition, illegal duplicate keys are handled in a different manner.

When the keys are added to the file, any illegal duplicates are detected then. Should a record be found that contains an illegal duplicate key value, that record is deleted. Your program is informed of this only if it contains a valid declarative for the file. If there is no declarative available, the record is quietly deleted. Otherwise, the file status data item is set to “22”, the file’s record area is filled with the contents of the rejected record, and the declarative executes. When the declarative finishes, the file record area is restored to its previous contents so that it contains the correct data when the suspended file operation resumes.

When the file’s declarative executes in this way, the program may not perform any file operations in the declarative. This is because the program is in the middle of doing a file operation already, the one that triggered the addition of the keys. In addition, the declarative may not start or stop any run units (including chaining), nor may it do an EXIT PROGRAM from the program that contains the declarative. Finally, note that the declarative runs as a locked thread — no other threads execute while the declarative runs.

You can configure Vision to write any rejected records to a file. This gives you a way to log the rejected records even though you may not perform a file operation from within your declarative. To create this log, set the **DUPLICATES_LOG** configuration variable to the name of the file in which you want to store the records. Vision will erase this file first if it already exists. You must use a separate log file for each file opened with bulk addition. You can do this by changing the setting of DUPLICATES_LOG between OPEN statements. For example:

```
SET ENVIRONMENT "DUPLICATES-LOG" TO "file1.rej"  
OPEN OUTPUT FILE-1 FOR BULK-ADDITION  
  
SET ENVIRONMENT "DUPLICATES-LOG" TO "file2.rej"  
OPEN EXTEND FILE-2 FOR BULK-ADDITION
```

If DUPLICATES_LOG has not been set or is set to spaces, then no log file is created.

In addition, the duplicate-key log file may not be placed on a remote machine using AcuServer. The log file must be directly accessible by the machine that is running the program.

Any record that Vision rejects due to an illegal duplicate key value is written to the log file. The format of the file is a binary sequential file with variable-size records. You can read this file with a COBOL file that has the following layout:

```

FILE-CONTROL.
    SELECT OPTIONAL LOG-FILE
    ASSIGN TO DISK file-name
    BINARY SEQUENTIAL.

FILE SECTION.
FD    LOG-FILE
      RECORD IS VARYING IN SIZE DEPENDING ON REC-SIZE.
01    LOG-RECORD.
      <<indexed record layout goes here>>

WORKING-STORAGE SECTION.
77    REC-SIZE                PIC 9(5).

```

If no duplicate records are found, the log file is removed when the Vision file is closed.

There is an unusual circumstance that can cause a file opened for bulk addition to reject a record that would not have been rejected if the file had been opened normally. This occurs only when the file has at least one alternate key that does not allow duplicates. This happens due to the changed order in which the keys are written to the file.

Consider a case where a file has two numeric keys, the primary key and one alternate that does not allow duplicates. Now suppose the following three records were written to this newly created file:

Primary key	Alternate key
1	1
2	1
2	2

In a file opened normally, the first record would be written to the file, the second record would be rejected because of an illegal duplicate on the alternate key, then the last record would be written. The result would be a two-record file, the records (1,1) and (2,2).

If the file is opened for bulk addition, the three records are added, then the primary keys are added, then the alternate keys are added. First the three records are added. Then the first and second record's primary keys are added. The third record's primary key is rejected because it duplicates the second record's key. The third record is removed as a result of this. Then the alternate keys are processed. The first record's key adds fine. The second record's key is rejected because it is a duplicate, and the second record is removed. The third record's alternate key is not processed because that record has already been removed. The result is a one-record file, the record (1,1).

To summarize, as a result of bulk addition, you may end up with records rejected because of the duplicate key conflict with other (eventually) rejected records and not necessarily with any accepted records.

This difference would not occur if the keys were added "row-wise" instead of "column-wise," but doing so would sacrifice much of the efficiency gained by bulk addition mode.

In most practical applications, this scenario is not very likely. If need be, you can adjust for this difference by logging the rejected records and then trying to add them to the file normally after leaving bulk-addition mode. The second attempt at writing out the records will still reject the records with illegal duplicates, but take any records that conflict only with other rejected records.

Because of the various issues surrounding illegal duplicate key values, it is best to use bulk addition in cases where illegal duplicates are rare. Processing records with a great many illegal keys significantly reduces the performance benefits of using bulk addition.

Progress reporting

Programs that use bulk addition are frequently the types of programs where it is desirable to report the program's progress to the user. For example, a program that reformats a file would typically display its percentage complete

while running. However, a single COBOL statement may represent the majority of the running time, so progress reporting is difficult to do. The file reformatting program, for example, could spend 20% of its time writing out the reformatted records and 80% of its time in the CLOSE statement while the records are having their keys written.

You can use a special declarative section to do progress reporting. This section is called directly by Vision in a periodic fashion while the keys are being added to the file. To create the declarative, use the following form of the USE statement:

```
USE FOR REPORTING ON file-name.
```

Vision executes this section at regular intervals. This reporting period is approximately once for each percentage point completed per key.

Because the declarative is called from within a file operation, the declarative section's code may not execute any file operations. In addition, the declarative may not start or stop any run units (including chaining), nor may it do an EXIT PROGRAM from the program that contains the declarative. Finally, note that the declarative runs as a locked thread — no other threads execute while the declarative runs.

To determine how far along Vision is in adding the keys, you can call the library routine "**C\$KEYPROGRESS**". You pass this routine a single parameter, which has the following layout:

```
01 KEYPROGRESS-DATA, SYNC.
   03 KEYPROG-CUR-KEY   PIC XX COMP-N.
   03 KEYPROG-NUM-KEYS  PIC XX COMP-N.
   03 KEYPROG-CUR-REC   PIC X(4) COMP-N.
   03 KEYPROG-NUM-RECS  PIC X(4) COMP-N.
```

A copy of this group item can be found in the COPY library "keyprog.def".

When C\$KEYPROGRESS returns, the group item is filled with current data. The individual items contain the following:

- **KEYPROG-CUR-KEY** — this is the current key being worked on by Vision. The primary key is key "1", the first alternate is key "2", and so on.
- **KEYPROG-NUM-KEYS** — this is the total number of keys in the file.

- **KEYPROG-CUR-REC** - this is the number of the last record written for the current key, ranging from 1 to the total number of records to write.
- **KEYPROG-NUM-RECS** - this is the total number of records to be keyed.

You may report this information in any fashion. If you want to report the actual percentage complete, the formula is the following:

```
total-operations = keyprog-num-recs * keyprog-num-keys

operations-complete =
    (keyprog-cur-key - 1) * keyprog-num-recs + keyprog-cur-rec

pct-complete =
    (operations-complete / total-operations) * 100
```

That formula computes the percentage complete for just adding the keys. If you want to treat the original record writes and the adding of the keys in a single percentage scale, the formula is slightly different:

```
total-operations = keyprog-num-recs * (keyprog-num-keys + 1)

operations-complete =
    keyprog-cur-key * keyprog-num-recs + keyprog-cur-rec

pct-complete =
    (operations-complete / total-operations) * 100
```

Here is an example of a typical reporting declarative:

```
77  PROGRESS-BAR-1          HANDLE OF FRAME.

DECLARATIVES.
MYFILE-REPORTING SECTION.
    USE FOR REPORTING ON MYFILE.
MYFILE-REPORT.
    CALL "C$KEYPROGRESS" USING KEYPROGRESS-DATA
    MODIFY PROGRESS-BAR-1, FILL-PERCENT =
        (((KEYPROG-CUR-KEY - 1) * KEYPROG-NUM-RECS
        + KEYPROG-CUR-REC) / (KEYPROG-NUM-RECS
        * KEYPROG-NUM-KEYS)) * 100).
END DECLARATIVES.
```

Note: As mentioned above, the progress reporting code in the Declaratives Section tracks only the bulk addition of keys to the file. To also indicate the time spent writing records, similar code should be added to the corresponding WRITE statements in the Procedure Division.

Bulk addition and AcuServer

You may not use bulk addition mode with files that you are accessing via AcuServer. If you attempt to do so, AcuServer will open the file in MASS-UPDATE mode instead.

Programs that are appropriate targets for bulk addition mode are generally much more efficient when run directly on the server. You can arrange to do this directly by manually starting the job on the server, or you can use AcuConnect from a workstation to remotely start the job on the server.

In addition, the duplicate-key log file may not be placed on a remote machine using AcuServer. The log file must be directly accessible by the machine that is running the program.

Using bulk addition with transactions

You may use bulk addition for files that use transaction management. No transaction management rules are affected by bulk addition.

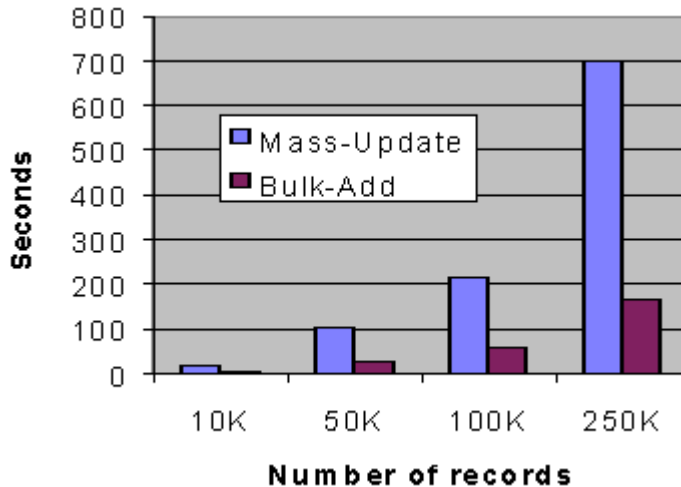
The START TRANSACTION, COMMIT and ROLLBACK verbs are not treated as operations that trigger the bulk addition of keys. However, a ROLLBACK can cause the bulk addition of keys if it has to delete or rewrite a record as part of its operation. Note that a file's declaratives will not execute as part of a ROLLBACK process. This applies to both the error handling declarative and the progress reporting declarative.

Performance tips

Using bulk addition can provide very substantial performance gains in appropriate cases. Generally, it is best used when you are adding a large number of records to a file and has a more noticeable effect on files with a large number of records.

The following chart shows one set of execution times for creating a new file with eight 10-byte keys generated in random order and a 130-byte record size. These were run on a Windows 98 machine.

Note that the times will vary widely from machine to machine and between different file organizations; use these numbers just as a comparison between techniques:



Notice how the run times for MASS-UPDATE mode rise at a much steeper rate than those for bulk addition mode as the number of records grows.

You get best performance from bulk addition when you maximize the number of records being keyed at once. This means that you want to WRITE as many records as possible to the file without performing any other intervening operations on the file.

There are two configuration variables that have an important effect on the bulk addition performance. The first of these is **V_BUFFERS**, which determines the number of 512-byte blocks in the Vision cache. Besides having its usual caching effect, the Vision cache is especially important when you are doing bulk addition, because the cache is used to gather file blocks together into larger groups that are written out in a single call to the operating system. While the cache always does this, the bulk addition algorithm tends

to produce very large sets of adjacent modified blocks, which can all be written out at once. By increasing the cache size, you can increase the number of blocks written out at once.

For this reason, you should use a cache size of at least 64 blocks. Note that this is the default cache size for most systems. If memory is plentiful, then a cache size of 128 or 256 blocks is recommended. You can go higher if you want; however, there is usually little benefit seen after about 512 blocks (256K of memory).

The other important factor is the memory buffer used to hold the keys (**V_BULK_MEMORY**). Unlike V_BUFFERS, this is most useful when it is large. The default size is 1 MB. Larger settings will improve performance when you are adding many records to a file. Essentially, this buffer is used to hold record keys. The more keys that it can hold, the better the overall performance. For runs that will write out between 250,000 and 500,000 records, a setting of 4 MB generally works well. For more than 500,000 records, we recommend at least an 8 MB setting. Be careful that you do not set this too large, however. If you set it so large that the operating system must do significantly more memory paging to disk, you could lose more performance than you gain. You will need to experiment to see which setting works best for your system.

Finally, the process of removing records due to illegal duplicate keys is expensive. You should try to arrange it so that bulk addition is used in cases where illegal duplicate keys are rare.

Summary

Using bulk addition can provide very significant performance gains in certain cases. These cases involve writing a very large number of records to a file. In order to optimize performance, certain rules of COBOL are changed and some other restrictions apply. Here are the key points to remember:

1. Files open for bulk addition are locked for exclusive use.
2. WRITE does not add keys to the file. The keys are added when some other file operation occurs.

3. Duplicate record errors (status “22”) are not returned by WRITE. Instead, they are reported to the file’s declarative procedure only during some other file operation.
4. The declarative must not perform any file operations or start or stop any run units when processing status “22”.
5. You may log records rejected due to illegal duplicated keys by setting the option `DUPLICATES_LOG` to the name of the desired log file.
6. Disk space occupied by deleted records is not re-used when you are adding records with bulk addition.
7. You can report on the progress of adding the keys in a `USE FOR REPORTING` declarative. This declarative may not perform any file operations or start or stop any run units.
8. You may not use bulk addition on files you are accessing via AcuServer. Use AcuConnect or some other technique to start the job directly on the machine with the files.

Avoid doing an abnormal shutdown on a job running with bulk addition. If the job aborts without completing its “close” operation, the file will almost certainly have keys missing and need to be rebuilt.

6.1.7 Performance Considerations

File performance varies considerably from machine to machine and from program to program. This section discusses general measures that can be taken to improve file handling performance for most situations.

Generally speaking, sequential files are fastest and indexed files are slowest. Relative files are usually in between in performance, but are generally close to sequential files in speed.

When designing indexed files, you should try to keep the size and number of keys to a minimum. Each key added to a file significantly increases the processing time required. The key size is important because it affects how many keys can be stored in one disk block. If you are faced with a decision that trades off making a new key or making an old key larger, it usually is better to make the old key larger.

One important and often overlooked aspect of file performance is file locking. Generally speaking, the more restrictive the access to a file, the more efficiently that file can be processed. On some machines, the difference can be quite dramatic. For example, under RMS, writes to a sequential file are usually at least five times faster if that file is locked for exclusive use. If this is a report file, there is no reason not to lock the file, and the report program will run much faster if you do so.

As a general rule of thumb, you should lock files as restrictively as you can, given the needs of your application. Note that if you have a file that is read by many programs, but rarely or never written to, you can open the file for INPUT ALLOWING READERS. This allows many users to access the file, but also tells the runtime system that it is not being updated by anyone. The runtime system can use this knowledge to buffer the file more effectively. One common form of file where this technique can be used is a file that contains menu options or screen layouts.

On the other hand, you should lock records only when you need to. While locking a record is generally fairly fast, the time to do so increases proportionally to the number of locks you are holding. Also, on networked file systems, locking records can be fairly slow because the networked machines must all be informed of the lock. Note that this is another reason to lock files when possible: when the file is locked, the runtime does not need to establish locks on individual records.

If you have enough memory, you can also increase your indexed file performance by increasing the V-BUFFERS configuration option described in Appendix H. This will increase the number of disk buffers used by the runtime system to hold key information.

Finally, you should read the information on the MASS-UPDATE option described in [section 6.1.6.2](#), as well as that on the Bulk Addition option described in [section 6.1.6.3](#). Using these options can significantly improve performance on some machines.

6.1.8 Limits on Open Files

Because Vision Versions 5 and 4 generate files in a dual file format, a COBOL application registers at least two open files for each Vision file. (Versions 2 and 3 use a single file format.) For large files with data

extensions, the number of open files per Vision file can be greater than two. Various techniques exist for controlling the number of files that can be open at one time:

1. The COBOL configuration variable **MAX_FILES** sets the maximum number of files that can be opened by the runtime system. The default value is "32". You can set this to any value up to "32767". Keeping this value small conserves memory.
2. Many operating systems limit the number of files that can be opened by a single process, so you may need to make some adjustments. On UNIX systems, file limits are controlled by kernel parameters. On VMS, file limits are controlled by process parameters.
3. If you are using an external file system or database instead of (or in conjunction with) Vision, limits may be imposed by the external file system or by the interface. See the ACUCOBOL-GT document for the specific interface that you are using.

6.2 Terminal I/O

ACUCOBOL-GT uses a sophisticated Terminal Manager that can provide many screen functions and improve terminal performance. To get the most out of the Terminal Manager, it helps to understand how it works. (For details on the Terminal Manager, see **Chapter 4, "Chapter 4: Terminal Manager."**)

6.2.1 Performance Considerations

ACUCOBOL-GT comes with a built-in window-oriented Terminal Manager. Efficient use of windows partially depends on the hardware characteristics of the host machine and terminal. Machines with memory-mapped screens (such as most personal computers) will run efficiently in any case. Machines with attached terminals, however, can significantly benefit from careful use of the windows.

As an example of an unwise choice of a window, consider trying to scroll a window that is almost the size of the screen. Without any hardware support, the Terminal Manager might have to move 1000 characters or so. On a memory-mapped machine, this will happen quickly, but on a machine with a terminal that is running at 9600 baud, this will take about one second. If several lines are being scrolled onto the screen, the entire operation will take several seconds to complete. This might be acceptable on occasion, but is very tedious if used regularly.

The Terminal Manager attempts to minimize screen I/O by keeping track of every character on the screen. Characters that do not need to be displayed (because they are already on the screen) are not sent to the terminal. Because much of a screen is often empty, substantial savings are achieved when you are clearing a window. Other general optimizations are done which result in generally faster screen I/O.

Special hardware characteristics of most terminals are used for time-consuming operations. The following guidelines will help you use windows efficiently on most machines by taking advantage of these characteristics.

- Large windows that will be cleared regularly should extend to the right edge of the screen. This allows the use of the terminal's "clear to end-of-line" function.
- Large windows that will be scrolled regularly should be the width of the screen and extend to the bottom edge. This case will use the "insert-line" and "delete-line" functions of most terminals.
- Reverse-video windows should be kept small. Creating a large reverse-video window is time-consuming because each location of the window must be initially written to. If you want a large window to stand out, consider making it boxed instead.

6.2.2 Terminal Manager Restrictions

Because of the windowing capabilities of the Terminal Manager, the programmer must abide by certain restrictions. This section describes these restrictions and how to work with them.

The primary restriction is that the Terminal Manager must retain control of the terminal. It needs to do this in order to implement the windowing functions. Also, by doing this, it can significantly improve screen performance over a “dumb” screen interface.

The main effect of this restriction is that you may not send arbitrary “escape” sequences to the terminal. Sending “escape” sequences that command the terminal in a hardware-specific fashion will confuse the Terminal Manager because it will not know what effect the “escape” sequence had on the screen. This will result in the Terminal Manager’s making incorrect assumptions about the screen and, eventually, will cause it to display incorrect data.

This restriction is not too serious, however, because the Terminal Manager can perform most functions that are normally accomplished through these “escape” sequences. Using the Terminal Manager to perform these functions has the added benefit of making them terminal-independent. This allows you to run the same application on any kind of terminal without having to keep a database of terminal capabilities maintained by the program.

The following points outline the Terminal Manager’s solution to various functions that are usually implemented via “escape” sequences.

Line Drawing

Many programs draw special-forms on the user’s screen using the terminal’s line drawing characters. This works fine if the line drawing characters are part of the terminal’s default character set (such as on a PC), but does not work if you need to send an “escape” sequence to switch character sets.

With ACUCOBOL-GT, you can use `DISPLAY BAR`, `DISPLAY LINE`, and `DISPLAY BOX` to perform line drawing. These verbs have the advantage that you can simply draw the form using screen coordinates without having to worry about different character sets on different terminals or using special characters to handle the intersections.

132-Column Handling	Some programs send “escape” sequences to switch between 80-column and 132-column mode on the terminal. This can be accomplished with the DISPLAY SCREEN verb in ACUCOBOL-GT.
Attached Devices	ACUCOBOL-GT can send data to a device attached to the terminal. For example, a printer or a cash register might be attached. The built-in library routine “ CSLOCALPRINT ” can send data to the attached device with a variety of line spacing options. This routine is described in Book 4, Appendix I.
Scrolling	ACUCOBOL-GT directly supports scrolling with the SCROLL option of the DISPLAY verb.
Function Key Labels	you can program function key labels by providing the appropriate command string in the “is” entry in “a_termcap”. For details, see section 4.6.9, “Initialization.” Currently, ACUCOBOL-GT does not provide a way to program function key labels at run time.

6.3 Memory Management

This section discusses how ACUCOBOL-GT manages runtime memory, the tools available to you to help manage program memory, and the facilities available to help find and correct memory management problems.

When **runcbl** initiates, it loads the main program into memory and begins execution. As other programs are called they are also loaded into memory. Once loaded, a program remains in memory until it is canceled (typically with the CANCEL statement). Canceling a program:

- places the program in its initial state
- closes its open files

- ensures that any VALUE clauses are in effect when the program is called again
- removes the program from memory, by default

This handling allows for a full ANSI implementation of subprograms (variables retain their previous values when a subprogram is re-entered, files can be left open in a subprogram, and so forth).

If subprograms are never canceled, a large system can eventually occupy very substantial amounts of memory. This is a common problem with large menu driven applications where the master menu calls programs for the various menu selections.

There are several mechanisms for managing runtime memory, including the CANCEL verb and the INITIAL and RESIDENT clauses of the PROGRAM-ID paragraph. To provide more flexibility, you can also enable a caching system in which canceled programs are not removed from memory until a specified memory limit is reached (see CANCEL below).

CANCEL

The CANCEL statement places the specified program in its initial state (see above). By default, the program is removed from memory. Subsequent calls to the program cause it to be reloaded from disk. This type of cancel, in which the program is removed from memory, is called a *physical cancel*. Alternatively, as an aid to tuning system performance, you can set up a system in which canceled programs are placed in their initial state but are left in memory until a specified memory limit is reached. Such cancels are called *logical cancels*. Subsequent calls to these programs start very quickly because the programs are already in memory. When the memory limit is reached, all logically canceled programs are removed from memory. Two runtime configuration variables are used to set up this system. See the entries for **DYNAMIC_MEMORY_LIMIT** and **LOGICAL_CANCELS** in Appendix H of Book 4. See also the **CANCEL Statement** in section 6.6 of Book 3.

When using logical cancels, replacing an object file on disk does not necessarily mean that subsequent calls to the program will use the new object. To force a subsequent call to load the new object file from disk, you must configure ACUCOBOL-GT to perform a physical cancel on that program.

Also, the use of logical cancels has some implications on debugging. The ACUCOBOL-GT debugger periodically reads source code from the object file on disk. When program code is cached by a logical cancel, the object file is closed and could be replaced or deleted. For the debugger to function correctly, it needs to keep the object file open and ensure that the object code in the disk file is identical to the code in memory. If the program is cached, the debugger accomplishes this by unloading the program from the cache, reopening the object file, and reloading the object code into memory. Because this effectively turns off logical cancels and the code caching feature when the debugger is active, the runtime configuration variable `DEBUG_NEWCOPY` can be used to control the runtime's behavior. See Book 4, Appendix H.

CANCEL ALL

Because it can be inconvenient to individually CANCEL every subprogram, ACUCOBOL-GT includes an extension to the CANCEL verb: CANCEL ALL. When executed, CANCEL ALL cancels every subprogram that is not active. (A program is active if it is either the main program or it has been called, but not yet exited.) Executing a CANCEL ALL after a menu selection completes is an easy way to free the memory used by that program.

INITIAL clause

The INITIAL clause in the PROGRAM-ID paragraph causes a subprogram to be automatically canceled whenever it exits. The program is removed from memory, regardless of whether the logical cancel feature is enabled. This can be used to help manage memory or to ensure that VALUE clauses are set whenever a subprogram is called. Note that the compiler will automatically apply this clause to a program compiled with the "-Zi" option. This can be particularly useful when you are converting programs from RM/COBOL version 2.

In RM/COBOL, the status of a program that exits is not precisely defined. Subprograms remain in memory until either their caller exits or until inadequate memory remains to load another subprogram. This occurs at fairly unpredictable times. Because their fate in memory is unclear, most RM/COBOL subprograms are treated as if they have been canceled when they exit. By applying the "-Zi" option at compile time, the ACUCOBOL-GT programmer can simulate this behavior. Note, however, that utility subprograms that are called repeatedly should not be compiled with "-Zi", because this introduces extra overhead each time they are called. Also, any subprogram that depends on retaining variables between calls should not be compiled with "-Zi".

Note: RM/COBOL treats spooled print files specially in that they are not closed when a subprogram exits (all other files are closed). ACUCOBOL-GT does not treat spooled files specially, so if this feature of RM/COBOL is used, the subprogram that does this should not be compiled with "-Zi".

RESIDENT clause

The RESIDENT clause in the PROGRAM-ID paragraph causes the program to remain resident in memory after its first execution. A program with the RESIDENT clause is not affected by a CANCEL or CANCEL ALL statement. Should the condition occur where there isn't enough system memory available to load a program, the CALL will fail with a "Inadequate memory available" error.

Other memory management tools

The **CHAIN Statement** (section 6.6 in Book 3) and the **CSCHAIN** library routine (Appendix I in Book 4) can be used to replace the running program with another program. This can conserve substantial amounts of memory. Segmentation (overlays) can be used to reduce the size of a single program. This is described in section 6.5 of Book 3. Finally, you can adjust the size of ACUCOBOL-GT's runtime buffers with the **V_BUFFERS** and **SORT_MEMORY** configuration options described in Appendix H of Book 4.

6.3.1 External Data Items

ACUCOBOL-GT manages external data items by allocating them in “pools”. The minimum size of each pool is set by the **EXTERNAL_SIZE** configuration option.

When a new external data item is needed, it is allocated from an existing pool. If it does not fit in any of the allocated pools, a new pool is allocated. The size of this pool is the same as the size of the data item, but never smaller than the value specified by the `EXTERNAL_SIZE` option. This reduces memory fragmentation. There are 32 pools available altogether.

Because external data items remain allocated after programs are canceled, it is best to allocate the external data items together so they do not break up the memory space. The default value for `EXTERNAL_SIZE` is “8192”.

6.4 Memory Testing and Error Handling

This section discusses techniques for recovering from common *runtime* errors. It also describes several mechanisms for tracking, testing, and debugging memory in the runtime, including linked C programs.

(*Compiler* error messages are listed alphabetically in Appendix D, **List of Errors**.)

6.4.1 Memory Access Violations

If your program tries to access a portion of memory that is protected by the operating system, ACUCOBOL-GT generates a *memory access violation* message and shuts down.

The most common cause of this violation is indexing beyond the legal range of a table. Often this sort of error is difficult to detect, because it can be data-dependent, as shown in this example:

```
PERFORM UNTIL IDX > NO-OF-EMPLOYEES
  DISPLAY "EMPLOYEE NAME:  ", EMP-TABLE(IDX)
  ADD 1 TO IDX
```

```
END PERFORM.
```

If NO-OF-EMPLOYEES ever exceeds the size of EMP-TABLE, or if IDX is ever set to zero, you could get a memory access violation when the program is run.

To uncover this type of memory access violation, use the “-Za” compile-time option while you are debugging your program. When you run a program that has been compiled with “-Za”, ACUCOBOL-GT prints a *subscript out of bounds* error message, detailing the legal range of indexes for any table-indexing error that it encounters. (Note that the *subscript out of bounds* error appears when you *run* the program, not when you compile it.) You can then correct the program as necessary.

A memory access violation can also occur if there is a bug in any C routine linked into the runtime. Because of the nature of C, the violation can occur long after the called C function executes. Techniques for debugging these problems are discussed in section 6.4.3 and in section 6.7 of *A Guide to Interoperating with ACUCOBOL-GT*.

6.4.2 Logging Errors to the Runtime's Error File

The code fragment below shows how to log permanent file errors into the runtime's error file. Normally, you would also use the “+e” runtime flag to name the error file and cause new messages to be appended to the end of the file.

Example:

```
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
SPECIAL-NAMES.  
    SYSERR IS ERROR-LOG.  
  
PROCEDURE DIVISION.  
DECLARATIVES.  
CUSTOMER-ERROR-PROCEDURE.  
    USE AFTER STANDARD ERROR PROCEDURE ON CUSTOMER-FILE.  
CUSTOMER-ERROR.  
    IF CUSTOMER-FILE-STATUS = "30" OR "98"  
        CALL "C$RERR" USING EXTENDED-STATUS
```

```
DISPLAY "CUSTOMER FILE ERROR", EXTENDED-STATUS  
UPON ERROR-LOG.
```

6.4.3 Runtime Memory Tracking and Testing

There are several mechanisms available for tracking, testing, and debugging memory in *extend* products and linked C programs. For information about using these facilities from a C program, see section 6.7 in *A Guide to Interoperating with ACUCOBOL-GT*.

All *extend* products have the ability to monitor, test, and debug memory allocations in three ways. They can:

1. Track memory boundaries, so that if the boundaries are corrupted a report is generated. This facility is called *memory bounds checking*.
2. Track how much memory is allocated in each of six subsystems. This facility is called *memory tracking*.
3. Output a description of each memory allocation, reallocation, and free by file and line, including a text message. This facility is called *memory handling descriptions*.

6.4.3.1 Memory handling descriptions

Memory handling descriptions report detailed information about memory allocation, reallocation, and frees. To turn on memory handling descriptions in the ACUCOBOL-GT runtime, simply specify the “-m *value file*” command-line option. When “-m *value file*” is specified, description and tracking information is written to the file named by the *file* argument. Exactly what information is included in the report is determined by the *value* argument as follows:

If *value* is odd, a final memory dump, showing all blocks still allocated, is performed.

If (*value* / 2) is odd, each allocation, reallocation, and free is written to the file.

If $(value / 4)$ is odd, a full memory dump is written for each allocation, reallocation, and free.

For example, if *value* equals 3, each allocation, reallocation, and free is reported and a final memory dump is performed when the runtime shuts down. This is because *value* is odd and $value / 2$ is also odd.

You can also enable memory handling descriptions for any *extend* product by setting two environment variables prior to product startup: A_MEM_DESC and A_MEM_DESC_FILE.

A_MEM_DESC must be set to *value*.

A_MEM_DESC_FILE specifies the name of the output file.

Both environment variables must be set to enable memory handling descriptions. Note that these are environment variables, *not* configuration variables. Also note that when memory descriptions are enabled with environment variables, the output does not (and for technical reasons cannot) include a final memory dump.

6.4.3.2 Memory tracking

The memory tracking feature is similar to the debugger “U” command, which displays the amount of dynamically allocated memory currently used by the runtime system. You can turn on memory tracking by setting the environment variables TRACE or A_DEBUG to positive numeric values. Apart from writing some C code (see 6.7.4 in *A Guide to Interoperating with ACUCOBOL-GT*), or running in the debugger (which turns this feature on automatically), there is no way to get the values stored for memory tracking.

6.4.3.3 Memory bounds checking

Memory bounds checking in COBOL programs is easily performed by compiling with the “-Za” option. See [section 6.4.1](#) and the entry for “-Za” in [section 2.2.16](#).

A second method can be helpful when the problem may be in a linked C routine. (See, also, section 6.7.5 in *A Guide to Interoperating with ACUCOBOL-GT*.)

There is a simple method of testing allocated memory bounds errors in any *extend* product. On Windows systems, the method is turned on and off with a registry entry. On other systems, the method is turned on and off with an environment variable. When an array bounds error is detected, a message is written to the file “dbgmalloc.log” in the current working directory.

To enable memory bounds checking on Windows, you must create a registry value “Test Allocated Memory Bounds” in the HKEY_CURRENT_USER\Software\Acucorp hive, with a type of DWORD and a value of non-zero. To turn memory bounds checking off, you must set the value to zero.

To enable memory bounds checking on UNIX, Linux, and other non-Windows platforms, you must set an environment variable named TEST_ALLOCATED_MEMORY_BOUNDS to a non-zero value. To turn memory bounds checking off, removed the environment variable or set its value to zero.

When memory bounds checking is enabled, messages are printed whenever the memory checking routines detect that allocated memory bounds have been overwritten.

Note: Application performance can be adversely impacted by memory bounds checking.

6.5 Screen Section

The ACUCOBOL-GT Screen Section uses data structures that are similar to the record descriptions in Working-Storage. However, the Screen Section offers an alternate method for displaying information to the user’s screen and accepting data from the keyboard. It lets you display and update data items in groups, rather than as individual fields.

The Screen Section enables a *non-intelligent* terminal to emulate an *intelligent* one by acting as if it were displaying and accepting an entire screen of data at one time.

Intelligent terminals can readily accept and transmit an entire terminal screen. The user moves around the screen and fills in a *form* consisting of several fields. The intelligent terminal stores all of the entries for the form in a buffer, and then transmits an entire screen full of information at one time.

Non-intelligent terminals are not designed to process data in this way. Instead, data is displayed and entered on a field-by-field basis. However, the ACUCOBOL-GT Screen Section lets non-intelligent terminals emulate intelligent terminals by acting as if they are displaying and accepting an entire screen of data at one time.

Although the data is actually handled behind the scenes on a field-by-field basis, the COBOL programmer can display or accept either individual fields or entire screens with one COBOL statement. Although it's common for a screen entry in the Screen Section to describe an entire screen, it need not do so. The language is so flexible that the programmer can decide whether to include one field, a few fields, or a screen full of fields in a Screen Section entry. Information about the Screen Section can be found in section 5.8, "Screen Section" in Book 3, *Reference Manual*.

6.5.1 Advantages

The Screen Section lets you display or accept a screen full of items with only one COBOL statement. In addition, a significant amount of cursor handling is built in, so that end users move easily among the fields on the screen. Entry of tables of data becomes very simple, as shown in the table example later in this section. End users gain the flexibility of entering and validating a screen full of related data at one time.

6.5.2 Structure

A *screen description entry* is similar to a data description entry. Just as a record's data description typically consists of

- a level number
- a data-name
- a Picture Clause

so a screen description entry typically consists of

- a level number
- a screen-name
- a Picture Clause

Because the programmer decides how many fields to group together, the *screen-name* may not actually refer to a full screen of data items. It might refer to a single field, a few fields, or an entire screen. In any case, each group or subgroup must be given a screen-name.

To place a screen or *form* on the terminal and receive data into that form, you need two different COBOL statements. The *literals* in a screen description entry are displayed by a **DISPLAY Statement**.

These are the prompting words that guide the end user. The *information* entered into the data items is accepted from the keyboard by an **ACCEPT Statement**. So, first you DISPLAY the screen, and then you ACCEPT the end user's entries for that screen.

6.5.3 Syntax

Each *screen description entry* must start with a level-number from level 01 through level 49. At the top level (01), the *screen* is given a name. For example:

```
01  employee-info-screen.
```

The other levels of a screen description entry (level 03 and so forth) can either name subscreens within the top-level screen, or:

- provide literals that prompt the end user for entries
- position the cursor
- identify the data items that will store the information the end user types into each field

A complete description of Screen Section *screen description entry* formats, their syntax and general rules can be found in **section 5.9, “Screen Description Entry,”** in Book 3, *Reference Manual*.

When your program DISPLAYs a screen or ACCEPTs entries for a screen, the screen-name you reference in the DISPLAY or ACCEPT statement determines which fields are included. Everything subordinate to the screen-name is affected. For example, if you display a screen-name that's defined as a level 03 with two subscreens defined as level 05s below it, then all of the fields subordinate to the level 03 are displayed, and both of the subordinate level 05 screens and their attributes are also displayed.

Here are some examples.

This is an entry that names the screen and provides one literal:

```
01 employee-info-screen.  
   03 emp-number-prompt value is "EMPLOYEE NUMBER: ".
```

This entry positions the cursor:

```
   03 column plus 1  
      line plus 2.
```

This entry identifies a data item to receive a typed entry:

```
   03 PIC 9(9) to emp-number.
```

This defines a subscreen:

```
   03 pay-period-dates.  
      05 PIC 9(6) USING period-start.  
      05 PIC 9(6) USING period-end.
```

Here's a complete program example for a table. It accepts a simple table on one line:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. TABLE-SAMPLE-1.  
  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
  
01 TABLE-1.  
   03 TABLE-ITEM OCCURS 5 TIMES          PIC X(5).
```



```
SCREEN SECTION.
```

```
01 SCREEN-1.  
   03 "TABLE ITEMS:".  
   03 OCCURS 5 TIMES USING TABLE-ITEM, COLUMN + 2.
```

```
PROCEDURE DIVISION.
```

```
MAIN-LOGIC.
```

```
    DISPLAY WINDOW ERASE.  
    DISPLAY SCREEN-1.  
    ACCEPT SCREEN-1.  
    STOP RUN.
```

For additional examples that use tables, see Book 3, *Reference Manual*, [section 5.9.3, “OCCURS Clause.”](#)

The name that you give to a screen or subscreen in your Data Division must be referenced by the DISPLAY and ACCEPT statements that use the screen. Here’s how that works.

First, you display the screen with a Format 2 DISPLAY statement. When this type of DISPLAY is executed, the screen-name in the DISPLAY statement tells *which group of literals and data items* should be displayed at the terminal. All of the literals in and subordinate to that screen-name definition are displayed in their appropriate positions. These typically serve as the prompts on the screen, guiding the end user. So the end user sees a *form* on the terminal, with multiple locations to enter data. That’s why a Format 2 DISPLAY is said to be a *form-level* DISPLAY. Typical literals in a payroll program might be “Employee Name: “, “Employee Number: “, and “Pay Period Ending Date: “. The DISPLAY statement *ignores* the data items in the screen definition that are entry-only (TO) fields.

Second, you accept data into the screen with a Format 2 ACCEPT statement. When an ACCEPT statement of this type is executed, the screen-name that it references tells which *screen* of data items will be accepted. The ACCEPT statement *ignores* the literals in the screen definition as well as display-only (FROM) fields. The cursor moves to the position you’ve assigned to the first (non-literal) data item in the group, and waits for an entry. The end user can move around on the screen, from field to field and back again, with the tab

and arrow keys (or other keys if you have customized the keyboard). When all items have been entered, the user presses a termination key to indicate that the screen is complete.

There are three types of screen description entries:

- If a VALUE clause is specified, then the entry is a display literal. The word VALUE may or may not appear in a Screen Section VALUE clause.
- If a PICTURE clause is specified, then the entry is a data field. The word PICTURE may or may not appear in a Screen Section PICTURE clause.
- If neither a VALUE nor a PICTURE clause is specified, the entry is either a screen-name or an attribute.

6.5.4 Comparison to Field-level

The following paragraphs compare a field-level DISPLAY and ACCEPT to a form-level DISPLAY and ACCEPT for the same data. Both examples display a simple name and address form, and prompt the user to enter data.

Field-level ACCEPT & DISPLAY

```
DISPLAY-AND-ACCEPT-ADDRESS.  
  DISPLAY "Name:", ERASE SCREEN, LINE 1.  
  DISPLAY "Address 1:", LINE 2.  
  DISPLAY "Address 2:", LINE 3.  
  DISPLAY "City:", LINE 4, "State:", COLUMN 25.  
  ACCEPT NAME, LINE 1, COLUMN 12.  
  ACCEPT ADDRESS-1, LINE 2, COLUMN 12.  
  ACCEPT ADDRESS-2, LINE 3, COLUMN 12.  
  ACCEPT CITY, LINE 4, COLUMN 12.  
  ACCEPT STATE, LINE 4, COLUMN 32.
```

Form-level DISPLAY and ACCEPT (Screen Section)

```
SCREEN SECTION.  
01 ADDRESS-SCREEN.  
  03 ERASE SCREEN.  
  03 "Name:".  
  03 TO NAME, COLUMN 12.
```

```

03 "Address 1:", LINE + 1.
03 TO ADDRESS-1, COLUMN 12.
03 "Address 2:", LINE + 1.
03 TO ADDRESS-2, COLUMN 12.
03 "City:", LINE + 1.
03 TO CITY, COLUMN 12.
03 "State:", COLUMN 25.
03 TO STATE, COLUMN 32.
DISPLAY-AND-ACCEPT-ADDRESS.
  DISPLAY ADDRESS-SCREEN.
  ACCEPT ADDRESS-SCREEN.

```

There are three primary advantages of the Screen Section:

- The centralized screen definition is easier to maintain, especially when screen processing occurs in more than one place in the program.
- Automatic handling of arrow keys allows the user to move between fields without specialized programming.
- Users gain improved automatic mouse handling on systems where the mouse is supported.

6.5.5 Using Screen Section Embedded Procedures

The following example shows how to use embedded procedures to provide an automatic look-up function plus field validation on a key field. In this example, an ellipsis in braces indicates omitted code.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SCREEN-EXAMPLE.
REMARKS.

```

This program shows how to use embedded procedures in the Screen Section to:

- show a field-specific legend when the user arrives at that field,
- perform validation of a key field and,
- perform a look-up procedure when a special function key is pressed.

In this example, a customer-number field is included in an order-entry screen. When the user enters a customer number, the program validates that it's an existing customer and, if so, displays the customer's

name. If it's not valid, the user must re-enter the field. If the user presses the F1 key, a look-up procedure locates the desired customer.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SPECIAL-NAMES.

CRT STATUS IS CRT-STATUS

SCREEN CONTROL IS SCREEN-CONTROL.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

{ . . . }

DATA DIVISION.

FILE SECTION.

{ . . . }

WORKING-STORAGE SECTION.

01 CRT-STATUS PIC 9(3).
88 F1-KEY VALUE 1.

01 SCREEN-CONTROL.

03 ACCEPT-CONTROL PIC 9.
88 GOTO-FIELD VALUE 1.
03 CONTROL-VALUE PIC 999.
03 CONTROL-HANDLE HANDLE.
03 CONTROL-ID PIC XX COMP-X.

{ . . . }

SCREEN SECTION.

01 ORDER-SCREEN.

{ . . . }

03 "Cust #: ".

03 USING CUSTOMER-NO

BEFORE PROCEDURE IS SHOW-CUST-LEGEND

AFTER PROCEDURE IS TEST-CUSTOMER

EXCEPTION PROCEDURE IS CHECK-FOR-LOOKUP.

03 SHOW-CUSTOMER-NAME, PIC X(30) FROM
CUSTOMER-NAME, COLUMN + 3.

{ . . . }

PROCEDURE DIVISION.

MAIN-LOGIC.

```
{ . . . }  
  DISPLAY ORDER-SCREEN.  
  ACCEPT ORDER-SCREEN  
    ON EXCEPTION CONTINUE  
    NOT ON EXCEPTION WRITE ORDER-RECORD  
  END-ACCEPT.  
{ . . . }  
STOP RUN.
```

- * SHOW-CUST-LEGEND executes whenever the user
- * arrives at the customer number field. It
- * displays a legend. This legend is removed by
- * both the AFTER and EXCEPTION procedures
- * associated with the customer-number field.

SHOW-CUST-LEGEND.

```
  DISPLAY "F1 = Customer Lookup", LINE 24,  
  ERASE TO END OF LINE.
```

- * TEST-CUSTOMER checks for a valid customer number
- * entry by reading the customer file. If it finds a
- * customer record, it displays the customer's name.
- * If it does not find a record, it forces the user
- * to re-enter the field by setting the SCREEN-
- * CONTROL condition, GOTO-FIELD, to TRUE. Since
- * the ACCEPT statement initializes CONTROL-VALUE to
- * the field number of the customer number field,
- * setting GOTO-FIELD to TRUE will cause the ACCEPT
- * statement to return to the customer-number field.

TEST-CUSTOMER.

```
  DISPLAY SPACES, LINE 24, ERASE TO END OF LINE.  
  READ CUSTOMER-FILE RECORD  
  INVALID KEY  
    DISPLAY "CUSTOMER NOT ON FILE - PRESS RETURN",  
    LINE 24, BOLD  
  ACCEPT OMITTED  
  SET GOTO-FIELD TO TRUE
```

```
  NOT INVALID KEY  
  DISPLAY SHOW-CUSTOMER-NAME.
```

* CHECK-FOR-LOOKUP executes when the user types a
* function key when in the customer-number field.
* It erases the legend and then checks to see if
* Function Key 1 was pressed. If it was, it
* executes a look-up procedure. If the procedure
* returns with a valid customer selected, it
* displays the customer's name and causes control
* to pass to the next field. Otherwise, it forces
* the user to re-enter the customer-number field.
* It does this by setting GOTO-FIELD to TRUE while
* leaving CONTROL-VALUE unchanged.

```
CHECK-FOR-LOOKUP.  
  DISPLAY SPACES, LINE 24, ERASE TO END OF LINE.  
  IF F1-KEY  
    PERFORM CUSTOMER-LOOKUP-PROCEDURE  
    IF HAVE-CUSTOMER-NUMBER  
      DISPLAY SHOW-CUSTOMER-NAME  
      ADD 1 TO CONTROL-VALUE  
    END-IF  
  SET GOTO-FIELD TO TRUE.
```

A complete description of the rules that govern the execution of embedded procedures can be found in section 5.9.6 of Book 3, *Reference Manual*. Rules covering the use of event procedures, a variant of embedded procedures, are also found in this discussion of the PROCEDURE clause.

6.6 Data Validation

ACUCOBOL-GT includes several mechanisms that instruct the runtime to perform specific types of *validation* on user supplied data. In instances where an input field is marked for validation, the runtime typically performs data validation when the user attempts to leave the data entry field. Data validation mechanisms include:

- the REQUIRED (EMPTY-CHECK) and FULL (LENGTH-CHECK) phrases of the ACCEPT and Screen Section *Screen Description Entry* statements (see their respective entries in Book 3, *Reference Manual*, **section 6.4.9, “Common Screen Options.”**)

- the SCREEN CHECK-NUMBERS=Validate option of the KEYBOARD variable (see **section 4.3.2.1, “The KEYBOARD variable,”** in this book).

Although validation is usually performed when the user attempts to leave the field, this is not always the case. Validation is not performed when the ACCEPT terminates as a result of the following:

- an event, except CMD-GOTO and CMD-TABCHANGED
- an exception key, except for keys that move the cursor from field to field in a Screen Section, such as the arrow keys
- a message

The special exceptions noted above (CMD-GOTO, CMD-TABCHANGED, and the arrow keys) are treated as normal termination with regards to validation (they trigger validation), but they also generate their usual exception conditions.

6.7 Exiting From ACUCOBOL-GT Programs

If an application written in ACUCOBOL-GT is aborted, the following exit techniques produce different results:

- **Catastrophic exit:** A power failure, turning off the computer, or issuing a “kill -9” from the console are all examples of catastrophic exits. The runtime cannot trap exits of this kind. Any files open at the time of a catastrophic exit may be corrupted.
- **Graceful abort:** A “kill” (not a “kill -9”) from the console and a Control-C from the keyboard are forms of program abort that ACUCOBOL-GT tries to detect. If the abort signal is intercepted by the runtime system, the runtime will close any open files and set the user count back.
- **Safe mode exit:** The runtime option “-s” instructs the runtime to trap graceful abort signals such as Control-C and *prevent the abort from occurring*. Only normally coded exit paths are allowed in safe mode.

Note: Even in safe mode, the runtime cannot trap a catastrophic exit. So even if you run with “-s”, turning off the computer or issuing a “kill -9” will risk corrupting the file.

6.8 Multiple Execution Threads

ACUCOBOL-GT includes support for the specification and control of multiple execution paths in the program (multithreading). This means that you can define separately executing program *threads* and control their execution.

A *thread* is an execution path through a program. All programs have at least one thread, the thread (execution path) that starts when the program starts and ends when the program ends. In a program that has multiple threads, more than one thread may be *active* at a time.

When more than one thread is active, the runtime monitors and switches among threads, dividing processing time in accordance to program activity and thread priority (thread priorities are described below). In this way, threads can be thought of as running in parallel. Although conceptually they run in parallel, keep in mind that your program is still executing one statement at a time. The runtime controls the movement among active threads in response to a variety of conditions, such as computational activity, file I/O, user input, and programmatic controls, such as *messages* and *locks* (described below).

While most programs do not need multiple execution paths, there are some cases where threads can be helpful.

1. Threads can be used to simplify modeless window handling. By tying threads to modeless windows you simplify the task of managing multiple windows. If each modeless window has its own thread, the runtime can automatically activate or suspend each window (via its associated thread). The code within each thread can then focus solely on managing its own window. This approach is much easier than the alternative, in which the program must be made to handle requests from any window at any time.

2. Threads can improve system throughput. There are times when you can improve system throughput by performing tasks in one thread while waiting for user input in another thread. For example, you could do your program initialization while the user enters his or her password. Or, you could allow the user to edit the first item found by a complex search while your program continues searching for additional items.
3. Threads can be used to control program actions. For example a program can present and accept a “cancel” dialog box in one thread while running a report in another. If the user clicks on the cancel button, that thread stops the report by terminating the report’s thread.
4. Threads can be used to perform periodic updates. Any periodic effect, such as updating an on-screen clock or performing animation, can easily be accomplished in a thread by placing that thread in a loop that sleeps for some period of time, wakes and performs the update, and then loops back to the sleep cycle. This allows the rest of your program to be written without concern about performing the periodic update.

6.8.1 Thread Fundamentals

Threads are created with either the `PERFORM` statement or the `CALL` statement. You simply add the word “`THREAD`” after the verb. The thread runs the target of the `PERFORM` or `CALL` statement. When the target completes, the thread is destroyed.

For example, to run the paragraph “`OPEN-FILES`” in a thread, you would code:

```
PERFORM THREAD OPEN-FILES
```

The main thread continues to execute at the statement following the `PERFORM`, while the `OPEN-FILES` code begins execution in a separate thread. When `OPEN-FILES` finishes, the thread is destroyed.

Threads are identified by *thread handles*. When you create a thread, you can optionally store its handle. Thread handles are used to communicate among threads. The next example demonstrates how this works.

There are times when you will want to ensure that a thread is complete before continuing in the program. For example, if one thread is used to open an application's files while another thread gets the user's password, you will not want to continue in the program until both tasks are complete, otherwise you might try to look up the password in a file that is closed. You can use the WAIT verb to make sure that both threads are complete before continuing. The WAIT verb causes the thread to wait for the specified thread to finish or send a message. The following example illustrates this process:

```
77  THREAD-1  USAGE HANDLE OF THREAD.  
  
PERFORM THREAD OPEN-FILES, HANDLE IN THREAD-1  
PERFORM GET-PASSWORD  
WAIT FOR THREAD THREAD-1  
PERFORM VALIDATE-PASSWORD
```

In this example, the paragraphs OPEN-FILES and GET-PASSWORD run in parallel. The WAIT statement ensures that OPEN-FILES is finished before the main thread goes on to validate the password.

6.8.1.1 LAST THREAD

The need to synchronize threads is so common that there is an optional phrase, LAST THREAD, that can be used to simplify coding. Note that the thread that LAST THREAD refers to is dynamic, so some care must be taken in its use. LAST THREAD refers to:

1. the last thread created by the current thread, or the last thread that the current thread communicated with, whichever was the last action
2. or, if neither of the actions described in (1) has occurred, the parent thread (the thread that created the current thread)

You can use the LAST THREAD phrase to eliminate the need to store a thread handle. For example, the preceding example could also be written as:

```
PERFORM THREAD OPEN-FILES  
PERFORM GET-PASSWORD  
WAIT FOR LAST THREAD  
PERFORM VALIDATE-PASSWORD
```

A thread normally ends when the PERFORM or CALL statement that created it completes. You can end a thread earlier in the execution path with the STOP THREAD statement. Used by itself, STOP THREAD terminates the current thread. You can also specify a thread handle to stop another thread. If there is only one thread running, stopping that thread is equivalent to doing a STOP RUN.

6.8.2 Data Sharing Among Threads

In the program that created it, a thread and its parent thread share the same data. This includes both working-storage and files (including the file's open state and record position). Thus any changes made in one thread will be seen by the other thread. The only data that are not shared are those items internally generated by the compiler that are related to the flow of control within a thread, such as a paragraph's return address or an internal counter associated with PERFORM "N" TIMES.

When a thread CALLs a program, any data local to that program are private to that thread (and its child threads). As a result, you can write utility routines that many threads can call, without having to worry about data sharing issues in the utility. Note that any data passed (BY REFERENCE) to the called program are shared if they are shared in the program passing the data.

External data items and external files are always shared by all threads.

ACUCOBOL-GT ensures that any one logical operation on a data item is fully completed before control switches between threads. Consider the following example:

```
77 DATA-1      PIC X(3) .

PERFORM THREAD
    MOVE "ABC" TO DATA-1
END-PERFORM
MOVE "DEF" TO DATA-1
WAIT FOR LAST THREAD
DISPLAY DATA-1
```

The last DISPLAY will either print “ABC” or “DEF” depending on the order in which the threads execute. It will not, however, print something like “AEF” or “DEC” because each MOVE that affects DATA-1 is performed in its entirety before control switches between threads.

6.8.2.1 LOCK THREAD and UNLOCK THREAD

There may be times when you want to be sure that a group of operations is performed without switching threads. For example, if you have a utility paragraph that does a series of math operations on a data item, you will not want to begin that paragraph and then switch to another thread if another thread uses that data item. In this case, you can use the LOCK THREAD statement to ensure that a series of operations are all accomplished together. For example:

```
PARA-1.  
    LOCK THREAD  
    ADD DATA-1 TO DATA-2  
    IF DATA-2 > DATA-2-DIV-LIMIT  
        MOVE DATA-2-DIV-LIMIT TO DATA-2  
    COMPUTE DATA-2 = DATA-2 / DATA-3  
    UNLOCK THREAD.
```

The LOCK THREAD statement ensures that no other thread executes. This condition remains in effect until the thread is unlocked (with UNLOCK THREAD) or the thread terminates. In the preceding example, if the LOCK THREAD was omitted, another thread referencing DATA-2 could see its value after the ADD but before the COMPUTE. This could be a value that is not meaningful.

A thread can have more than one lock. Each time a LOCK THREAD statement executes in the thread, the number of locks held by that thread increases by one. To allow other threads to run again, an equal number of UNLOCK THREAD statements must also execute. Each UNLOCK THREAD statement removes the last lock applied to the thread. This capability allows a thread to lock itself, call a subroutine that also locks itself, and remain locked after that subroutines unlocks itself.

The data sharing aspect of threads is very powerful. However, ensuring that the data are always consistent can be a difficult programming problem. When writing multithreaded programs, you should strive to share data in a

well-defined manner to minimize this problem. The best way to do this is to share as little data as possible and to be clear when each thread is allowed to use that data.

6.8.3 Thread Communication

As noted in the previous section, you can share data items among threads. This is the easiest form of communication among threads. The drawback to this form of data sharing is that you must carefully code the program so as to ensure that data is accessed and updated in a consistent manner. A second drawback is that there is no efficient way for one thread to wait for another to update a particular data item. You could arrange a set of data items to act as semaphores, but it is inefficient for a thread to loop until a data item reaches a particular value. It would be much better for the thread to sleep until the data item it needs is available.

To solve this problem, ACUCOBOL-GT provides a way for threads to send *messages* to each other. A message can be any data item. You decide in your program how you want to format messages.

6.8.3.1 SEND and RECEIVE

A thread sends a message to another thread with the SEND statement. A thread receives a message from another thread with the RECEIVE statement. Messages can either be *broadcast*, in which case they are sent to all threads, or *directed*, in which case the message is sent to a particular thread or set of threads.

For example, suppose you have a thread that updates a complicated display in response to a message. Another thread receives user input and sends messages to the update thread. The code might look like this:

```
77 H-DISPLAY-THREAD    HANDLE OF THREAD.
77 RECORD-NUMBER      PIC 9(5) .
77 UPDATE-RECORD-NO   PIC 9(5) .

MAIN-LOGIC.
    PERFORM INITIALIZE
    PERFORM CREATE-MAIN-SCREEN
    PERFORM THREAD DISPLAY-THREAD,
        HANDLE IN H-DISPLAY-THREAD
```

```
PERFORM UNTIL DONE
  PERFORM ENTER-MAIN-SCREEN
  IF NOT DONE
    SEND RECORD-NUMBER TO
      THREAD H-DISPLAY-THREAD
  END-IF
END-PERFORM
PERFORM SHUT-DOWN
STOP RUN.

DISPLAY-THREAD.
PERFORM CREATE-STATUS-SCREEN
PERFORM UNTIL 1 = 0
  RECEIVE UPDATE-RECORD-NO FROM ANY THREAD
  PERFORM UPDATE-STATUS-SCREEN
END-PERFORM.
```

The thread that updates the screen sits in an infinite loop waiting for messages. It will terminate when the runtime shuts down. Because it uses the `RECEIVE` statement, this thread is very efficient even though it contains an infinite loop. Note that the program includes two copies of the “record number”: one sent by the main thread, and one to hold the value received by the update thread. This isolates the data sharing to the `SEND` and `RECEIVE` statements. If the main thread goes on to change `RECORD-NUMBER` while the update thread is performing its screen updates, the action will not affect the update thread because it has its own copy in `STATUS-RECORD-NO`.

The following example expands on the previous one to make the code more robust. In particular, the new code handles the case where other threads may be sending different types of messages, as well the case where all the sending threads die for some reason. Note that some of the code from the first example has been omitted for brevity:

```
77 H-DISPLAY-THREAD      HANDLE OF THREAD.
77 RECORD-NUMBER        PIC 9(5).

78 UPDATE-MSG-TYPE      VALUE 1.

01 SENDING-RECORD.
   03 SENDING-MSG-TYPE   PIC 99.
   03 SENDING-REC-NO     PIC 9(5).

01 UPDATE-RECORD.
   03 FILLER             PIC 99.
```

```

      88 IS-UPDATE-MSG      VALUE UPDATE-MSG-TYPE.
03  UPDATE-RECORD-NO      PIC 9(5) .

MAIN-LOGIC.
  PERFORM THREAD DISPLAY-THREAD,
    HANDLE IN H-DISPLAY-THREAD
  PERFORM UNTIL DONE
    PERFORM ENTER-MAIN-SCREEN
  IF NOT DONE
    MOVE UPDATE-MSG-TYPE TO SENDING-MSG-TYPE
    MOVE RECORD-NUMBER TO SENDING-REC-NO
    SEND SENDING-RECORD TO
      THREAD H-DISPLAY-THREAD
    END-SEND
  END-IF
END-PERFORM.

DISPLAY-THREAD.
  PERFORM CREATE-STATUS-SCREEN
  PERFORM UNTIL 1 = 0
    RECEIVE UPDATE-RECORD FROM ANY THREAD
    ON EXCEPTION
      PERFORM SENDING-THREADS-DIED-ERROR
    END-RECEIVE
  IF IS-UPDATE-MSG
    PERFORM UPDATE-STATUS-SCREEN
  END-IF
END-PERFORM.

```

The preceding example assumes that all messages in the program will be formatted with a two-digit type code as the first element. The *update* thread checks the message received to see if it contains a type that it knows how to respond to. If it does not, it simply ignores the message. This check is a good idea because the update thread uses the ANY THREAD option in its RECEIVE statement. If some other thread broadcasts a message (SEND TO ALL THREADS), the update thread would receive the message even though it might not be an “update” message. Adding the message-type code resolves this issue. It also makes debugging easier if you have more than one message type in your program.

Messages can also interrupt a thread that is in an ACCEPT statement. However, to allow that you must declare that the ACCEPT statement may be interrupted. This is specified with the ALLOWING MESSAGES phrase in the ACCEPT statement. The next section includes an example that illustrates when this capability may be useful.

6.8.4 Thread Priorities

In a multithreaded program, it is likely that many threads will be active at the same time. To ensure that each active thread gets its proper share of CPU time, the runtime tracks certain types of operations, such as DISPLAY, file IO and basic computation operations, and periodically switches the executing thread. Each of these opportunities to change threads is called a switch point. The execution priority of each thread helps to determine which thread gets control at each switch point.

A thread's execution priority is an integer value. By default, all threads start with a priority value of 100. You use a Format 12 SET statement to change a thread's priority value.

Threads receive control in proportion to their priority. The higher the priority, the more often that thread gets control at a switch point. Thus, a thread with a priority of 50 gains control half as often as a thread with a priority of 100. Of course, if a thread is paused for any reason, such as waiting for input, the thread does not gain control. The minimum priority for a thread is one (1), the maximum is 32767.

Three runtime configuration variables, **SWITCH_PERIOD**, **DISPLAY_SWITCH_PERIOD**, and **IO_SWITCH_PERIOD**, can be used to affect how the runtime manages thread switching.

6.8.5 Threading Rules That Affect Windows and ACCEPT Statements

There are a number of special rules that govern how and when threads can receive user input. In the following discussion, the phrase "ACCEPT statement" refers specifically to formats 1, 2, and 7 of the ACCEPT statement. These are the forms that retrieve the user's input.

When a thread attempts to execute an ACCEPT statement, two rules determine if that thread is allowed to proceed:

1. The thread may proceed if no other thread is in an ACCEPT statement. This rule ensures that threads do not compete for a single piece of input. It prevents the possibility, for example, of alternate keystrokes going to different threads.
2. The thread may proceed if the window in which the ACCEPT will occur is active, or if the active window is controlled by the current thread. A window is controlled by the thread that created it or the last thread that did an ACCEPT statement in it. A window can be controlled by only one thread at a time. This rule ensures that windows do not activate arbitrarily.

If an ACCEPT does not meet both of these conditions at the time that it would normally execute, its thread is suspended. The thread will remain suspended until both conditions are met (at which time the thread resumes and the ACCEPT executes).

Note: Even if the thread is suspended and the ACCEPT is not started, the ACCEPT may still be timed out with ACCEPT BEFORE TIME. It can also be terminated with a message, if ACCEPT ALLOWING MESSAGES is specified.

These rules apply only if there is more than one thread. If a suspended thread becomes the only remaining thread, then it automatically resumes execution.

If a thread suspends due to rule (2), it will suspend forever unless its window becomes active. This can happen in one of two ways: (a) another thread activates the window, or (b) the user activates the window. Arranging for the latter is much more common. It is the case where you want to give the user the ability to work in multiple windows at once, freely switching between them.

To create windows that the user can activate at will, you describe the windows as MODELESS when you create them. A modeless window allows the user to activate another window (the other type of window, called modal, does not allow the user to activate another window). When the user activates a modeless window, by clicking on it or using the host's method, the runtime

generates a CMD-ACTIVATE event. In a single-threaded program, the program must detect this event and ACCEPT something in the appropriate window in order to activate that window. In a multithreaded program, there is a second option. To use this option you code your program in such a way that each modeless window runs in a separate thread, and you use the phrase LINK TO THREAD or BIND TO THREAD when you create each window. This phrase instructs the runtime to handle the CMD-ACTIVATE events on its own. In this arrangement, when a CMD-ACTIVATE event occurs, the runtime suspends the current thread and activates the new window. The thread controlling the newly active window then resumes execution. Technically speaking, the thread running the ACCEPT in the newly active window can leave the suspended state because it meets rules (1) and (2) described above.

This automatic runtime handling allows you to easily manage multiple windows. Each window is contained in a thread that need only manage that one window. The threads do not need to be aware of each other or know which window is active. If the threads have some need to communicate among themselves, they can do so with *messages* (see [Section 6.8.3, “Thread Communication”](#)).

Here is an example. The following program presents a maintenance screen for a data file. It also pops-up a window that has a list of all the records in the file. The user can update a record on the maintenance screen, and can select records from the list. When the user selects a record, the maintenance screen is updated with the selected record.

```
MAIN-LOGIC.  
  PERFORM INITIALIZE  
  DISPLAY STANDARD GRAPHICAL WINDOW,  
    LINK TO THREAD  
  PERFORM THREAD SEARCH-THREAD,  
  PERFORM UNTIL DONE  
    PERFORM DISPLAY-RECORD  
  ACCEPT MAINTENANCE-SCREEN  
    ALLOWING MESSAGES FROM LAST THREAD  
  ON EXCEPTION  
    IF KEY-STATUS = W-MESSAGE  
      PERFORM SAVE-CURRENT-RECORD  
      RECEIVE RECORD-NUMBER  
        FROM LAST THREAD  
    END-IF  
  END-ACCEPT
```

```
END-PERFORM
PERFORM SHUT-DOWN
STOP RUN.

SEARCH-THREAD.
  DISPLAY FLOATING GRAPHICAL WINDOW,
  MODELESS, LINK TO THREAD
  HANDLE IN H-SEARCH-WINDOW
PERFORM DISPLAY-SEARCH-LIST
PERFORM UNTIL 1 = 0
  PERFORM ACCEPT-SEARCH-LIST
  SEND RECORD-NUMBER-SELECTED TO LAST THREAD
END-PERFORM.
```

This example shows how using threads can simplify managing multiple windows. In it, one thread manages the main window, and one manages the search window. The search window is very simple, its thread sits in an infinite loop fetching a list item from the user and sending it to the main thread. The main thread sits in a loop accepting the main screen. If a message arrives from the search thread, it saves the current record and retrieves the record sent from the search window. By using multiple threads, each piece of your window-handling code stands alone. If it were written as a single-threaded program, the two loops accepting the windows would have to be combined into a single loop, with a state variable tracking which window is currently active.

Tip: The use of modeless windows and threads to provide the ability to switch between screens by pressing Taskbar buttons is demonstrated in an AcuBench sample project located in the Support area of the Micro Focus Web site. To download the project, go to: <http://supportline.microfocus.com/examplesandutilities/index.asp>. Select **Acu samples > Graphical User Interface Sample Programs > Threadds.zip**.

6.8.6 Thread Pausing

There are some conditions in which all of the threads in your program will pause. One reason this occurs is when some event in the system causes the entire runtime to suspend. The pause ends when the runtime regains control. The main cases where this occurs are:

1. when moving any of the application's windows with the mouse. The host system manages this task.
2. when a message box is displayed (by calling the "MESSAGE" utility program or DISPLAY MESSAGE BOX). The message box is managed by the host system. You can use this to your advantage by using message boxes to report critical errors. The message box requires that the user respond to the box before any part of the application will resume.

Although the host system does not handle these tasks in character-based versions of the runtime, architectural similarities between the character and graphical versions cause these tasks to produce the same effect on character systems.

Another reason for all threads in your multithread program to pause is waiting for locked records. You can configure the runtime to automatically wait for locked records by either setting the WAIT-FOR-LOCKS configuration option, or by compiling with the RM/COBOL compatibility ("-cr" option) and omitting a declarative for this particular file. When you do this in a multi-threaded program, all threads are frozen while waiting for a locked record to unlock. This occurs because the runtime employs a very tight wait loop that does not allow other threads to continue running.

Note: You can avoid this by writing your own wait loop. Simply turn off the runtime option and perform the affected file operations in a loop until the file status is not "99". For efficiency, we recommend calling "C\$SLEEP" for at least one second between file operations when waiting for a lock.

6.8.7 Multithreading and Multiprocessor Systems

ACUCOBOL-GT implements multithreading in a machine-independent fashion. It neither needs nor uses any multithreading capabilities of the host system. This has several advantages:

1. You can run multithreaded programs in environments that do not otherwise support multithreading.

2. Multithreaded programs run the same way under different environments.
3. ACUCOBOL-GT can provide features not universally available in multithreaded systems, such as the link between windows and threads.

There are, however, a few disadvantages to this implementation. Chief among these is that the operating system is not aware that there are multiple threads running in the runtime's process. This is primarily an issue on systems that have multiple processors. Some of these systems can allocate additional processors to a task if they know that the task contains more than one thread--up to one processor per thread. But because these systems do not see the runtime as a multithreaded entity, multiprocessor allocation does not happen.

Typically, multiprocessor systems are used to provide support for more users. By having more processors in the system, the system can run more processes at once, and in this way increase the number of users the machine can effectively support. But the system will not increase performance for a single process (assuming that the process gets 100% of the CPU's time--an unrealistic assumption in a production environment). In practice, a multiprocessor machine will usually not benchmark any better than a single processor machine for single-task benchmarks, but they will provide much better total throughput in a multi-user production environment.

In summary, multithreading an ACUCOBOL-GT program will not affect its performance in any special way on a multiprocessor machine. Any gain, or loss, will be the same as for a single processor system.

6.8.8 Thread Interaction With Run Units

The following rules apply to the interaction of threads with run units:

1. The CHAIN and CALL PROGRAM statements normally act as if the program does a STOP RUN followed by the beginning of a new run unit. The implied STOP RUN destroys all threads other than the one executing the CHAIN or CALL PROGRAM statement.
2. If you have more than one thread running, you may not start a nested run unit (accomplished with the CALL RUN statement).

6.9 Working with External Sort Modules (UNIX)

The ACUCOBOL-GT runtime on UNIX platforms can call most third-party sort modules that support the EXTSM interface to perform SORT and MERGE operations. The EXTSM call interface allows you to substitute other sort modules that may enhance performance or offer more flexibility for sort features.

Sort modules can perform file handling using the EXTFH interface, which the ACUCOBOL-GT Runtime makes available to third-party sort modules. The sort module may also interface directly with the Vision library.

6.9.1 Before Using an External Sort Module

Depending on the sort and merge results you want to achieve, you may need to recompile programs and set a new environment variable:

- To use Micro Focus-style sign representation for data items, compile your programs with the “**-Dcm**” switch.
- To obtain the behavior of SORT...WITH DUPLICATES IN ORDER, set the following environment variable:

```
COBSW=+S
```

Note: You must set COBSW even if the COBOL program already specifies the WITH DUPLICATES IN ORDER phrase.

6.9.2 Linking in a Third-Party Sort Module

In order to use a third-party sort module with the ACUCOBOL-GT Runtime, you must relink the runtime using the following steps:

1. Install the third-party sort software.
2. Refer to the documentation for the EXTSM interface to understand its operation and use.

3. Make a copy of the lib/Makefile file in your ACUCOBOL-GT installation. You can use this backup copy to ensure that the relink of the runtime works before you make the changes to support the EXTSM interface.
4. Edit lib/Makefile to enable linking with the third-party sort library, as instructed in the Makefile section, “EXTSM configuration.”
5. Execute the **make** command to relink the runtime. For information on using the **make** command, see section, 6.3.6, “Creating a New Runtime System”, in *A Guide to Interoperating with ACUCOBOL-GT*.

After creating the new runtime, which includes the third-party sort module, set the **USE_EXTSM** configuration variable to “1” and then run programs as usual. For more information, refer to Appendix H, “Configuration Variables.”

You can then run programs with SORT and MERGE verbs run as they normally would. For information on these verbs, refer to the Procedure Division Statements section of the *ACUCOBOL-GT Reference Manual*.

Index

Symbols

\$ in Indicator Area 5-30

\$SET 2-73

%TMP%, using in place of a file name 2-111

Numerics

132-column mode, using escape sequences to switch 6-37

32-bit Windows, user-defined keys 4-35

88/Open COBOL specification 2-38

 required data storage option 2-42

A

-a option 3-70

A_SEQ_DEFAULT_BLOCK_SIZE configuration variable, and **vutil** 3-88, 3-89

A_TERM and TERM variables, on different systems 4-6

a_termcap 4-6

A_TMPDIR environment variable 1-20, 3-133

Abend Diagnostic Report

 configuration variables 3-56

 description 3-54

 generating 3-56

 restrictions 3-58

abort, graceful 6-56

About the Runtime, debugger menu option 3-45

ACCEPT

 ALLOWING MESSAGES 6-64, 6-66

 and DISPLAY video, compiler options 2-49

 and DISPLAY, example using Screen Section 6-51

 BEFORE TIME 6-66

- color 4-44
- debugger command 3-43
- menu selection command 3-43
- methods of accepting a field (Format 1) 4-14
- Screen Section 4-14, 4-52
 - example 6-49
- undefined location 4-60

ACCEPT IN HEX

- menu selection and debugger command 3-43

- ACU_DUMP configuration variable 3-56
- ACU_DUMP_FILE configuration variable 3-56
- ACU_DUMP_TABLE_LIMIT configuration variable 3-57
- ACU_DUMP_WIDTH configuration variable 3-57
- ACU_MON_FILE configuration variable 3-114
- Acu4GL 1-10
- ACU-CBLFLAGS 2-74
- ACUCOBOL-GT documentation overview 1-12

acuprof 3-112

- flags 3-115
- sorting report data 3-115

acurfap, described 2-69

- AcuServer 1-11, 5-16
 - requirements 5-17
 - with transaction management 5-15

acushare 2-121

- indicating programs to share 2-123
- kill 2-125
- license error messages 2-127
- runtime errors 2-127
- starting 2-124

AcuSort utility 3-121

- command-line format 3-122
- comparison expressions 3-129
- conditional constants 3-128
- data field types 3-124
- designating input file and output files 3-125
- environment variables 3-133

- key types 3-127
- log file 3-134
- substring search 3-130
- take file format 3-122
- version information 3-122
- AcuSort** utility instructions 3-123
 - CHAR-EBCDIC instruction 3-123
 - GIVE instruction 3-125
 - INCLUDE instruction 3-128
 - MERGE instruction 3-124
 - OMIT instruction 3-128
 - SIGN-EBCDIC instruction 3-123
 - SORT instruction 3-124
 - USE instruction 3-125
- AcuSort** utility phrase
 - FIELDS phrase 3-124
 - KEY phrase 3-127
- ACUSORT_FILE_MEMORY environment variable 3-133
- ACUSORT_MEMORY environment variable 3-134
- ACUSORT_TRACE environment variable 3-134
- AcuSQL calling options 2-136
- AcuXML, Internet file mapping 2-102
- addresses, specifying in the debugger 3-51
- alfred, indexed file record editor 3-107
- alias file
 - CICS and **Boomerang** 3-141
 - creating with **Boomerang** 3-137
 - DB2 and **Boomerang** 3-143
 - Pro*COBOL and **Boomerang** 3-140
 - UniKix and **Boomerang** 3-142
- alias for file name 2-101
- allocating dynamic memory 3-21
- ALLOWING MESSAGES phrase, ACCEPT statement 6-64
- ALPHA directive 5-31
- alternate ENTRY points in a program 2-119
- Alternate Terminal Manager (ATM) runtime 4-4
- ANSI

- compatibility options, compiler 2-15
- source format 2-93
 - compiler options 2-30
- ANSI/terminal format 2-73
- ANSI-compliant directive 5-30
- apostrophe and QUOTE literal 2-39
- APPLY_CODE_PATH configuration variable 2-117
- array references, compiler option to test 2-61
- arrow keys 4-37
- Asian character sets 4-60
- ASSIGN phrase 2-17
 - file name interpretation 2-107
 - reassigned with configuration variable 2-101
 - WITH COMPRESSION 6-17
- assigning files to local printers 2-114
- ATM runtime 4-4
- auto step 3-23, 3-46
 - debugger 3-14
- AUTO_PROMPT configuration variable 4-58
- automatic blank stripping, in sequential files 6-4
- AXDEFGEN utility 2-74

B

- background color, compiler option 2-50
- backspace key 4-12, 4-34, 4-38
- backup, **vio** archive utility 3-98
- backwards compatibility, compiler options 2-18
- BELL runtime configuration variable 4-58
- BELL, compiler option to inhibit 2-50
- BINARY data item storage, compiler option 2-34
- BINARY data item, treating as COMPUTATIONAL-5 2-42
- BINARY directive 5-32
- binary sequential files
 - block size, and 6-3
 - compatibility options, compiler 2-15

- creating an indexed file from 3-89
- uploading from Vision files 3-87
- binary sequential records 6-3
- block size, Vision files 6-13
- blocking factor, setting with **vutil** 3-73
- Boomerang** 3-135
 - alias file 3-137
 - CICS alias creation 3-141
 - client-side commands 3-148
 - client-side operation 3-147
 - configuration 3-145
 - DB2alias creation 3-143
 - INCLUDE files 3-150
 - Pro*COBOL alias 3-140
 - server commands 3-146
 - server setup 3-136
 - UniKix alias 3-142
- breakpoints 3-6
 - conditional 3-39
 - menu in debugger 3-35
 - saving 3-35
 - threads 3-24
 - viewing 3-12, 3-21
- bulk addition, described 6-21

C

- C\$KEYPROGRESS routine 6-27
- C\$LOCALPRINT routine 6-37
- C\$RECOVER routine 5-5, 5-14
- CALL
 - loading object libraries 3-59
 - runtime resolution process 2-115
- CALL THREAD 6-58
- calling preprocessors 2-132
 - without the compiler 2-134

- calling subprograms 2-115
- calling the AcuSQL pre-compiler 2-136
- CANCEL statement
 - effects described 2-118
 - to free memory 2-115
- CARRIAGE_CONTROL_FILTER configuration variable 2-107
- case, lower/upper in command line 2-68
- catastrophic exit 6-56
- cblconfig, runtime configuration file 2-101
- CBLFLAGS environment variable 2-71
- cblutil** program 3-60
 - native option 3-64
 - object file information command 3-63
 - object library options 3-60
- char2gui runtime option 2-78
- character-based emulation of windows and controls 4-78
- characters, double-byte 4-60
- Character-to-GUI Wizard 2-64, 2-78
- CHAR-EBCDIC instruction 3-123
- CHECK-NUMBERS=Validate 6-55
- CICS, **Boomerang** alias creation 3-141
- C-ISAM files, converting with **vutil** 3-92
- clear all breakpoints debugger command 3-12
- clear all variable monitors debugger command 3-12
- clear all watches debugger command 3-12
- clear breakpoint debugger command 3-12, 3-40
- clear variable monitor debugger command 3-12
- CLOSE statement, effects on record locking 6-15
- CLOSE_ON_EXIT configuration variable 2-106
- CMD-ACTIVATE event 6-66
- CMD-GOTO event 6-55
- CMD-TABCHANGED event 6-55
- COBOL dialect compatibility modes 2-92
- COBOL-Trigger directive 5-33
- COBSW environment variable 6-71
- code and data file search paths 2-103
- code generation

- host processor, compiler option 2-7
- Intel processor, compiler option 2-7
- Sparc processor, compiler option 2-9
- Sparc_v9 processor, compiler option 2-9
- code sharing 2-122
- CODE_CASE configuration variable 2-116
- CODE_PREFIX configuration variable 2-103, 2-117
- CODE_SUFFIX configuration variable 2-104, 2-117
- color
 - adding to monochrome displays 4-43
 - disabling on MS-DOS 4-58
 - map, and SET ENVIRONMENT 4-45
 - setting foreground and background 4-74
 - terminal can display only one color at a time 4-75
- COLOR_MAP configuration variable 4-43
- columns, and horizontal scroll 4-56
- commands, notation of 1-20
- COMMENT directive 5-35
- comment field in a Vision file 3-97
- COMMIT TRANSACTION, implicit 5-6
- comparison expressions 3-129
- compatibility modes, compiler 2-92
- compatibility options, compiler
 - ANSI 2-15
 - binary sequential 2-15
 - compatibility with prior versions 2-18
 - default extension 2-16
 - HP COBOL 2-17
 - IBM DOS/VS COBOL 2-18
 - IBM/COBOL, ASSIGN phrase of SELECT 2-17
 - ICOBOL 2-16
 - key number generation 2-17
 - pre-85 compiler 2-16
 - RM/COBOL 2-18
- compiler data storage options
 - alignment boundary 2-34
 - alignment of SYNCHRONIZED data items 2-38, 2-74

- BINARY 2-34
- BINARY STORAGE 2-43
- COMP-1 and COMP-2 treatment 2-37
- COMPUTATIONAL-2 2-42
- COMPUTATIONAL-5 2-42
- different numeric format 2-36
- floating point 2-37
- IBM 2-35
- initializing Working-Storage by type 2-38
- LINKAGE data item alignment 2-35
- Micro Focus 2-35
- minimum bytes 2-38
- modifying definition of data types 2-40
- one byte 2-42
- PACKED-DECIMAL 2-43
- reference modification 2-46
- RM/COBOL 2-38
- SIGN clause 2-39
- size checking rule modification 2-42
- treating binary as SYNCHRONIZED 2-42
- VAX COBOL 2-36
- compiler debugging options
 - all-level 2-52
 - by line-number 2-52
 - extra symbol 2-53
 - list of 2-52
 - minimal symbol 2-53
 - source-level 2-52
- compiler directives. *See* directives
- compiler file name handling options
 - @ character in file names 2-68
 - abbreviation of file names 2-68
 - See also* compiler remote file name handling options
- compiler introduction 2-2
- compiler mapping options
 - COPY file list 2-56
 - data items, report 2-56

- list of 2-54
- list of special styles and properties 2-55
- lowercase text 2-55
- map report 2-55
- mixed-case text 2-55
- Procedure Division report 2-55
- report of reserved words 2-55
- source file list 2-56
- uppercase text 2-56
- compiler native code options
 - for host machine 2-7
 - for Intel-class processors 2-7
 - for Sparc v9-class processors 2-9
 - for Sparc-class processors 2-9
- compiler options
 - ANSI 2-15
 - array references test 2-61
 - as if INITIAL PROGRAM 2-62
 - binary sequential 2-15
 - CENTURY-DATE, DAY 2-65
 - compatibility with prior versions 2-18
 - creating a cross-reference 2-9
 - creating XFD files 2-24
 - data dictionaries 2-23
 - building XFD 2-28
 - data items larger than 64 KB 2-62
 - default extension 2-16
 - directory for data dictionary files 2-27
 - error output 1-21
 - errors, write to file 2-4
 - extended statistics 2-13
 - file handle size 2-29
 - full listing, create 2-11
 - HP COBOL 2-17
 - IBM DOS/VS COBOL 2-18
 - IBM/COBOL, ASSIGN phrase of SELECT 2-17
 - ICOBOL 2-16

- identifier and statement internal tables 2-14
- ignore CBLFLAGS environment variable 2-5
- include source lines 2-99
- indexing tables 2-63
- key number generation 2-17
- matching field names, XFD files 2-24
- miscellaneous options, list of 2-61
- no recursive PERFORMs 2-63
- object file name 2-4
- output file for listings 2-12
- page length of listings 2-11
- pre-85 compiler 2-16
- recursive PERFORMs 2-63
- reference modification 2-46
- remote object file name 2-4
- RM/COBOL 2-18
- runtime, backward compatibility 2-66
- screen import utility 2-64
- segmentation in source 2-62
- single locking 2-25
- START TRANSACTION, implied 2-27
- subscript statement internal tables 2-14
- summary listing, create 2-11
- symbol tables 2-13
- tab stops 2-30
- transaction files 2-23
- transaction management 5-13
- transaction, implied 2-28
- treating spaces as zero 2-65
- turning off local optimizer 2-62
- USAGE DISPLAY 2-65
- verbose 2-5
- warning message suppression 2-5, 2-82
- wide format 2-13
- XFD files, creating 2-23

compiler options, listed

- compatibility options 2-15

- file options 2-23
- internal tables options 2-14
- listing options 2-9
- native code options 2-7
- preprocessor options 2-133
- source options 2-30
- compiler preprocessors 2-129
- compiler remote file name handling options
 - @ character in remote file names 2-69
 - abbreviation of remote file names 2-69
 - See also* compiler file name handling options
- compiler reserved words options
 - changing 2-33
 - ignoring 2-34
 - list of 2-32
 - suppressing 2-34
- compiler source format options
 - ANSI 2-30
 - compiler directive 2-74
 - terminal 2-32
- compiler source options
 - ANSI 2-30
 - COPY library directories 2-31
 - debugging lines 2-31
 - exclude source lines 2-32
 - include source lines 2-31
 - input source, encoding scheme 2-30
 - RM/COBOL tab stops 2-31
 - tab stops, RM/COBOL 2-31
- compiler truncation options 2-46
- compiler video options
 - background color 2-50
 - BLANK EOL 2-48
 - exception keys 2-49, 2-50, 2-51
 - high intensity 2-49
 - inhibiting bell 2-50
 - intensity 2-48

- low intensity 2-50
- ON EXCEPTION 2-49
 - treating as if CONVERT specified 2-48
- compiler, using 2-2
 - examples 2-68
 - handling problems 1-21
- COMPRESS_FACTOR configuration variable 6-18
- compression
 - configuration file option 6-18
 - described 6-17
 - using WITH COMPRESSION 6-18
 - Vision files 6-14
- compression factor, option to set 3-93, 3-95
- COMPUTATIONAL 2-34
- COMPUTATIONAL-1 2-37
- COMPUTATIONAL-2 2-37, 2-42
- COMPUTATIONAL-2 data items, compiler option 2-42
- COMPUTATIONAL-5 2-42
- COMPUTATIONAL-6 2-43
- COND phrase 3-128
- conditional compiling
 - compiler options 2-57
- config85.c 2-120
- configuration file 1-2
 - embedded in an object 2-81
 - information 2-72
 - runtime option for listing of 2-85
 - specifying an alternate 2-78
 - versus user's environment 2-107
- configuration options that can affect file operations and performance 2-106
- configuration variables 2-100, 2-108
 - defining and displaying in the debugger 3-52
 - display related 4-58
 - runtime 2-100
- configuration variables, list of
 - ACU_DUMP_FILE 3-56
 - ACU_DUMP_TABLE_LIMIT 3-57

ACU_DUMP_WIDTH 3-57
ACU_MON_FILE 3-114
APPLY_CODE_PATH 2-117
AUTO_PROMPT 4-58
BELL 4-58
CARRIAGE_CONTROL_FILTER 2-107
CLOSE_ON_EXIT 2-106
CODE_CASE 2-116
CODE_PREFIX 2-103, 2-117
CODE_SUFFIX 2-104, 2-117
CODE-PREFIX 2-117
COLOR_MAP 4-43
COMPRESS_FACTOR 6-18
DEFAULT_MAP_FILE 5-65
DEFAULT_PROGRAM 2-75
ERRORS_OK 2-105
EXPAND_ENV_VARS 2-108
EXTEND_CREATES 2-105
F10_IS_MENU 4-37
FILE_ALIAS_PREFIX 2-108
FILE_CASE 2-108
FILE_PREFIX 2-103, 2-112
FILE_STATUS_CODES 2-104, 2-105
FILE_SUFFIX 2-104, 2-109
*filename*_LOG 5-11
IO_CREATES 2-105
KBD_AUTO_RETURN 4-16
KBD_CASE 4-17
KBD_CHECK_NUMBERS 4-17
KBD_CURSOR_PAST_END 4-18
KBD_DATA_RANGE_HIGH 4-18
KBD_DATA_RANGE_LOW 4-18
KBD_EXCEPTION_RANGE_HIGH 4-18
KBD_EXCEPTION_RANGE_LOW 4-18
KBD_IMPLIED_DECIMAL 4-19
KBD_RM_2_DEFAULT_HANDLING 4-19
KBD_SCREEN_DEFAULT 4-19

KEYBOARD 4-16
KEYSTROKE 4-16, 4-19
LOCK_DIR 2-107
LOG_BUFFER_SIZE 5-11
LOG_DEVICE 5-11
LOG_DIR 5-11
LOG_ENCRYPTION 5-11
LOG_FILE 5-9, 5-11
LOGGING 5-11
MAKE_ZERO 2-89
MASS_UPDATE 2-106
MONOCHROME 4-58
RESTRICTED_VIDEO_MODE 4-59, 4-62
SCREEN 4-46
SCRN_ALPHA_AUTO_PROMPT 4-47
SCRN_ALPHA_UPDATES 4-47
SCRN_CONVERT_OUTPUT 4-47
SCRN_EDITED_AUTO_PROMPT 4-49
SCRN_EDITED_UPDATES 4-49
SCRN_ERROR_BELL 4-49
SCRN_ERROR_BOX 4-50
SCRN_ERROR_COLOR 4-51
SCRN_ERROR_LINE 4-51
SCRN_FORM_FEED 4-51
SCRN_INPUT_DISPLAY 4-52
SCRN_INPUT_MODE 4-52
SCRN_JUSTIFY 4-52
SCRN_NUMERIC_AUTO_PROMPT 4-53
SCRN_NUMERIC_UPDATES 4-53
SCRN_PROMPT 4-53
SCRN_PROMPT_ALL 4-53
SCRN_PROMPT_ATTR 4-54
SCRN_PROMPT_DEFAULT 4-53
SCRN_REFRESH_LINES 4-54
SCRN_REFRESH_MODE 4-55
SCRN_SHADOW_STYLE 4-56
SCRN_SIZE_COLS 4-56

- SCRN_SIZE_ROWS 4-56
- SCRN_WARN 4-49
- SCRN_WINDOW_X 4-57
- SCRN_WINDOW_Y 4-57
- SCROLL 4-58, 4-59
- server_MAP_FILE 5-65
- SORT_DIR 2-106, 2-111
- STOP_RUN_ROLLBACK 5-7
- TEMP_DIR 2-111
- transaction logging, used with 5-11
- V_BULK_MEMORY 6-22
- WARNINGS 2-89
- WRAP 4-58, 4-59
- XFD_DIRECTORY 5-65
- console runtime 1-6
- contents command 3-45
- continue command 3-22
- CONTROL KEY clause 4-14, 4-16, 4-18, 4-28, 4-31
- control keys, how to define 4-34
- controls
 - character-based emulation, default characters 4-78
 - key letter, treatment in text mode 4-80
 - properties and styles, compiler listing 2-55
 - redisplaying a moved control 4-81
 - removing runtime support for 2-120
- CONVERT phrase, compiler/video option 2-48
- converting
 - C-ISAM files 3-92
 - Micro Focus Files 3-94
- COPY file list, compiler option 2-56
- COPY libraries 2-95
 - directories, compiler option 2-31
 - excluding a library from a COPY statement 2-97
 - specifying the name of 2-96
- COPY REPLACING with XFDs 5-19
- COPY RESOURCE statement 2-96
- COPYPATH environment variable 2-96

creating

- empty files 3-81
- object libraries 3-60
- remote object libraries 3-62
- XFD files, compiler option 2-24

cross-reference list, compiler option 2-9

current line command 3-25

cursor

- mouse handling in source-level debugging 3-7
- undefined state 4-60

D

-d runtime option, to start debugger 3-5

data

- alignment modulus 2-34
- alternate data alignment 2-74
- data items report, compiler option 2-56
- large data items, compiler option 2-62
- validation 6-55
 - when not performed 6-55

data alignment 2-73

data dictionaries 5-19

file name 5-39

XFD, file options, compiler 2-28

data execution protection, Windows 2-7

data field types 3-124

data item alignment 2-35

data items, 31-digit support 2-37

Data menu in debugger 3-28

data sharing among threads 6-60

data storage

- alignment of SYNCHRONIZED data items 2-38
- BINARY 2-34
- BINARY STORAGE 2-43
- COMP-1 and COMP-2 treatment 2-37

- COMPUTATIONAL-2 2-42
- COMPUTATIONAL-5 2-42
- data alignment modulus 2-34
- different numeric format for 2-36
- floating-point 2-37
- IBM 2-35
- initializing Working-Storage by type 2-38
- Micro Focus 2-35
- minimum bytes 2-38
- modifying definition of data types 2-40
- one byte 2-38, 2-42
- PACKED-DECIMAL 2-43
- Realia 2-36
- SIGN clause 2-39
- size checking rule modification 2-42
- SYNCHRONIZED, for treating binary data as 2-42
- VAX COBOL 2-34, 2-36
- DATE directive 5-35
 - FY and RY formats 5-38
- date format, -Zy option 2-65
- DB2, **Boomerang** alias creation 3-143
- DBCS 4-60
- dd_SYSOUT environment variable 3-135
- debugger 3-2
 - activating the System menu 3-9
 - command function keys 3-44
 - configuration variables, setting and displaying 3-52
 - cursor position in source code 3-25
 - entering 3-5
 - example 1-23
 - exiting 3-20
 - file tracing option 3-49
 - low-level debugging 3-4
 - macros 3-50
 - multithreading issues 3-15
 - name qualification 3-52
 - running under windows 3-5

- screen tracing option 3-49
- scrolling 3-43
- searches 3-26
- source debugging 3-3
- specifying addresses 3-51
- specifying program addresses 3-53
- specifying variables 3-51
- symbolic debugging 3-4
- three modes 3-3
- toolbar 3-45
- debugger commands 3-8
- debugger file tracing option 3-47
- debugger menus
 - breakpoints 3-35
 - data 3-28
 - file 3-17
 - help 3-45
 - Monitor submenu 3-32
 - run 3-22
 - selection 3-41
 - source 3-25
 - view 3-20
- debugger restrictions 3-53
- debugger scroll bar 3-3
- debugger window 3-5
- debugger, using
 - in background mode 3-4
 - mouse with 3-7
 - with application servers 3-4
- debugging mode 2-79
- debugging options 2-52
- decimal ASCII 4-28
- Declarative 2-105
- default extension, compatibility options, compiler 2-16
- DEFAULT_MAP_FILE configuration variable 5-65
- DEFAULT_PROGRAM configuration variable 2-75
- defines compiler option 2-72

- DELETE and record locking 6-15
- deleted records, recovering 3-80
- DEP 2-7
- device locking 2-107
 - under UNIX 6-16
- device naming conventions 2-102
- devices 2-110
- directives 2-73
 - >>IMP 2-73
 - Alpha 5-31
 - binary 5-32
 - COBOL-trigger 5-33
 - comment 5-35
 - date 5-35
 - file 5-39
 - line and file with preprocessor 2-138
 - name 5-40
 - numeric 5-43
 - secondary-table 5-44
 - subtable 5-45
 - syntax 5-30
 - use group 5-46
 - var-length 5-47
 - when 5-48
 - xsl 5-54
- disable at cursor line debugger command 3-40
- DISPLAY
 - color 4-44
 - Screen Section and CONVERT-OUTPUT 4-47
 - undefined location 4-60
- display
 - attributes, order of precedence 4-44
 - configuring 4-42
- DISPLAY BOX 6-37
- display command
 - data menu 3-12, 3-30
 - selection menu 3-41

- display in hex debugger command
 - data menu 3-31
 - selection menu 3-42
- display interface 4-42
- DISPLAY LINE 6-37
- display, debugger command 3-30, 3-41
- DISPLAY_SWITCH_PERIOD configuration variable 6-65
- DOUBLE, floating-point precision compiler option 2-37
- double-byte character handling 4-60
- Down, in debugger 3-43
- duplicate records, loading with vutil 3-91
- dynamic memory 3-21
- DYNAMIC_FUNCTION_CALLS configuration variable 2-116

E

- embedded-config-file compiler option 2-81
- embedded procedures 6-52
- emulating graphical controls on character-based systems 4-78
- enable at cursor line debugger command 3-40
- encoding input source, scheme assumed by compiler 2-30
- encryption 6-14
- ENTRY points, call resolution 2-116
- ENTRY statement 2-119
- environment variables 1-18
 - CBLFLAGS 2-71
 - in the path-name 2-96
 - to define a path 2-95
- environment, file name search in 2-108, 2-110
- error file, compiler options 2-4
- error handling 6-42
- error messages, preprocessors 2-141
- error output 2-80
 - compiler option 1-21
- errors
 - acushare** 2-127

- logging to runtime's error file 6-43
- ERRORS_OK configuration variable 2-105
- escape sequences, terminal manager 6-36
- examining file information 3-67
- EXCEPTION clause 4-18, 4-28
- exception keys 4-14
 - compiler option 2-50, 2-51
- exclude source lines, compiler option 2-32
- Exit Debugger debugger command 3-20
- exiting from ACUCOBOL-GT programs 6-56
- EXPAND_ENV_VARS configuration variable 2-108
- EXTEND_CREATE configuration variable 2-105
- extended statistics listing options, compiler 2-13
- external data item 6-41
- external sort function 3-121
- EXTFH interface, external sort modules 6-71
- extracting records from a file 3-79
- EXTSM interface
 - linking into the runtime 6-71
 - support for 6-70

F

- f option 3-70
- F10_IS_MENU configuration variable 4-37
- FIELDS phrase 3-124
- FILE directive 5-39
- file error 30, runtime option 2-90
- file integrity 3-69
- file locks
 - fn compiler option 2-26
- File memory, debugger command 3-21
- File menu in debugger 3-17
- file name
 - abbreviation when compiling 2-68
 - aliasing 2-101

- assignments 2-101
- case of 2-68
- dynamic reassignment 2-107
- examples 2-111
- FILE_NAME_PREFIX configuration variable 2-107
- handling of 2-68
- interpretation of 2-107
- remote, handling of 2-69
- starting with hyphen 2-108, 2-110
- file options, compiler
 - data dictionaries 2-23
 - directory for data dictionary files 2-27
 - matching field names, XFD files 2-24
 - single locking 2-25
 - START TRANSACTION, implied 2-27
 - transaction, implied 2-28
 - transactions 2-23
 - XFD files 2-23
 - XFD, building data dictionaries 2-28
- file size, summary report 3-92
- file tracing 1-24, 3-47, 3-49
 - flushing output 3-48
 - timestamps 3-49
- file utilities
 - vio** 3-98
 - vutil** 3-66
- FILE_ALIAS_PREFIX configuration variable 2-108
- FILE_CASE configuration variable 2-108
- FILE_PREFIX configuration variable 2-103, 2-112
- FILE_STATUS_CODES configuration variable 2-104, 2-105
- FILE_SUFFIX configuration variable 2-104, 2-109
- FILE_TRACE configuration variable 3-48
- FILE_TRACE_TIMESTAMP configuration variable 3-49
- File-Control paragraph 5-20
- fileIdSize compiler option 2-29
- filename_LOG configuration variable 5-11
- files

-
- binary sequential
 - creating an indexed file from 3-89
 - uploading from Vision files 3-87
 - C-ISAM, converting with **vutil** 3-92
 - compression 6-14, 6-17
 - encryption 6-14
 - examining file information 3-67
 - handling 2-105, 6-2
 - improving performance 6-32
 - limits on open files and control of 6-34
 - locking, performance considerations 6-33
 - Micro Focus, converting with **vutil** 3-94
 - preventing fragmentation with **vutil** 3-74
 - rebuilding 3-71
 - recovery routine for 5-5, 5-14
 - reducing number of 3-59
 - resource, adding to an object library with **cblutil** 3-60
 - runtime option for input from 2-83
 - sequential 6-2
 - status codes 2-104
 - translation, examples of rules 2-111
 - vutil** integrity check 3-69
- FILLER data items and XFDs 5-25
- Find Backwards debugger command 3-26
- Find debugger command 3-46
- Find Forwards debugger command 3-26
- Find from Top debugger command 3-26
- Find Next debugger command 3-46
- Find Previous debugger command 3-46
- fixed length records, compiler option 2-16
- FLOAT 2-37
- format
 - compiler option for ANSI source 2-30
 - compiler option for terminal source 2-32
 - of source code 2-93
- form-level ACCEPT and DISPLAY 6-51
- fragmentation of files, preventing with **vutil** 3-74

- FULL phrase 6-55
- function codes for terminals, list of 4-66
- function keys
 - debugger 3-44
 - defining 4-75

G

- general preprocessor interface 2-129
- generating MOVE code for LINKAGE data items 2-35
- GF-GUI-MAP 4-79
- GIVE instruction 3-125
- Go debugger command 3-46
- Go to cursor line debugger command 3-13, 3-22
- Go until paragraph returns debugger command 3-22
- Go until program exits debugger command 3-22
- GO-GUI-MAP 4-79
- graphical window and control emulation 4-78
- graphics on terminals requiring multiple-character escape sequences 4-77
- group items 5-25

H

- hang up signal, blocking on UNIX 2-78
- hardware supported 1-17
- help automation, mapping context IDs 2-73
- help command 2-72
- help on help command 3-45
- help with debugger 3-16
- high-intensity video, compiler option 2-49
- hot keys 4-29
 - assigning 4-29
 - PRNTSCRN 4-29
- HOT_KEY configuration variable 4-58
- HP COBOL compatibility, compiler option (-Cp) 2-17
- hyphens, special treatment of file names with 2-110

I

- I/O efficiency 5-46
- IBM data storage, compiler option 2-35
- IBM DOS/VS COBOL
 - ASSIGN phrase of SELECT, compatibility options, compiler 2-17
 - compatibility options, compiler 2-18
 - using SUPPRESS in a COPY statement 2-97
- ICOBOL
 - compatibility, compiler options 2-16
 - file status codes, and 2-104
 - reserved words option 2-33
 - terminal source format 2-94
- identifier and statement, internal tables options, compiler 2-14
- ignore CBLFLAGS environment variable, standard options, compiler 2-5
- importing screens and controls 2-84
- INCLUDE instruction 3-128
- index values, reusing 2-62
- indexed file record editor (alfred) 3-107
- indexed file utility (**vutil**) 3-66
- indexed files 6-5
 - features 6-17
- info options, listed 3-67
- initialization strings, for terminals 4-83
- input from a file, runtime option for 2-83
- input source, encoding scheme assumed by compiler 2-30
- intermediate runtime errors 5-12
- internal tables options, compiler
 - explained 2-13
 - identifier and statement 2-14
- interrupt key 4-39
- IO_CREATES configuration variable 2-105
- IO_SWITCH_PERIOD configuration variable 6-65
- IS INITIAL PROGRAM phrase, compiler option 2-62

J

- javaclass compiler option 2-21
- JUSTIFIED, with formatted and centering option 4-49

K

- k option 3-67, 3-70
- KBD_AUTO_RETURN configuration variable 4-16
- KBD_CASE configuration variable 4-17
- KBD_CHECK_NUMBERS configuration variable 4-17
- KBD_CURSOR_PAST_END configuration variable 4-18
- KBD_DATA_RANGE_HIGH configuration variable 4-18
- KBD_DATA_RANGE_LOW configuration variable 4-18
- KBD_EXCEPTION_RANGE_HIGH configuration variable 4-18
- KBD_EXCEPTION_RANGE_LOW configuration variable 4-18
- KBD_IMPLIED_DECIMAL configuration variable 4-19
- KBD_RM_2_DEFAULT_HANDLING configuration variable 4-19
- KBD_SCREEN_DEFAULT configuration variable 4-19
- key codes 4-31
- key fields 5-20
 - by either explicit or implicit redefinition 5-23
- key interpretation 4-10
- key letter
 - designating on character based systems 4-22
 - treatment in text mode 4-80
- key mapping 4-10
- key number generation, compatibility options, compiler 2-17
- KEY phrase 3-125, 3-127, 5-20
 - XFDs, and 5-23
- key translation 4-11
- KEYBOARD
 - AUTO-RETURN keyword 4-16
 - CASE keyword 4-17
 - CHECK-NUMBERS keyword 4-17
 - CURSOR-PAST-END keyword 4-17
 - DATA-RANGE keyword 4-18

-
- EXCEPTION-RANGE keyword 4-18
 - IMPLIED-DECIMAL keyword 4-18
 - RM-2-DEFAULT-HANDLING keyword 4-19
 - SCREEN-DEFAULT keyword 4-19
 - keyboard
 - additions under 32-bit Windows 4-35
 - configuration, default 4-12
 - default, ACUCOBOL-GT 4-39
 - functions, and Terminal Manager 4-9
 - interface 4-9
 - modification examples 4-41
 - redefining 4-16
 - KEYBOARD configuration variable 4-16
 - keys 6-11
 - code example 6-11
 - information, retrieving with C\$KEYPROGRESS 6-27
 - list of function codes to represent 4-65
 - performance considerations 6-33
 - sample code 6-11
 - split 6-12
 - table of actions 4-12
 - table of redefinable keys 4-31
 - with more than one name 4-34
 - KEYSTROKE
 - AT-END keyword 4-20
 - DATA keyword 4-20
 - EDIT keyword 4-21
 - values for 4-22–4-28
 - EXCEPTION keyword 4-28
 - HOT-KEY keyword 4-29
 - INVALID keyword 4-31
 - table of redefinable keys 4-31
 - TERMINATE keyword 4-31
 - KEYSTROKE configuration variable 4-16, 4-19
 - keystroke, playback of keystroke file 2-85
 - kill -9 6-2, 6-56

L

- large data items, compiler option 2-62
- last line debugger command 3-25
- LAST THREAD 6-59
- library routines
 - C\$KEYPROGRESS 6-27
 - C\$LOCALPRINT 6-37
 - C\$RECOVER 5-5, 5-14
 - M\$ALLOC 3-21
 - W\$BITMAP 2-98
 - WIN\$PLAYSOUND 2-98
- license error messages, displaying with LICENSE_ERROR_MESSAGE_BOX 2-127
- license files, AcuServer 5-17
- limits on open files 6-34
- line 1 debugger command 3-25
- line drawing 4-76
 - and terminal manager 6-37
- line number, reported when abnormal termination 2-52
- line segments requiring multiple-character escape sequences 4-77
- line sequential files
 - ACUCOBOL-GT rules 6-3
 - creating indexed files from 3-89
 - two types 6-4
 - uploading from Vision files to 3-87
- line wrapping 4-59
- LINK TO THREAD 6-66
- linking the runtime 2-120
- listing options, compiler
 - cross-reference, create 2-9
 - extended statistics 2-13
 - full listing, create 2-11
 - output file for 2-12
 - page length of listings 2-11
 - summary listing, creating 2-11
 - symbol tables 2-13
 - wide format 2-13

- loading a file 3-89
- local printers 6-37
 - assigning files to 2-114
- LOCALPRINT 2-114
- Location, debugger command 3-38
- LOCK THREAD statement 6-61
- LOCK_DIR configuration variable 2-107, 6-16
- locked records, RM/COBOL compatibility mode 6-15
- locking
 - rules for transaction management 5-7
 - threads 6-60
- locking files, -fn compiler option 2-26
- log files, multiple 5-10
- LOG_BUFFER_SIZE configuration variable 5-11
- LOG_DEVICE configuration variable 5-11
- LOG_DIR configuration variable 5-11
- LOG_ENCRYPTION configuration variable 5-11
- LOG_FILE configuration variable 5-9, 5-11
- logging
 - configuration file variables used with 5-11
 - rollback of file update operations, and 5-9
 - specifying log files for 5-10
- LOGGING configuration variable 5-11
- logutil**
 - log file editor 3-107
 - options 3-107
 - report headings 3-110
 - syntax 3-107
- lost records 3-71
- low-intensity video, compiler option 2-50
- low-level debugging 3-4

M

- M (monitor) debugger variable 3-13
- M\$ALLOC routine 3-21

- macro debugger 3-50
- magic cookie terminals, and attribute settings 4-59, 4-73
- MAKE_ZERO configuration variable 2-89
- mapping options 2-54
- MASS_UPDATE configuration variable 2-106
- MASS-UPDATE phrase, described 6-19
- MBP COBOL sign storage 2-35
- memory
 - shared 2-121
 - types of 3-21
- memory access violations 6-42
- memory usage debugger command 3-21
- MERGE instruction 3-124
- MERGE statement, external sort modules 6-70
- merging records 3-124
- messaging, among threads 6-61
- Micro Focus
 - data storage, compiler option 2-35
 - files, converting with **vutil** 3-94
 - reserved word option 2-33
- minimizing the application window 4-57
- modeless window
 - managing with threads 6-57, 6-66
 - sample project 6-68
- monitor variables, in debugger 3-32
 - monitor option 3-42
 - stop when value changes 3-13
 - versus watch 3-33
- MONOCHROME configuration variable 4-58
- mouse
 - support for X terminals 4-82
 - using with debugger 3-7
- mouse-action keys 4-34
- multiple execution threads 6-57
- multiple log files 5-10
- multiple record definitions and XFDs 5-23
- multithreading 6-57, 6-69

N

- name aliasing for files 2-101
- NAME directive 5-40
- naming the XFD 5-27
- native code
 - generating for Intel-class processors 2-7
 - generating for Sparc v9-class processors 2-9
 - generating for Sparc-class processors 2-9
 - list of options 2-7
 - option in **cblutil** 3-64
 - options, explained 2-5
 - overview 1-4
 - supported processors 1-18
- netdll compiler option 2-23
- netexe compiler option 2-22
- no recursive PERFORMs compiler option 2-63
- notation for commands 1-20
- NUMERIC directive 5-43

O

- object file
 - cblutil** 3-58
 - command to get information 3-63
 - library, runtime option for 2-91
 - name, standard options, compiler 2-4
 - native code 3-64
 - utility 3-58
- object library 3-59
 - creating 3-60
 - remote 3-62
- object module, three states 2-115
- object-code file 2-2
- OCCURS clause, and XFDs 5-25
- OMIT instruction 3-128
- ON EXCEPTION phrase, compiler option to alter rules for 2-49

- open files, limits on and control of 6-34
- OPEN statement, MASS-UPDATE option 6-19
- operating systems supported 1-17
- optimizer, compiler option to turn off local 2-62
- optimizing data storage 2-40
- OPTIONAL phrase 2-27
- options
 - compiler, introduction 2-2
 - runtime 2-76
- ORG phrase 3-125
- OTHER 5-49
- output file for listing options, compiler 2-12
- overhead memory, debugger command 3-21
- overlays in source, compiler option 2-62

P

- p option 3-67
- PACKED-DECIMAL data items, treating as COMPUTATIONAL-6 2-43
- page length of listings, listing options, compiler 2-11
- paragraph command 3-25
- perform stack command, debugger 3-11, 3-20
- perform step, in debugger 3-23
- PERFORM THREAD 6-58
- performance
 - improving 3-59, 3-68, 6-19, 6-32, 6-35
 - with shared memory 2-121
 - V-BUFFER considerations 6-33
- PICTURE clause, 31-digit support 2-37
- pipe 2-110
- portability
 - of XFDs 5-19
 - option for maximizing 2-41
- pre-85 compiler, compatibility options, compiler 2-16
- pre-compiler
 - AcuSQL, calling options 2-136

- invoking 2-136
- invoking from ACUCOBOL-GT 2-136
- options 2-136
- remote processing with **Boomerang** 3-135
- preprocessed output 2-12
- preprocessor, remote processing with **Boomerang** 3-135
- preprocessors
 - calling the AcuSQL pre-compiler 2-136
 - calling two or more 2-132
 - calling without the compiler 2-134
 - command-line options 2-136
 - compiler options 2-133
 - error messages 2-141
 - general interface 2-129
 - line and file directives 2-138
 - pre-compiler options 2-136
 - use of 2-130
 - writing a preprocessor 2-136
 - written in ACUCOBOL-GT 2-131
 - written in C/C++ 2-130
- print files 6-4
- print functions, and Terminal Manager 4-84
- printers 2-107
 - accessing 2-113
 - ASSIGN clause, and 2-102
 - assigning files to local 2-114
 - attaching to terminal 4-84
 - file name assignments 2-102
 - machine-independent access of 2-113
 - sending data to local device 6-37
- printing, and UNIX 6-17
- PRNTSCRN 4-29
- Pro*COBOL **Boomerang** alias 3-140
- Procedure Division report, compiler option 2-55
- processors, native code supported 1-18
- product overview 1-2
- profiler utility 2-86

- profiler, configuration 3-114
- profiling tool 3-111
- program execution problems 1-22
- program failure, and non-zero user count 3-68
- Program Memory, debugger command 3-21
- P-step, debugger command 3-23, 3-46

Q

- q option 3-67, 3-70, 3-71, 3-79, 3-80
- Quit command, debugger 3-20
- QUOTE literal 2-39

R

- RA, Run All Threads debugger command 3-15
- READ NEXT statement, and locked records 6-15
- READ statement, and record locking 6-15
- Realia, data storage option 2-36
- rebuilding files 3-71
- record editor (alfred) 3-107
- record locking 6-15
 - performance considerations 6-33
 - transaction management, with 5-7
- RECORD phrase 3-125
- record script command 3-11
- Record Script debugger command 3-19
- RECORD SEQUENTIAL 6-2
- record size, changing 3-96
- records
 - adding, modifying, deleting from an indexed file 3-107
 - variable-length and fixed-length 6-2
- recovery of files through transaction management 5-4
- recovery routine for file 5-5, 5-14
- recursive PERFORM statements, compiler option 2-63
- REDEFINES, and XFDs 5-24

- redefining the keyboard 4-16
- reducing the size of the runtime 2-120
- reference modification, compiler switch 2-46
- relative files 6-4
 - creating an indexed file from 3-89
- relinking the runtime 2-120
- remote file access with AcuServer 5-16
- remote file names, handling of 2-69
- remote name notation
 - alternate configuration file 2-78
 - defined 5-18
 - error output file 2-80
 - object file library 2-91
 - specifying file aliases 2-101
- remote object file name, standard options, compiler 2-4
- remote object library 3-62
- remote preprocessing 3-135
- Remove All Breakpoints, debugger command 3-47
- RENAMES clause, and XFDs 5-23
- repeat find command 3-27
- required functions, terminal manager 4-70
- REQUIRED phrase 6-55
- reserved words
 - compiler options 2-32
 - compiler options for listing 2-55
- reserved words list, where to find 2-32
- resetting
 - internal revision numbers 3-78
 - user counts 3-78
- RESIDENT 6-41
- resource files
 - adding to an object library with **cblutil** 3-60
 - defined 2-97
 - rules for 2-98
- resources, accessing with W\$BITMAP and WIN\$PLAYSOUND 2-98
- restricted attribute handling 4-61
- restricted video modes 4-62

RESTRICTED_VIDEO_MODE configuration variable 4-59, 4-62

retrieving key information with C\$KEYPROGRESS 6-27

revision numbers, resetting 3-78

REWRITE statement, and record locking 6-15

RM/COBOL

- compatibility mode

 - compiler options 2-18

 - locked records 6-15

- configuration option and ACCEPT fields 4-19

- data storage options 2-38

- file status codes, and 2-104

- reserved words option 2-33

- RM/COBOL-85 (ANSI 85) 2-106

 - Auto-Insert 4-22

 - keyboard layout 4-15

- tab stops, compiler option 2-31

- version 2 (ANSI 74)

 - CONVERT 2-48

 - file handling 2-106

rollback 5-4

ROLLBACK statement, implicit 5-6

run all threads 3-14, 3-15

- debugger command 3-24

Run menu, debugger 3-22

Run Script debugger command 3-11, 3-19

Run to Procedure debugger command 3-43

runcbl

- d option 3-5

- introduction 2-75

- running the debugger 3-2

running programs with a hot key 4-29

runtime

- CALL resolution steps 2-115

- configuration file 4-4

- configuration variables 2-100

- debugger 3-3

- introduction 2-75

- relinking 2-120
- removing optional components 2-120
- serial number 2-89
- size, reducing 2-120
- system 2-2
- terminal-related configuration variables 4-8
- timer 2-88
- runtime errors, intermediate 5-12
- runtime options
 - alternate runtime configuration file 2-78
 - basic version information 2-89
 - collecting application information with the profiler 2-86
 - collecting zero count paragraph information with the profiler 2-87
 - converting character screens to graphical 2-78
 - debugging 2-79
 - debugging mode 2-87
 - debugging with commands from a file 2-87
 - display output file 2-86
 - embedded-config-file 2-81
 - error output file 2-80, 2-85
 - file error 30 2-90
 - generating memory location output 2-92
 - ignore hang-up signals 2-83
 - information 2-89
 - inhibiting terminal initialization 2-77
 - input from file 2-83
 - keystroke file playback 2-85
 - measuring an application's real-time execution 2-88
 - memory access violation 2-92
 - object file library 2-91
 - profiler 3-111
 - redirecting display output to a file 2-86
 - safe mode 2-87
 - SPECIAL NAMES switches 2-77
 - suppression of warning messages 2-82
 - terminal output file 2-88
 - warning messages 2-89

S

safe mode exit 6-56

SCREEN

- ALPHA-UPDATES keyword 4-46
- configuration examples 4-57
- CONVERT-OUTPUT keyword 4-47
- EDITED-UPDATES keyword 4-48
- ERROR-BELL keyword 4-49
- ERROR-BOX keyword 4-50
- ERROR-COLOR keyword 4-50
- ERROR-LINE keyword 4-51
- FORM-FEED keyword 4-51
- INPUT-DISPLAY keyword 4-51
- INPUT-MODE keyword 4-52
- JUSTIFY keyword 4-52
- NUMERIC-UPDATES keyword 4-53
- PROMPT keyword 4-53
- PROMPT-ALL keyword 4-53
- PROMPT-ATTR keyword 4-54
- REFRESH-LINES keyword 4-54
- SHADOW-STYLE keyword 4-55
- SIZE keyword 4-56
- WINDOW keyword 4-56

screen functions 4-71

screen import utility 2-84

- case of text 2-55

- compiler option 2-64

SCREEN runtime configuration variable 4-46

Screen Section

- ACCEPT/DISPLAY statements 6-48
- advantages 6-47, 6-52
- comparison to field-level 6-51
- CONVERT-OUTPUT, and 4-47
- embedded procedures example 6-52
- example code 6-49
- introduction to 6-46

-
- reserved words suppression option 2-33
 - structure of 6-47
 - syntax 6-48
 - three types of screen description entries 6-48
 - screen tracing 3-49
 - screen, scrolling 4-59
 - scripts
 - recording in the debugger 3-19
 - running in the debugger 3-20
 - SCRN_ALPHA_AUTO_PROMPT configuration variable 4-47
 - SCRN_ALPHA_UPDATES configuration variable 4-47
 - SCRN_CONVERT_OUTPUT configuration variable 4-47
 - SCRN_EDITED_AUTO_PROMPT configuration variable 4-49
 - SCRN_EDITED_UPDATES configuration variable 4-49
 - SCRN_ERROR_BELL configuration variable 4-49
 - SCRN_ERROR_BOX configuration variable 4-50
 - SCRN_ERROR_COLOR configuration variable 4-51
 - SCRN_ERROR_LINE configuration variable 4-51
 - SCRN_FORM_FEED configuration variable 4-51
 - SCRN_INPUT_DISPLAY configuration variable 4-52
 - SCRN_INPUT_MODE configuration variable 4-52
 - SCRN_JUSTIFY configuration variable 4-52
 - SCRN_NUMERIC_AUTO_PROMPT configuration variable 4-53
 - SCRN_NUMERIC_UPDATES configuration variable 4-53
 - SCRN_PROMPT configuration variable 4-53
 - SCRN_PROMPT_ALL configuration variable 4-53
 - SCRN_PROMPT_ATTR configuration variable 4-54
 - SCRN_PROMPT_DEFAULT configuration variable 4-53
 - SCRN_REFRESH_LINES configuration variable 4-54
 - SCRN_REFRESH_MODE configuration variable 4-55
 - SCRN_SHADOW_STYLE configuration variable 4-56
 - SCRN_SIZE_COLS configuration variable 4-56
 - SCRN_SIZE_ROWS configuration variable 4-56
 - SCRN_WARN configuration variable 4-49
 - SCRN_WINDOW_X configuration variable 4-57
 - SCRN_WINDOW_Y configuration variable 4-57
 - scroll bar, debugger 3-2

- SCROLL configuration variable 4-58, 4-59
- scrolling in Terminal Manager 6-38
- search
 - debugger 3-26
 - debugger, Windows Help menu 3-45
- search command 3-45
- search paths
 - for code and data files 2-103
 - for COPY libraries 2-95
- SECONDARY-TABLE directive 5-44
- segmentation in source, compiler option 2-62
- SELECT statement 5-20, 5-27
- Selection
 - item in debugger menu 3-8
 - menu in debugger 3-41
- sequential files 6-2
 - automatic blank stripping 6-4
- server_MAP_FILE configuration variable 5-65
- set breakpoint command 3-35
- SET ENVIRONMENT 1-19
 - and color map 4-45
- set procedure breakpoint command 3-43
- SET statement, to set priority value of a thread 6-65
- setting variables in the debugger 3-32
- shared memory 2-121
- shared object library 2-116
- sharing, indicating programs to share 2-123
- shell command 3-11
- Shell debugger command 3-19
- SIGN clause, data storage compiler option 2-39
- SIGN-EBCDIC instruction 3-123
- sign-storage convention 2-39
- single and double precision 2-37
- single locking, file options, compiler 2-25
- size checking rules, modification 2-42
- Skip count, debugger Set command 3-39
- skip to cursor line command 3-10, 3-24

- sort files, external sort modules 6-70
 - application preparation 6-71
 - linking the module 6-71
- SORT instruction 3-124
- SORT statement 2-106
 - external sort modules 6-70
- SORT_DIR configuration variable 2-106, 2-111
- sorting records 3-124
- source file options
 - default source name extension 2-16
 - list of 2-30
 - setting tab stops 2-30
- source format 2-93
 - ANSI compiler option 2-30
 - terminal compiler option 2-32
- source-code
 - control and XFDs 5-19
 - control of 2-99, 2-100
 - debugging 3-3
 - excluding or including lines at compile time 2-99
 - format of 2-93
- spaces, treating as zero 2-65
- SPECIAL-NAMES, runtime switches 2-77
- split keys 6-12
- standard options, compiler
 - errors, write to file 2-4
 - ignore CBLFLAGS environment variable 2-5
 - object file name 2-4
 - remote object file name 2-4
 - verbose 2-5
 - warning message suppression 2-5
- START TRANSACTION, implied 2-27
- starting and using **acushare** 2-124
- Step, debugger commands 3-15, 3-23, 3-46
- Stop Recorder debugger command 3-19
- STOP THREAD 6-59
- STOP_RUN_ROLLBACK configuration variable 5-7

- subprogram calls 2-115
- subprograms 3-59
 - calling 2-115
- subscripting, compiler options for internal tables 2-14
- substring search 3-130
- subtable directive 5-45
- summary list, creating, listing options, compiler 2-11
- support, what we need if you call us 1-25
- supported hardware 1-17
- suppressing reserved words, compiler option 2-34
- suspended thread 6-66
- SWITCH_PERIOD configuration variable 6-65
- switches, SPECIAL-NAMES 2-77
- symbol tables, listing options, compiler 2-13
- symbolic debugging 2-54, 3-4
- SYNCHRONIZED
 - alternate data alignment 2-74
 - compiler option for treating binary data as 2-42
 - data items, compiler option 2-38
- synonyms for reserved words 2-33
- syntax, **logutil** 3-107
- System Menu, activating in the debugger 3-9
- System-Menu function, activation key on character-based systems 4-28

T

- tab key 4-41
- tab stops, source options, compiler 2-30
- tables, indexing, compiler option 2-63
- Teletext 925 4-62
- TEMP_DIR configuration variable 2-111
- temporary files, specifying a directory for 1-20
- termcap 4-65, 4-69
- terminal
 - characteristics, by reference 4-84
 - configuration variables 4-8

- database file 4-3, 4-6
 - by reference entry 4-84
 - editing 4-65
 - on different systems 4-5
 - systems that do not use 4-5
- definition 4-7
- format 2-93
- function codes 4-3, 4-42
- handling options 2-105
- identification 4-5
- source format, compiler option 2-32
- Terminal Manager
 - 132-column handling 6-37
 - attached devices 6-37
 - functions 4-2
 - guidelines to improve performance 6-35
 - introduction 4-2
 - line drawing 6-37
 - mouse support, and 4-82
 - performance and screen I/O 6-34
 - required functions 4-70
 - restrictions 6-36
 - scrolling 6-38
 - solutions for escape sequences restrictions 6-37
- Terminal Manager modes, standard and auto 4-14
- terminal source, compiler options 2-32
- terminals
 - initialization strings 4-83
 - list of function codes for 4-66
 - non-ANSI conforming 4-61
 - preparing for use with ACUCOBOL-GT programs 4-5
 - restrictions imposed by special 4-62
- termination key, standard and auto modes 4-14
- testing file integrity 3-69
- thread fundamentals 6-58
- thread, debugger command 3-24
- Threadds.zip sample project 6-68

threads

- and threading 6-57
 - issues, debugger 3-15
 - starting 6-57
 - suspending 6-66
- communication among 6-61
- creating 6-58
- effects on windows 6-65
- fundamentals 6-58
- interaction with run units 6-70
- locking 6-60
- logical operations preserved 6-60
- messages, messaging among 6-61
- modeless windows
 - managing 6-57
 - sample project 6-68
- multiple locks 6-61
- multiprocessor issues 6-69
- pausing 6-68
- priorities 6-64
- sharing data 6-60
- stopping 6-59
- switch point 6-65
- synchronizing 6-58
- to manage modeless windows 6-66

timestamp, file trace option 3-49

timing program execution 2-88

TMPDIR environment variable 1-20, 3-133

toggle at cursor line command 3-40

Toggle Breakpoint debugger command 3-46

toolbar, debugger 3-45

Trace Files debugger command 3-18

Trace Paragraphs debugger command 3-18

TRACE_STYLE configuration variable 3-48

transaction error codes 5-11

transaction log file

- how to edit 3-107

- logutil** 3-107
- transaction logging 5-3, 5-4, 5-6
 - configuration file variables used with 5-11
 - recovery with AcuServer, and 5-15
 - specifying log files 5-10
- transaction management 5-1
 - compiler options used with 5-13
 - features and file types supported 5-4
 - overview 5-2
 - verbs 5-6
- transactions
 - file options, compiler 2-23
 - implied, file options, compiler 2-28
- TRANSACTION-STATUS register 5-11
- trigger 5-33
- truncation of XFD names 5-42
- truncation options 2-46

U

- UniKix **Boomerang** alias 3-142
- UNIX
 - command to check modules for debug mode 3-63
 - device locking, and 6-16
- unloading to binary and line sequential format 3-87
- UNLOCK 6-15
- UNLOCK THREAD 6-61
- Up, in debugger 3-43
- upper and lower case in command-line arguments 2-68
- URLs, mapping XML files to 2-102
- USE GROUP directive 5-46
- USE instruction 3-125
- USE_LARGE_FILE_API environment variable 3-135
- user count 3-70, 3-71, 6-14
 - action if non-zero 3-68
 - resetting 3-78

- user-defined keys 4-35
- user-defined words 2-32
- using shared memory 2-122
- utilities
 - acuprof** 3-112
 - alfred** 3-107
 - AXDEFGEN 2-74
 - cblutil** 3-58
 - logutil** 3-107
 - vio** 3-98
 - vutil** 3-66

V

- V_BULK_MEMORY configuration variable 6-22
- validating data 6-55
 - when not performed 6-55
- variable-length records 6-2
 - binary sequential records 6-3
 - sequential files, and short records 6-4
- variables
 - clearing watches 3-12
 - debugger, and 3-51
 - displaying in debugger 3-30
- VAR-LENGTH directive 5-47
- VAX COBOL 2-15
 - CONVERT 2-49
 - data storage option 2-34, 2-36
 - file names 2-110
 - file status codes, and 2-104
 - line sequential files 6-3
 - reserved words option 2-33
 - sequential files 6-3
 - terminal source format 2-94
- VAX/VMS
 - file names 2-110

- line sequential files 6-3
- sequential files 6-3
- V-BUFFERS, performance considerations 6-33
- verbose, standard options, compiler 2-5
- version command 2-72
- video
 - background color, compiler options 2-50
 - BLANK EOL, compiler option 2-48
 - control with SCREEN variable 4-46
- video attributes 4-73
 - terminals with non-hidden 4-62
- video compiler options
 - exception keys, -Ve 2-49
 - exception keys, -Vx 2-50, 2-51
 - high intensity 2-49
 - inhibiting bell 2-50
 - intensity 2-48
 - low intensity 2-50
 - ON EXCEPTION phrase 2-49
 - treating as if CONVERT specified 2-48
- view breakpoints command 3-21, 3-40
- view monitors command 3-21
- view perform stack command 3-20
- view procedure command 3-43
- view screen command 3-20
- vio**
 - examples 3-105
 - file transfer utility 3-98
 - known limitations 3-106
 - options 3-100
 - Windows considerations 3-104
- Vision file system
 - described 6-5
 - Versions 5 and 4, and 3 6-6
- Vision file utility 3-66
- Vision files
 - block size 6-13

- comment field, setting 3-97
- recovering deleted records (Vision 5) 3-80
- segment naming 6-7

VMS file utilities 3-66

VMS operating systems 2-107

vutil

- changing record size 3-96
- checking for file integrity 3-69
- collating sequence 3-83
- converting a C-ISAM file 3-92
- converting a Micro Focus file 3-94
- creating empty files 3-81
 - indexed format 3-84
 - interactive version 3-82
 - non-interactive version 3-84
 - sequential and relative 3-87
- examining file information 3-67
- extracting records 3-79
- loading a file 3-89
 - with large records 3-91
- resetting revision numbers 3-78
- resetting user counts 3-78
- setting the comment field 3-97
- specifying a compression factor 3-74
- unloading to binary and sequential format 3-87
- utilities 3-66

vutil options

- augment option 3-96
- note option 3-97
- tree option 3-97
- version option 3-98

vutil options, listed 3-70, 3-72

vutil, rebuilding files 3-71

- automatic placement 3-72
- blocking factor 3-73
- default method fails 3-75
- directory specification 3-73

- extension factor, setting a new one 3-74
- from an interrupted rebuild 3-77
- in key order 3-75
- latest Vision format 3-77
- limited disk space and 3-74, 3-76
- locking options 3-72
- naming temporary files 3-72
- record compression 3-73
- slow rebuild 3-77
- specifying size of spool media 3-77
- spooling 3-76

Vx compile-time option 4-14

W

W\$BITMAP routine, accessing resources 2-98

WA variable 3-15

WAIT statement 6-58

warning messages

- compiler option 2-5
- runtime option 2-82

WARNINGS configuration variable 2-89

watch 3-15

- clearing all 3-12
- watching a variable 3-15

watch size command 3-15, 3-27

watch variable vs. monitor variable 3-33

watch variables in debugger, watch option 3-43

Watch Window 3-32

WHEN directive 5-48

wide format listing options, compiler 2-13

WIN\$PLAYSOUND routine 2-98

Window Memory, debugger command 3-21

window size 3-5

- command window 3-15, 3-27

window text-mode, reconstructing 4-81

Windows

- extra keys 4-35
- keys that cannot be re-defined 4-37
- line sequential files 6-3
- running the debugger under 3-5

Windows console runtime 1-6

Windows DLL 2-116

WRAP configuration variable 4-58, 4-59

write to error file, standard options, compiler 2-4

WRITE, bulk addition 6-21

writing a preprocessor 2-136

Wyse 50 4-62

X

-x option 3-67, 3-70

X/Open COBOL Standard 2-35

XENIX and setcolor 4-45

XFD 5-20

- file format 5-54

- file options, compiler 2-23

- name 5-27

- XML stylesheet directive 5-54

XFD data dictionaries, file options, compiler 2-24, 2-28

XFD files 5-19

- defaults used 5-23

- XML format 2-25

XFD_DIRECTORY configuration variable 5-65

XML documents, XFD files 2-25

XSL directive 5-54

Y

year format, -Zy option 2-65

