Runtime Manual

ACUCOBOL-GT®

Version 8.1.3

Micro Focus
9920 Pacific Heights Blvd.
San Diego, CA 92121
858.795.1900

E-01-UG-100501-Runtime Manual-8.1.3

Contents

1 Introduction

Key Topics

1.1 Overview

This *Runtime Manual* provides a summary of the options, variables, error codes, and utilities used with the ACUCOBOL-GT® runtime system, also known as the COBOL Virtual Machine™. The runtime system is part of the ***extend***® family of solutions.

After a program is successfully compiled with the ACUCOBOL-GT compler, it is ready for immediate execution with the ACUCOBOL-GT runtime. There is no link step. Programs called during execution are loaded dynamically. See **Chapter 4, "Chapter 4: Runtime Options,"** for details on starting the runtime. For more information on the use of both the compiler and the runtime, see Chapter 2, "Compiler and Runtime," in the *ACUCOBOL-GT User's Guide*.

1.1.1 Available Runtime Systems

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". In this manual, the runtime executable is often referred to as **runcbl**.

1.1.1.1 Windows runtime systems

Several types of runtimes are available on Windows systems, each of which supports a distinct type of deployment. Each runtime is licensed separately.

- The standard Windows runtime is encapsulated in a dynamic link library (DLL), and is started via the program named "wrun32.exe". This runtime is used for all standard Windows deployments.

- Thin client deployments use a special runtime named "acuthin.exe". For more information about thin client technology, see Chapter 1, section 1.3.2, "Thin Client," in the *AcuConnect*® *User's Guide*.

- The Windows *Console* runtime, named “crun32.exe”, supports applications originally developed for Extended DOS or other character-based systems. The console runtime uses the Windows Console API and runs in a virtual DOS window. For more information, see section 1.2.3.1 in Book 1, *ACUCOBOL-GT User’s Guide*.

- The Alternate Terminal Manager (ATM) runtime, named “run32.exe”, allows users 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 of *ACUCOBOL-GT User’s Guide*.

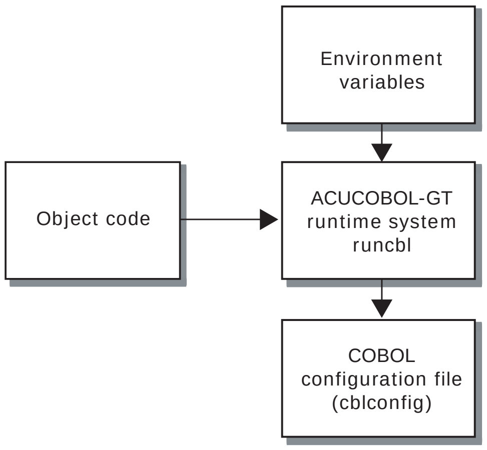
- To support Windows-based deployment of Web applications, there is a special Web runtime and CGI runtime. For more information on these options, see *A Programmer’s Guide to the Internet*.

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.).

1.1.2 Runtime Configuration File

Users of the ACUCOBOL-GT runtime can modify many aspects of the runtime environment on a site-by-site or user-by-user basis without recompiling. This is accomplished through a text file known as the *runtime configuration file*.

In addition, the runtime can be configured with operating system environment variables.


Environment variables
Object code
ACUCOBOL-GT runtime system runcbl
COBOL configuration file (cblconfig)

For example, the location of data files, names of devices, text of error messages, file buffering, and screen editing functionality can all be maintained outside of the compiled programs. For complete information, see **Chapter 3, "Chapter 3: Runtime Configuration File."**

All runtimes include a built-in source-level debugger. The end user may cooperate with the application developer to use some of the debugging capabilities. For more information, see **Chapter 5, "Chapter 5: Runtime Debugger."**

By default, the ACUCOBOL-GT runtime uses the Vision indexed file system to manage its indexed data files. On VMS/OpenVMS systems, the RMS file system is used. Several utilities are available to work with Vision files and transaction management log files. These utilities include **vutil**, **vio**, **logutil**, and **alfred**. These utilities are described in Chapters 7 through 10.

On most UNIX and Linux systems, the **acushare** utility program supports three key deployment services:

- ACUCOBOL-GT runtime license management
- ACUCOBOL-GT runtime shared memory management
- AcuServer® license management (for deployments using AcuServer)

For more information, see **Chapter 8, “Chapter 8: Shared Memory.”**

2 Setting Up Your Terminals

Key Topics

2.1 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 (GUI) components, such as floating windows (modal *and* modeless) and controls on text-mode systems, allowing you to customize the characters used to emulate graphical components. For more information, see **section 2.6.7, “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.

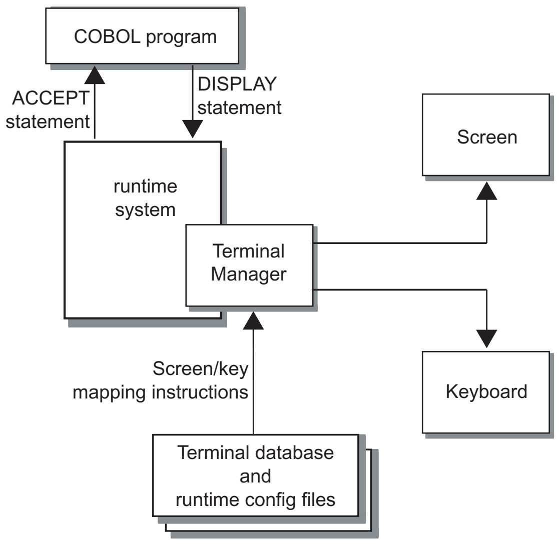
For example, in this chapter you’ll see how to specify what terminal you have, and how to make choices like:

- 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.

2.1.1 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 that 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. These are often used in conjunction with the key codes and terminal function codes mentioned above.

2.1.2 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.

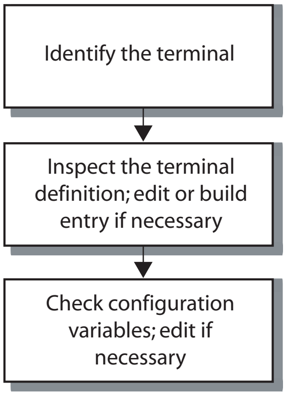
All major runtime functionality, except graphical support, is available with the ATM runtime. Because it is for character-based programs, certain Windows-specific features, such as pop-up dialog boxes, are not supported. Note that the ATM runtime automatically detects and uses Acu4GL DLLs, if present, and it supports calls to other DLLs.

Note: Should you need to relink the ATM runtime, see the instructions in section 4.3.6 of *A Guide to Interoperating with ACUCOBOL-GT.*

2.2 Getting Your Terminals Ready

This section 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:


Identify the terminal
Inspect the terminal definition; edit or build entry if necessary
Check configuration variables; edit if necessary

2.2.1 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 ***extend*** 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_TERMCAP; if that variable is present, the runtime opens the file named in it as the terminal database file. If the system variable A_TERMCAP 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	SYS$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_TERMCAP (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_TERMCAP.

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. For information on exceptions, see **section 2.2.2, "Step Two—Terminal Definition."** 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.

2.2.2 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. For more information, see **section 2.6, “The Terminal Database File.”**
2. Set A_TERM to the name of that entry.

2.2.2.1 Windows special considerations

Neither the ACUCOBOL-GT graphical runtime nor the console runtime for Windows uses 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 ***extend*** 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.

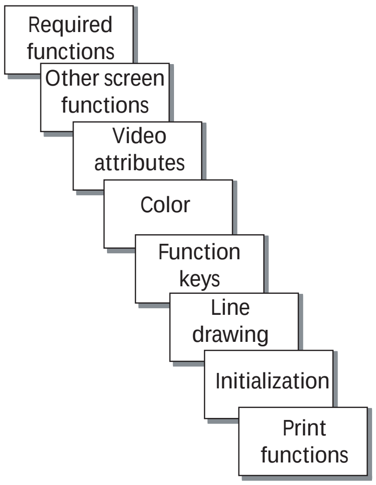
2.2.3 Step Three—Configuration Variables

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	\etc\cblconfi
UNIX and Linux	/etc/cblconfig
MPE/iX	/etc/cblconfig
VMS	SYS$LIBRARY:A_CONFIG.DAT

Options for naming and accessing the runtime configuration file, and descriptions of many of the entries in it, are discussed in **Chapter 4, "Chapter 4: Runtime Options."** 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:



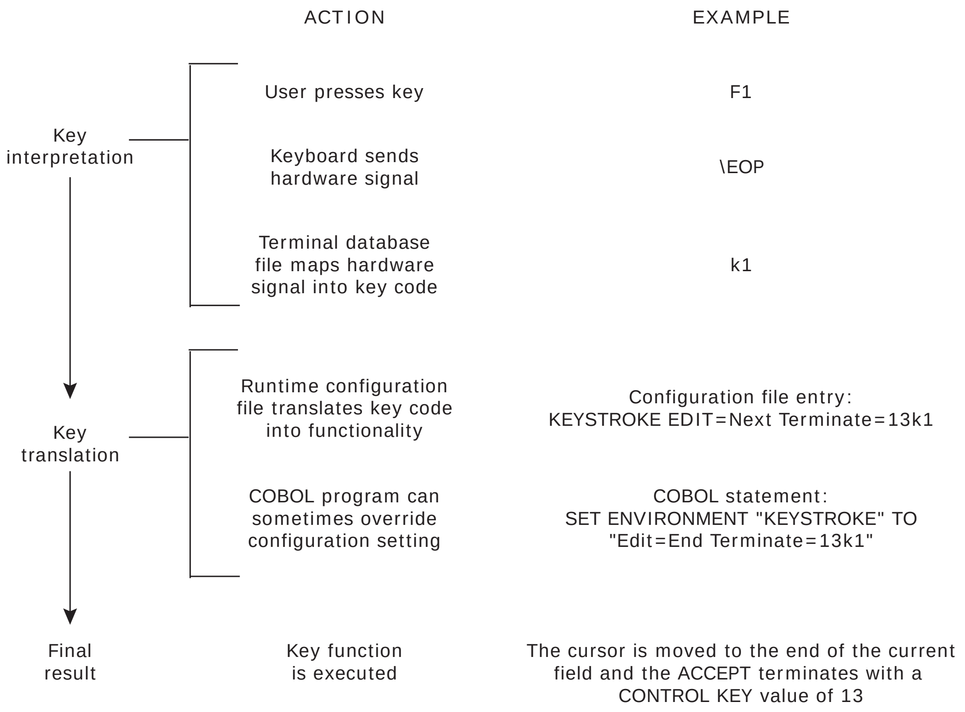
2.3 The Keyboard Interface

The Terminal Manager handles both the screen display and keyboard input. For more information, see **section 2.4, “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.

2.3.1 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 following diagram depicts the overall process, from keystroke to COBOL program.


ACTION
EXAMPLE
Key interpretation
User presses key
F1
Keyboard sends hardware signal
\EOP
Terminal database file maps hardware signal into key code
k1
Key translation
Runtime configuration file translates key code into functionality
Configuration file entry: KEYSTROKE EDIT=Next Terminate=13k1
COBOL program can sometimes override configuration setting
COBOL statement: SET ENVIRONMENT "KEYSTROKE" TO "Edit=End Terminate=13k1"
Final result
Key function is executed
The cursor is moved to the end of the current field and the ACCEPT terminates with a CONTROL KEY value of 13

The following sections describe in detail the steps shown above.

2.3.1.1 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 **section 2.3.2.3, "Table of keys."**

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 2.6, "The Terminal Database File."**

2.3.1.2 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.

2.3.1.3 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 the following 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, 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, this key terminates the field and returns that number as its termination key value. If the number is starred (*), this key also causes an exception condition. If there are both a number and an action, 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 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 2.3.2, "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 2.3.2, "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.

2.3.2 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.

2.3.2.1 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. For more information, 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 get 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”, 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), 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 in which 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 that 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 (for example, 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", 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.

2.3.2.2 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 2.3.2.3, "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 The character-based version of the runtime supports the use of key letters in controls. The user can activate a particular control by pressing a predesignated key in combination with the control's key letter.

Use the Alt function to designate the key to be pressed in combination with the control's key letter. On Windows systems, the user presses the "Alt" key in conjunction with the key letter.

Auto-Insert	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.
Backspace	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.
Default-Entry	The Default-Entry action erases the remainder of the field starting at the cursor position, *provided that* the cursor is not in the first position of the field. If the cursor *is* 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, this key is typed (which does nothing to the field). If the default is wrong, the correct value is entered and this key is typed (erasing the part of the old field after the new input).
Default-Next	Default-Next combines the Default-Entry action and the Next (described below) action.
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	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. **Note:** Under Windows, the field in which the cursor is currently positioned is erased, instead of all fields controlled by the ACCEPT statement being erased.
Erase-EOS	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. **Note:** 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.
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, it 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	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. **Note:** 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.
Left	The cursor is moved one position to the left; if it is already in the leftmost field position, it moves to the end of the previous field. If the cursor is in the leftmost 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. 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.
Menu	The key is defined as a Menu key. Pressing this key will cause a program-defined menu to appear on the screen.
Next	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.

Next-All 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.

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.

Next-Line Next-Line functions the same as the Down action, except that the cursor always moves to the beginning of the leftmost field on the new line (instead of maintaining the current cursor column).

Numeric-Default If the field is numeric, 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 *not* 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.

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.

Numeric-Next If the field is numeric, 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 *not* 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.

If the field is alphanumeric, 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.

Page-Down	This keyword sets the key that pages down a multiline entry field, list box, and combo box.
Page-Left	This keyword sets the key that scrolls left one page.
Page-Right	This keyword sets the key that scrolls right one page.
Page-Up	This keyword sets the key that pages up a multiline entry field, list box, and combo box.
Previous	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.
Previous-All	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. 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. **Note:** 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.
Previous-Line	The cursor moves to the beginning of the leftmost 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.

Right	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). 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. 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, *User Interface Programming*, section 11.4, “Regarding Configuration Variables.”)
Scroll-Left	This keyword sets the key that scrolls left one column.
Scroll-Right	This keyword sets the key that scrolls right one column.
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 *system* 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 2.3.2.3, "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=F11 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, 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, 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”, 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.

2.3.2.3 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, PrtSrcn, 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
Host’s Line-Kill	ZK	-
Cntrl-A - Cntrl-Z	^A - ^Z	Ctl A-Z
Escape	^[	Esc
Control-\	^\	Ctl-\
Control-]	^]	Ctl-]
Control-^	^^	Ctl-^

Key	**Key Code (terminal db file)**	**Windows Keyboard**
Control-_	^_	Ctl-_
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	-
Attention	KA	-
Bottom	KB	Ctl-PgDn
Clear	KC	Ctl-Home
Command (Do)	KD	-
End	KE	End
Find	KF	-

Key	**Key Code (terminal db file)**	**Windows Keyboard**
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
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

Action	**Key Code**	**Exception Value**
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 "kl" definition changed or removed.

2.3.2.4 Additional Windows keys

These extra keys are available for 32-bit Windows systems in addition to those listed in **section 2.3.2.3, "Table of keys"**:

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
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

Key	**Key code**
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
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 in Book 4 for more details.

2.3.2.5 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.

- 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.

- 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.

- 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 kl”
```

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:

- 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.

- 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.

- 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.

2.3.2.6 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

2.3.2.7 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.

2.4 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 2.5, "Restricted Attribute Handling,"** describe the runtime configuration file options. The terminal database file function codes and values are described in **section 2.6, "The Terminal Database File."**

2.4.1 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 file entry 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 “-Vl” 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 eight 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.

2.4.2 The SCREEN Option

There is a runtime configuration variable called "SCREEN" that controls many features of the video subsystem. 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*

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”, 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 effects 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 to allow the user to enter values between zero and 1 when the default value is zero. If this rule did not exist, 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 nonzero 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), 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
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, 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 SCRN_INPUT_MODE configuration variable.

JUSTIFY=*value*

The JUSTIFY setting determines the default justification of converted numeric and numeric-edited fields. If “Left” is chosen, leading spaces are removed from these fields when they are displayed. If “Right” is chosen, the leading spaces are retained. Finally, if “Auto” is chosen (the default), 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. For the Auto-prompt value, use 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 (for example, "Prompt= "). You may also specify the configuration variable SCRN_PROMPT to set this value. The SCRN_PROMPT_DEFAULT variable 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", 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:

```
SCREEN_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 2.4.4, "Double-Byte Character Handling"**) is nonzero, 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 recognizes several special values. If *either* the row or column is set to a negative number, the initial window is minimized (turned into an icon). If either value is set to “999” or larger, the initial window is maximized instead. Finally, if either value is zero, 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.

2.4.2.1 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
```

2.4.3 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 nonzero 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 nonzero 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 nonzero 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 2.5, "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.

2.4.4 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 therefore may 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.

2.5 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.

2.5.1 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 nonzero 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, 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).

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.

2.5.1.1 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 nonzero 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, 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) 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, you should make sure that you fully test your programs on one of them.

2.6 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:

\E	an escape character
\n	a new line (control-J)
\r	a carriage return (control-M)
\t	a tab (control-I)
\b	a backspace (control-H)
\f	a form-feed (control-L)
^X	X is any character, treated as control-X
\nnn	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 ìsgî 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.

2.6.1 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, 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 then x, 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:
```

2.6.2 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, 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:
```

2.6.3 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, you have a “magic cookie” style of terminal. If nothing happens, type some characters. These should show up in reverse video. If they do, you have an ANSI style terminal that allows for independent attributes for each screen position. If you do not get reverse-video at all, you did something wrong.

If you include RV, UL, BL, RU, or RB in your terminal database file entry, 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, 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, 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, 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, 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.

2.6.4 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
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.

2.6.4.1 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:).

2.6.5 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.

2.6.5.1 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 “ESCO” 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:
```

2.6.6 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.

2.6.6.1 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:kl=\E[D:\
    :GO=\E(0:GF=\E(B:GM=qxlkmjvtwun:
```

2.6.7 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 2.6.6, "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	([)
14.	Right entry field box and check box character	(])

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.

2.6.8 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 and this assumption may 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:cl=\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:\
:al=\E[L:dl=\E[M:ic=\E[@:dc=\E[P:\
:kh=\EO\000:kN=\E[6~:kP=\E[5~:\
:km=\E[M:\
:w8=\E[?3l: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=qxlkmjvtwun:\
:W8=\E[?3l:W3=\E[?3h:\
:hs:ts=\E[?E\E[?%i%dT:fs=\E[?F:es:ds=\E[?E:
```

2.6.9 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:\
    :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:\
    :GO=\E(0:GF=\E(B:GM=qxlkmjvtwun:\
    :is=\E>:
```

2.6.10 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:
```

2.6.11 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:
```

3 Runtime Configuration File

Key Topics

3.1 Introduction

Many aspects of the runtime system can be controlled through *runtime configuration variables*. This mechanism provides a great deal of flexibility, because these variables can be modified by each **runcbl** site as well as directly by an ACUCOBOL-GT program.

3.1.1 Variable Syntax

Configuration variables are maintained in a *runtime configuration file*. This standard text file can be modified by the host system's text editor. Each entry in the runtime configuration file consists of a single line. All entries start with a keyword, followed by one or more spaces or tabs, and then one or more values. The limit for each configuration value entry is 4095 characters.

Some examples of runtime configuration variables are:

```
AUTO_PROMPT   0
BELL   1
COMPRESS_FACTOR   70
CURSOR_TYPE   3
MENU_ITEM   Edit=Delete   200
SCROLL   on
```

For all runtime configuration variables, "=" placed between the keyword and the first value is optional, and is interchangeable with a space.

For the following configuration variables, a colon (:) may be used instead of an equals sign (=) in the value portion of the entry:

COLOR-TABLE	COLOR-MAP
FILE-CONDITION	KEYBOARD
KEYSTROKE	SCREEN
MENU-ITEM	MOUSE
HOT-KEY	

In the above cases, allowing a colon instead of an equals sign in the value portion of the entry makes it possible to specify these values in environment variables. This accommodates systems that do not allow an equals sign in the environment variable.

For some runtime configuration variables, the words “on”, “true”, and “yes” are synonyms for “1”, and the words “off”, “false”, and “no” are synonyms for “0”. The entry for each variable in this appendix indicates when these synonyms are allowed.

In the keyword, all lower-case characters are treated as upper-case and all hyphens are treated as underscores. Keywords longer than 60 characters are truncated to 60 characters.

Numeric Values within Variables

For some runtime configuration variables, a numeric value may be specified. By default, this numeric value is interpreted as a decimal number. If desired, you can specify the number using as a hexadecimal or base 16 number. To do so, prepend the characters "0x" or "0X" to the number or append "h" or "H".

The number itself is composed of the characters "0" through "9" and "A" through "F" and "a" through "f".

3.1.2 Variable Usage

The configuration file is optional, as are all of its contents. For this reason, no errors in the configuration file are ever reported. The “-l” **runcbl** option can help debug configuration file problems.

In the descriptions of some runtime configuration variables, you will find comments about behavior under the Windows environment; unless otherwise noted, these comments apply to all 32-bit versions of the Windows operating system.

Runtime configuration variables may be placed in either the runtime configuration file or the machine’s environment. When they are placed in the runtime configuration file, upper- and lower-case names are equivalent, as are hyphens and underscores. When placed in the machine’s environment, the keywords must be all upper case and must use underscores instead of hyphens. For more details about the configuration process, see the *ACUCOBOL-GT User's Guide*, section 2.8, "Runtime Configuration."

All configuration variables that have a default value are used by and affect the runtime in the same way that they would if they were in the configuration file. That is to say, a configuration variable that has a default value is treated as if it appears in the configuration file set to the default value.

The values of many runtime configuration variables may be changed at runtime with the SET ENVIRONMENT verb. The syntax is:

```
SET ENVIRONMENT env-name TO env-value
```

Env-name may specify either the literal name of the variable or a data-item whose value is the name of the variable. If you specify the actual name of the variable, such as CODE_CASE, then you must enclose the name in quotes. *Env-value* is the value to which *env-name* will be set. If it is a numeric data item, then it is treated as if it were redefined as an alphanumeric data item.

Most configuration variables can be read with the ACCEPT FROM ENVIRONMENT statement. If the variable to be read is numeric, then the receiving field must be defined either as a numeric field or as an alphanumeric field of five or more characters. If it is defined as alphanumeric and is longer than five characters, then the value that is read from the environment will occupy the leftmost five characters of the field and the remainder will be space-filled.

3.1.3 Configuration filename Resolution

runcbl uses the following rules to decide what the configuration file is called:

1. If the "-c" runtime option is used, the configuration file is the one named by that option; otherwise,
2. If the operating system environment variable "A_CONFIG" is defined, its value is the name of the configuration file; otherwise,

3. The configuration file is named according to the host operating system. This depends on the operating system used by the machine, as outlined in the following table.

System	**Configuration File**
Windows	\etc\cblconfi
UNIX/Linux	/etc/cblconfig
MPE/iX	/etc/cblconfig
VMS	SYS$LIBRARY:A_CONFIG.DAT

Caution: Do not give a data file a name that is the same as a configuration variable name. Doing so can cause problems if you map the data filename through a configuration entry. For example, if you have a data file named “CURRENCY”, the runtime may confuse the data file with the configuration variable of the same name, inadvertently changing the default currency character.

3.1.4 Nested configuration files

It is possible to use multiple configuration files by nesting one inside another. Within the configuration file, you can specify another file to process with the following syntax:

```
    !COPY filename
```

No name expansion is done to *filename* (for example FILE_PREFIX is not applied) so you must specify a file that the runtime can find. You can include remote name syntax if you are using AcuServer® or AcuConnect®. Otherwise, the file must be an absolute path or a path relative to the current directory.

For example, if you have some configuration variables in a global place such as “/etc/cblconfi”, then individual users can execute the runtime using this configuration file instead of the usual one. The settings in the usual configuration file take effect also, because their settings are copied in with !COPY:

```
#Get all the standard variables
!copy /etc/cblconfi

#Now set personal settings
COMPRESS_FILES 1
```

3.2 Configuration File Variables

This section contains an alphabetical list of the runtime configuration file variables. Many of these variables are also described in other parts of the manual.

3D_LINES

This variable has meaning only on graphical systems such as Windows. Set this variable to "1" (on, true, yes) to cause the runtime to display lines and boxes with 3-D shading. This makes the lines appear to be inscribed into the surface of the screen. The variable is especially helpful in giving a 3-D look to a program originally designed on a character system. Only black lines on a non-black background are shown with shading. Other lines are displayed normally.

The set of colors available to ACUCOBOL-GT significantly impacts how effective the shading will be. Normally, the shading is most effective when the background is low-intensity white. The other low-intensity colors are next best.

The shading is only marginally effective with a high-intensity background. For this reason, the 3D_LINES setting is not used when a high-intensity background is drawn. Note that, by default, ACUCOBOL-GT shows background colors in high-intensity, so you will need to use at least one other configuration variable to arrange for a low-intensity background color. For example, the BACKGROUND_INTENSITY variable could be set to "1" to force a low-intensity background.

You may freely change the way lines are displayed in COBOL by using the SET ENVIRONMENT verb to set 3D-LINES prior to displaying a line or a box.

- Setting it to “1” (on, true, yes) gives you the 3-D effect.
- Setting it to “0” (off, false, no) gives you normal lines.

The runtime remembers which lines are drawn with 3-D, so you don’t need to keep track of this yourself. Note, however, that if you attach a 3-D line to a non-3-D line, the intersection will use the 3D-LINES setting currently in effect.

The default value is “0”.

4GL_COLUMN_CASE

When set to “unchanged”, this variable causes the runtime to leave the case and hyphen usage of the field names found in XFDs unchanged. XFDs are used with the Acu4GL interface, AcuXDBC, or AcuXML. They are also required for international character mapping with AcuServer and they provide useful information to the **alfred** record editor. By default, the runtime converts all field names to lower case and all hyphens are converted to underscores.

For AcuXML, the case and hyphen usage of the XFD must match the XML file exactly, and 4GL_COLUMN_CASE should be set to “unchanged”. For Acu4GL, however, you should be aware that most databases do not accept hyphens in column names. If you set this variable to “unchanged” to protect case, you may need to modify the XFD by hand to replace hyphens with underscores.

7_BIT

When this configuration variable is set to “1” (on, true, yes), ACUCOBOL-GT supports 7-bit communications instead of 8-bit. This variable is designed specifically for machines that use 7-bit communications with parity enabled. When 7_BIT is set to the default of “0” (off, false, no), 8-bit communications are used.

A_CHECKDIV

This variable allows you to specify an alternate runtime response to a divide by zero condition when the statement does not include a SIZE ERROR clause.

In COBOL, a division by zero produces a size error condition. The SIZE ERROR clause allows the programmer to specify actions to take when this condition occurs. If there is no SIZE ERROR clause, by default in ACUCOBOL-GT the results are undefined. You can use the A_CHECKDIV configuration variable to specify alternate handling.

A_CHECKDIV can be set to:

NONE	or:	“0”	The default setting. This setting retains the default behavior of the runtime: the results are undefined.
ABEND	or:	“1”, STOP, ABORT	This setting causes the runtime to catch the divide by zero condition and exit with the error message: “Attempt to divide by zero”.
MOVE_ZERO	or:	“2”, ZERO_RESULT , MOVE_ZEROS	This setting causes the runtime to move zeroes to the destination item(s) and continue.

A_DEBUG

This variable is available for applications such as online transaction servers that call ACUCOBOL-GT through the C API (see Chapter 6 of *A Guide to Interoperating with ACUCOBOL-GT*). The default value is “0”. With the default setting, the debugger launches when the debug_method flag in the C interface is set to “1”.

Set this variable to “1” to turn on the ACUCOBOL-GT debugger in an xterm window the first time you call the C interface. The debugger shuts down when the program that caused it to launch shuts down.

A_DISPLAY

This variable is available for applications such as online transaction servers that call COBOL through the C API. The value of A_DISPLAY overrides the value of the DISPLAY environment variable. Set A_DISPLAY to the value of your X server host name or IP address in the runtime configuration file (or /etc/cblconfig). For example:

```
A_DISPLAY myvpn123.myhostname.com:0
```

A_EXTFH_FUNC

The value of this variable is an EXTFH function name needed for the EXTFH interface. If you are using a library that contains an EXTFH function name other than “cics_xfh”, “cobol_extfh”, or “EXTFH”, you also need to set one or more of these variables to specify the function name:

A_EXTFH_FUNC	Specifies a function to be used by all file types (indexed, relative, and sequential).
A_EXTFH_IDX_FUNC	Specifies a function name to be used by indexed file types.
A_EXTFH_REL_FUNC	Specifies a function name to be used by relative file types.
A_EXTFH_SEQ_FUNC	Specifies a function name to be used by sequential file types.

For example, to specify a function name to use for all file types:

```
A_EXTFH_FUNC=myExtfh
```

Or, to specify a different function for indexed, relative, and sequential files:

```
A_EXTFH_IDX_FUNC=myIdxExtfh
A_EXTFH_SEQ_FUNC=mySeqExtfh
A_EXTFH_REL_FUNC=myRelExtfh
```

If the library is a DLL, you can specify both the name of the DLL and the calling convention to use. Any calling convention specified this way overrides the DLL_CONVENTION variable setting. For information about specifying DLLs and calling conventions, see section 3.3.2, "Loading DLLs with Configuration Variables," in *A Guide to Interoperating with ACUCOBOL-GT.*

A_EXTFH_LIB

The value of this variable is an EXTFH shared library or DLL file name. You can use this variable to dynamically load an EXTFH library without relinking the ACUCOBOL-GT runtime. For example:

```
A_EXTFH_LIB libraryname.so
```

You can also use the following variables to specify library names for indexed, relative, and sequential files. The ACUCOBOL-GT runtime uses A_EXTFH_LIB as the default EXTFH library for all three file types. If one or more of these three variables is also set, the runtime uses its value instead of A_EXTFH_LIB for the corresponding file type.

A_EXTFH_IDX_LIB	Specifies the EXTFH library to use for indexed files.
A_EXTFH_REL_LIB	Specifies the EXTFH library to use for relative files.
A_EXTFH_SEQ_LIB	Specifies the EXTFH library to use for sequential files.

You can specify these variables in the runtime configuration file or as operating system environment variables.

If the library is a DLL, you can specify both the name of the DLL and the calling convention to use. Any calling convention specified this way overrides the DLL_CONVENTION variable setting. For information about specifying DLLs and calling conventions, see section 3.3.2, "Loading DLLs with Configuration Variables," in *A Guide to Interoperating with ACUCOBOL-GT.*

See section 11.6, "Working With an EXTFH Interface," in *A Guide to Interoperating with ACUCOBOL-GT*, for information on specifying EXTFH library and function names to use with the EXTFH interface.

A_EXTFH_SIMPLE_OPEN_OUTPUT

This variable is only used in UniKix environments, and the UniKix application automatically sets this variable to “1” (TRUE). When set to “1” (TRUE), an OPEN OUTPUT statement will cause the EXTFH functions to bypass the “make” process, and will open the file as OUTPUT. When set to “0” (FALSE), or not set at all, the EXTFH functions will execute the “make” process, and will open the file as EXTEND.

A_EXTFH_VARIABLE_IDX, A_EXTFH_VARIABLE_REL, A_EXTFH_VARIABLE_SEQ

These variables indicate whether the filesystem you are accessing with the the EXTFH interface can or cannot handle variable length files. Setting this variable to the default of “1” (on, true, yes) causes the EXTFH interface to pass the minimum and maximum record lengths to the file system for variable length files as defined in the COBOL program. Setting this variable to “0” (off, false, no) causes the EXTFH interface to ignore the variable record length defined in the COBOL program, instead passing a record length equal to the maximum record length.

You can specify the variable separately for indexed, relative, and sequential files. For example:

```
A_EXTFH_VARIABLE_IDX=0
A_EXTFH_VARIABLE_REL=0
A_EXTFH_VARIABLE_SEQ=1
```

When the file system does not process variable length files, set these configuration variables to “0” and the EXTFH interface treats variable length records as fixed lengths.

If the file system does process variable length files, set the configuration variables to “1” (or do not set them at all).

A_JAVA_CHARSET

This variable specifies the character set that the runtime should use when mapping Java strings or PIC X data items containing characters outside of the ISO-8859-1 range. The default setting is "IS0-8859-1". If you have data outside the IS0-8859-1 range (e.g.an umlaut or Euro symbol) specify a different character set that contains those characters.

Be aware of a common misconception that ISO-8859-1 is equivalent to Windows-1252. This is mostly true, but there are characters in the range 0x80 - 0x9F that differ. Windows-1252 uses these numbers for letters and punctuation while the ISO-8859-1 uses these for control codes.

A_JAVA_GC_COUNT

A_JAVA_GC_COUNT is a 32-bit value that determines how often the runtime calls the JVM garbage collector. The JVM garbage collector will run at unknown times, in order to deallocate memory which is no longer being used. Setting this to a non-zero value allows you to be a little more intentional about running the garbage collector. The value is the number of times C$JAVA is called before the runtime calls the JVM garbage collector. The default value is 9883, so every 9883 calls to C$JAVA will explicitly call the JVM garbage collector. (For more info on the JVM garbage collector, see your JVM documentation.)

A_JAVA_TRACE_FILENAME

A_JAVA_TRACE_FILENAME is the name of the file where the trace information is sent. This filename can include all of the format specifiers that the runtime error file can include. If this file can’t be opened for writing (for any reason), no trace information is collected.

A_JAVA_TRACE_VALUE

To track calls to the JVM made on behalf of the COBOL program, you can set one of the following three configuration variables: A_JAVA_TRACE_VALUE, A_JAVA_TRACE_FILENAME, and A_JAVA_GC_COUNT.

A_JAVA_TRACE_VALUE is a 32-bit value that determines the types of calls to trace. Add any of the following values together to create a single value to set.

1 - Show calls that return simple types (boolean, byte, character, short, integer, long, float, double).

2 - Show method calls that return simple types.

4 - Show string calls that return string references (that must be released).

8 - Show string calls that return simple types.

16 - Show calls that return references to a Java object (that must be released).

32 - Show method calls that return references to a Java object (that must be released).

64 - Show calls that return references to a Java array or array elements (that must be released).

128 - Show calls that return other array information.

256 - Show calls to the exception routines (some of which must be released).

512 - Show calls to get IDs (Method Identifiers or Field Identifiers).

1024 - Show calls to field functions that return references to a Java object (that must be released).

2048 - Show calls to field functions that return simple types.

4096 - Show other types of calls that return references to a Java object (that must be released).

8192 - Show calls to release a reference to a Java object.

16384 - Show other calls to the Java runtime.

Note for there to be no memory leaks, any call that returns a reference to a Java object (that must be released) needs to be paired with a call to release that reference. If the COBOL program gets that reference, it is responsible for releasing the reference. If the runtime gets the reference for internal purposes, the runtime is responsible for releasing the reference.

For example, setting A_JAVA_TRACE_VALUE to 13684 shows all calls to the JVM that obtain or release a reference to a Java object. Setting A_JAVA_TRACE_VALUE to -1 is equivalent to setting it to 32767 (which is the sum of all the above values), and has the added benefit of tracing new options that may be added in the future. However, for finding memory leaks, this may be too much information.

A_LICENSE_RETRIES

This variable affects UNIX networks with multiple-user licenses for the runtime. When set to a positive, non-zero value, this entry causes the runtime to retry ("value" times) any failed attempt to register with the network license manager, acushare. The configuration variable A_RETRY_DELAY specifies how many seconds the runtime will wait between retries.

The default value is "0" (no retries).

A_OPERATING_SYSTEM

As of Version 5.0, the runtime no longer differentiates between "UNIX-V" and "UNIX-4" in the OPERATING-SYSTEM field of the SYSTEM-INFORMATION data item. Instead, the value "UNIX" is used for all UNIX platforms. If you have an existing program that depends on one of the older values, set A_OPERATING_SYSTEM to a value of "UNIX-V" or "UNIX-4". Then, when an ACCEPT FROM SYSTEM-INFO statement is executed, this value overrides the value returned by the function. The default value is empty.

A_REMOVE_EMPTY_ERROR_FILE

Use this variable to prevent the accumulation of 0 byte files when using format specifiers such as “%p**”** (to include the process id) in the error file name. When this variable is set to “1” (on, true, yes), the runtime deletes its error file if the runtime has never written to that file. Note that on some operating systems, if your error file is shared by multiple processes (i.e., the file name does not include the process id or some other unique session information), setting A_REMOVE_EMPTY_ERROR_FILE to “1” may cause error messages to be lost. For example, on UNIX if the error file is empty when one runtime exits, that runtime would delete the file. The file will remain deleted even if another runtime process subsequently writes a message to it. The default value for this variable is “0” (off, false, no).

A_RETRY_DELAY

This variable affects UNIX networks with multiple-user licenses. If A_LICENSE_RETRIES is set to a positive integer value, then the value of A_RETRY_DELAY determines how many seconds the runtime will wait between repeated attempts to register itself with the network license manager, **acushare**.

The default value is “10”.

A_SEQ_DEFAULT_BLOCK_SIZE

This configuration variable determines the size of the buffer to use when accessing a sequential file whose definition has no BLOCK CONTAINS clause. When set, A_SEQ_DEFAULT_BLOCK_SIZE specifies the size of the buffer in characters, rounded up to the nearest power of 2 that is greater than or equal to that value. The default value is “0”, which sets the block size to one record. Note that this variable does not apply to print files or to files with names that start with a hyphen followed by “D” or “P”.

You can set A_SEQ_DEFAULT_BLOCK_SIZE in the environment to allow the “**vutil** -load” command to buffer the input file according to the variable’s value. The maximum buffer size is 1 GB. If this variable is not set, the default buffer block size is 4096 bytes. If it is set to “0”, “**vutil** -load” performs record-based I/O on a sequential file.

A_SYSLOG_HOSTNAME

This variable applies only on Windows and works in conjunction with the **A_SYSLOG_ON_RUNTIME_ERROR** configuration variable. Set A_SYS_HOSTNAME to the server name or IP address on which the event log is located. Do not include any slashes with the server name. The default value for this variable is empty. Then set A_SYSLOG_ON_RUNTIME_ERROR to “1” (on, true, yes). Shutdown messages will be sent to the event log on the local machine.

A_SYSLOG_ON_RUNTIME_ERROR

When this variable is set to “1” (on, true, yes), on a fatal error, the runtime will send its shutdown error message to the UNIX syslog daemon, console, or Windows event log. The runtime uses the same logic as the C$SYSLOG routine. (See C$SYSLOG in Appendix I of the *ACUCOBOL-GT Appendices Manual* for more information). The error message also includes the name of the runtime error file so that the administrator can view it for more information. The default value for this variable is “0” (off, false, no).

ACCEPT_AUTO

This configuration variable applies only when running in HP COBOL compatibility mode (with the “-Cp” compiler option). The ACCEPT_AUTO configuration variable causes the runtime to treat all Format 1 ACCEPT statements as if the AUTO phrase is used, whether or not AUTO appears in the statement. Set this variable to “1” (on, true, yes) to enable this behavior. The default value is “0” (off, false, no).

ACCEPT_TIMEOUT

This variable causes all ACCEPT statements to time out just as if there was a BEFORE TIME phrase present in the ACCEPT statement. The value assigned to ACCEPT_TIMEOUT is the timeout period, in seconds. This timeout value is applied to every ACCEPT statement that can have a BEFORE TIME phrase specified for it. If a particular ACCEPT statement has a BEFORE TIME phrase explicitly coded for it, that phrase takes precedence and ACCEPT_TIMEOUT does not apply to that statement. The default value of ACCEPT_TIMEOUT is “0”, which indicates no timeout value.

ACTIVE_BORDER_COLOR

This variable is used on character-based hosts to specify the color and video attributes of the characters used to form the border (box) around the active floating window. ACTIVE_BORDER_COLOR can be set to a variety of numeric values that express combinations of color and video attributes. See the documentation for the COLOR phrase in the “Common Screen Options” section of the *ACUCOBOL-GT Reference Manual* (section 6.4.9).

If ACTIVE_BORDER_COLOR is set to “0”, the active window’s border is drawn with the colors and video attributes specified in the COBOL program when the window is initially created. The default value is “0”.

ACU_DUMP, ACU_DUMP_FILE, ACU_DUMP_WIDTH, ACU_DUMP_TABLE_LIMIT

These configuration variables are used to enable and configure the Abend Diagnostic Report (ADR) facility. For a complete description of the ADR, see section 3.1.9, in Book 1, *ACUCOBOL-GT User’s Guide*.

ACU_DUMP

This variable enables the Abend Diagnostic Report. The default value is “0” (off, false, no). Set ACU_DUMP to “1” (on, true, yes) to turn on the ADR.

ACU_DUMP_FILE

This variable specifies the name of the report file. It allows the following 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.
- 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.
- 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.
- 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.
- If the name contains the string “%u”, that string is replaced with the username.
- If the name contains the string “%h”, that string is replaced with the hostname.

The default value for ACU_DUMP_FILE is “acudump.%p”.

ACU_DUMP_WIDTH

This 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 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
```

ACU_USER_DIR

The ACU_USER_DIR configuration variable specifies the default location of a user debugger settings file. In the past, the ACUCOBOL variable has been used for this purpose. When set, ACU_USER_DIR specifies the directory for the user's debugger settings (".adb") file. The default value is "NULL", which causes the runtime to use the ACUCOBOL variable.

ACUCOBOL

This variable holds the full path to the ACUCOBOL-GT installation directory. For example, if the runtime is installed in "C:\Program Files\Acucorp\Acucbl8xx\AcuGT\bin", you would set this configuration variable to:

```
    ACUCOBOL C:\Program Files\Acucorp\Acucbl8xx\AcuGT
```

This variable is used to locate extensions to the runtime.

AGS_BLOCK_SLEEP_TIME

This variable is used to specify the amount of wait time before attempting to retry writing data to a socket. It can improve file I/O peformance times for large files (32K or larger). The value of this variable by default is 10 milliseconds. The default value should provide performance at par with pre 7.3 versions.

This variable only has impact on UNIX/Linux.

AGS_MAX_SEND_SIZE

This variable allows you to control the size of a basic socket packet exchanged between ACUCOBOL-GT applications that use sockets to communicate. The default value is 16000. In the vast majority of cases, the default value provides excellent results. However, when performance problems are traced to packet size, you can change the size with AGS_MAX_SEND_SIZE. The value of this variable is checked every time that data is sent to the socket. When a program changes the value, the new value is applied the next time that data is sent to the socket.

AGS_RECEIVE_BUFFER_SIZE

This variable determines the size of the low-level receive buffer for a socket connection. For the value to have an affect, it must be set before any sockets have been created. The default value is 16384. The default value should be sufficient for most cases. The receive-buffer-size is passed directly to a call to setsockopt.

Note: The value of this variable is sent to a lower-level socket layer not controlled by ACUCOBOL-GT. It may not have any noticeable effect. Changes in this value are not seen in response to a “U” debugger command listing the memory usage of the runtime.

AGS_SEND_BUFFER_SIZE

This variable determines the size of the low-level send buffer for a socket connection. For the value to have an affect, it must be set before any sockets have been created. The default value is 16384. The default value should be sufficient for most cases. The send-buffer-size is passed directly to a call to setsockopt.

Note: The value of this variable is sent to a lower-level socket layer not controlled by ACUCOBOL-GT. It may not have any noticeable effect. Changes in this value are not seen in response to a “U” debugger command listing the memory usage of the runtime.

AGS_SOCKET_COMPRESS

This variable determines the type of data compression performed at the internal socket layer. AGS_SOCKET_COMPRESS must be set before any socket communication is done, and cannot be changed via SET ENVIRONMENT. This variable has three possible values:

NONE	This is the default setting. When AGS_SOCKET_COMPRESS is set to this value, no compression is performed.
ZLIB	When AGS_SOCKET_COMPRESS is set to this value, socket data is compressed using the same algorithm as the **gzip** compression utility.
RUNLENGTH	When AGS_SOCKET_COMPRESS is set to this value, simple compression is done, based on counting repeated bytes of data.

RUNLENGTH compression tends to be very fast, while ZLIB compression tends to compress the data more, but is slower as a result.

Windows supports ZLIB compression, but not all UNIX machines do. For those machines that do not, RUNLENGTH compression will be used whether this variable is set to ZLIB or RUNLENGTH. When the compression algorithm is being negotiated with a server, the method that both machines support will be used.

AGS_SOCKET_ENCRYPT

To turn on encryption at the internal socket-layer, set the configuration variable AGS_SOCKET_ENCRYPT to "1" (on, true, yes). It must be set before any socket communication is performed, and cannot be changed via a SET ENVIRONMENT statement.

Note: If the variables AS_CLIENT_ENCRYPT and/or THIN_CLIENT_ENCRYPT are set to "1", AGS_SOCKET_ENCRYPT is also set to "1" automatically.

AGS_TCP_NODELAY

This variable determines whether the Nagle algorithm is used when sending socket buffer messages. This algorithm automatically delays sending small socket packets for a short period of time in order to increase network efficiency by sending them in a batch. Setting this variable to the default of "1" (on, true, yes) causes socket packets to be sent immediately (not using the algorithm), while setting this variable to "0" (off, false, no) causes socket packets to be delayed (using the algorithm). The TCP-NODELAY socket option is used as follows:

```
setsockopt(s, IPPROTO_TCP, TCP_NODELAY, &tcp_nodelay, sizeof(int));
```

The value of this variable is sent to a lower-level socket layer not controlled by ACUCOBOL-GT. It may not have any noticeable effect.

alfred Configuration variables

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 configuring alfred (as well as detailed user information) in PDF format from our Web site at the following address: **http://supportline.microfocus.com/examplesandutilities/index.asp.** From this page select **ACUCOBOL-GT Technical Articles and Tips > alfred Indexed File Record Editor**

ALLOW_FS_OVERRIDE

This variable enables you to determine if the actual EXTFH return status will be returned, or if the return status should be translated by the runtime. The default setting is "True" or "1" and will cause the actual EXTFH return status to be returned to the user. Setting this variable to "False" or "0" will cause the EXTFH return status to be translated by the runtime

ANSI_OUTPUT_IN_DEBUG

This variable prevents a COBOL program that uses ANSI-style DISPLAY statements from interfering with the runtime debugger window. This variable accepts two possible values: “CANVAS” or “TERMINAL”.

When set to “CANVAS” (the default setting) the runtime constructs a default canvas on which to place the ANSI output. This prevents the ANSI output from interfering with the debugger window. Note that if your COBOL program sends escape sequences to the terminal, this mode will cause those escape sequences to not have the intended result.

When set to “TERMINAL”, the runtime will send ANSI output to the terminal, possibly interfering with the view of the debugger window. This is how the runtime behaved before the implementation of this new feature.

Note that this configuration variable must be set before the runtime initializes the terminal manager, which means you cannot set this variable from a COBOL program.

APPLY_CODE_PATH

When set to “1” (on, true, yes), this variable causes the CODE_PREFIX variable to be applied to object files with full path names (those beginning with a “/” (forward slash). Otherwise, CODE_PREFIX is not applied to files with full path names. For example, if your application specifies the file:

```
/accounting/objects/payroll
```

and your CODE_PREFIX variable is set to:

```
CODE_PREFIX  /master_obj
```

and APPLY_CODE_PATH is set to “on”, the runtime will look for your file in:

```
/master_obj/accounting/objects/payroll
```

The default value of APPLY_CODE_PATH is “0” (off, false, no).

APPLY_FILE_PATH

When set to "1" (on, true, yes), this variable causes the FILE_PREFIX variable to be applied to data files with full path names (those beginning with "/", forward slash). Otherwise, FILE_PREFIX is not applied to files with full path names. For example, if your application specifies the file:

```
/accounting/data/ind.dat
```

and your FILE_PREFIX variable is set to:

```
FILE_PREFIX  /master_data
```

and APPLY_FILE_PATH is set to "on", the runtime will look for your file in:

```
/master_data/accounting/data/ind.dat
```

The default value of APPLY_FILE_PATH is "0" (off, false, no).

AUTO_DECIMAL

When set to "1" (on, true, yes), this variable checks the data item descriptions of numeric entry fields with a decimal point for the number of digits that must be filled to the right of the decimal point. When all the digits after the decimal point are entered, the field will terminate if the AUTO_TERMINATE phrase is specified. The number of digits to the right of the decimal point can vary, depending on how many are indicated in the picture of each numeric entry field. You must specify AUTO_TERMINATE phrase for this feature to work.

The exception to this is when an entry field has an AUTO_DECIMAL property specified, in which case, the coded value will be used.

The default value of this variable is "0" (off, false, no).

AUTO_PROMPT

When set to “1” (on, true, yes), this variable causes all ACCEPT statements without a PROMPT clause to be treated as if they had a PROMPT SPACES clause. This causes the screen to be erased at the field position prior to the data’s being entered. This variable is provided for compatibility with ACUCOBOL-85 Version 1.3 and earlier, which behaved this way. The default setting is “0” (off, false, no).

AXML_CREATE_SCHEMA

This variable is designed for use with AcuXML for instances when you want to include a schema or schema name with your XML output. In order for this variable to have an effect, AXML_CREATE_STYLE must be set to “schema” and AXML_SCHEMA_NAME must name the schema file. Once these conditions are met, this variable tells AcuXML whether to create a schema file with XML output, or simply include the name of a schema file in the output.

By default, when AXML_CREATE_STYLE is set to schema, AcuXML creates a schema file for all XML output. Because only one schema is typically required, you should set AXML_CREATE_SCHEMA to “FALSE” after the first time a schema is created. Then, only the name of the schema file will be included in the output XML file. Similarly, if you already have a schema file and don’t want AcuXML to overwrite it, set this variable to “FALSE.”

AXML_CREATE_STYLE

This variable is designed for use with AcuXML. Use it to define the type of XML output that ACUCOBOL-GT should generate when it creates XML files. It can be set to “DTD”, “SCHEMA” or “NONE”. Set this variable to “NONE” if you want the resulting XML file to be raw XML. Set it to “DTD” if you want the output to include a Document Type Definition of the elements in the document. Often, the party with whom you trade data may require that your XML document include a DTD.

Set this variable to “SCHEMA” if you want ACUCOBOL-GT to create a schema to describe the XML documents that it writes. Schemas provide the highest level of detail about the contents of the associated XML document, and are typically required for development purposes. If you set this variable to “SCHEMA”, you must use the AXML_SCHEMA_NAME variable to name the schema file.

Please note that creating a schema for a file that was run through the **xml2fd** utility with a schema won’t result in an identical schema. In addition, note that setting this variable to “schema” causes a schema to be created for every XML output file by default. Once the first schema is created, you should set AXML_CREATE_SCHEMA to “FALSE” to prevent schemas from being created on subsequent XML outputs.

AXML_ENCODING

This variable is designed for use with AcuXML. Use it when you want to specify a character encoding method for the XML files that ACUCOBOL-GT creates. By default, the XML output generated by ACUCOBOL-GT is mapped to the UTF-8 encoding system (compatible with the US-ASCII character set). If you want to use a different encoding system, for instance a European encoding system that includes the British pound character (£), change this variable to reflect the new system name. For example:

```
AXML_ENCODING ISO-8859-1
```

This variable causes encoding information to be added to the header of XML files created by ACUCOBOL-GT. With the configuration file entry shown above, the following header would be included:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

This header causes the ISO-8859-1 Latin encoding system to be applied to the data file as desired.

AcuXML supports the following encoding systems:

- UTF-8, default [8-bit Unicode Transformation Format, backwards compatible with US-ASCII]

- US-ASCII
- UTF-16 [16-bit Unicode Transformation Format]
- ISO-8859-1 [Latin 1, European encoding]

AXML_EXACT_TABLE_MATCH

This variable affects the behavior of AcuXML. By default, all tables in an FD must match data in the XML file with respect to the values of the indices. Therefore, AXML_EXACT_TABLE_MATCH is set to “1” (on, true, yes) by default. To disable this requirement, set AXML_EXACT_TABLE_MATCH to “0” (off, false, no).

AXML_IGNORE_EMPTY_DATA

Set this variable to “TRUE” to omit empty and zero-filled data from AcuXML’s output file. In this case, AcuXML will not write tags for alphabetic data items that are all blank or numeric data items that are “0”. When you set this variable from your COBOL program, it affects any records written via AcuXML from that point on. Note that setting this variable to “TRUE” could cause AcuXML to generate parts of an XML file that are not consistent with any DTD or schema associated with the file. As a result, use this variable with care.

The default value of “FALSE” causes AcuXML to generate tags for all data items in the file. If your records are mostly empty, this may be overkill.

AXML_SCHEMA_DOC

This variable is designed for use with AcuXML. Use it when you want to add a documentation element to the schema that ACUCOBOL-GT creates when it writes an XML file (such as whenever a sequential file is OPEN OUTPUT).

If you do not require specific documentation in the schema file, or if you did not request that schemas be created for XML output, you can omit this variable.

If this variable is set, its value is included in the documentation element of the resulting schema. For example, if you set this variable as follows:

```
AXML_SCHEMA_DOC This is the documentation to be
included in the file...
```

The schema will include the following data:

```
<xs:annotation>
  <xs:documentation>
    This is the documentation to be included in the file.
    Created by AcuXML version 6.0.0 on 2002/05/16
  </xs:documentation>
</xs:annotation>
```

Note: For information on working with XML data, see section 11.2 in *A Guide to Interoperating with ACUCOBOL-GT.*

AXML_SCHEMA_NAME

This variable is designed for use with AcuXML. Use it to define the name of the schema file that ACUCOBOL-GT writes, if any, when it creates an XML file. If this variable is not set, or if it is set to a file name that cannot be created (for whatever reason), a schema is not created.

Note: To tell ACUCOBOL-GT to create a schema, use the AXML_CREATE_STYLE and AXML_CREATE_SCHEMA variables.

AXML_SCHEMA_NAMESPACE_DATA

This variable is designed for use with AcuXML for instances when you want to include a schema or schema name with your XML output and you want precise control over the schema namespace string shown in the output. The default value of this variable is:

```
xmlns:xs=\"http://www.w3.org/2001/XMLSchema-instance\"
    xs:noNamespaceSchemaLocation=\"%s\"
```

By default, when ACUCOBOL-GT writes XML output (and a schema has been requested), it substitutes the “%s” in this variable with the name of the schema file specified with the AXML_SCHEMA_NAME configuration variable. For instance, if AXML_SCHEMA_NAME is set to “myschema”, ACUCOBOL-GT will include the following line in the XML output:

```
xmlns:xs=\"http://www.w3.org/2001/XMLSchema-instance\"
    xs:noNamespaceSchemaLocation=\"myschema.xsd\"
```

If you need something different than “myschema.xsd” written in the namespace output, add this variable to your configuration file and alter the namespace value in the quotes to meet your requirements.

Note: If you want to include a single “%” character in the namespace, add a second percent sign “%%” to the definition of this variable.

In general, the value of this variable is used in the standard C library printf() function as the first argument, and all printf() rules apply.

AXML_STYLESHEET_HREF and AXML_STYLESHEET_TYPE

These variables are designed for use with AcuXML. Use them when you want to associate an XML style sheet with the XML documents that ACUCOBOL-GT creates. When you set these variables to a non-blank value, ACUCOBOL-GT includes an XML-stylesheet comment in the beginning of the resulting XML files. For instance, the following entry:

```
    AXML_STYLESHEET_TYPE text/css
```

causes the following comment to be added to the beginning of the XML file:

```
<?xml-stylesheet type="text/css"?>
```

If you set both of these variables, as in the following example,

```
AXML_STYLESHEET_TYPE text/css
AXML_STYLESHEET_HREF mystyle.css
```

then a comment like the following is added to the file:

```
<?xml-stylesheet type="text/css" href="mystyle.css"?>
```

If you do not require specific stylesheet data in the XML file, you can omit these variables.

These variables may be toggled on and off during program execution. If you have set these variables and want to generate a xml file without a xml style sheet association, you can do this by adding these statements before opening the xml file:

```
SET ENVIRONMENT "AXML_STYLESHEET_TYPE" TO ""
SET ENVIRONMENT "AXML_STYLESHEET_HREF" TO ""
```

Note: Setting these variables to NULL or SPACE will NOT toggle the variable off.

BACKGROUND_INTENSITY

This variable is used to choose a background intensity. Use one of these values:

0	The runtime uses the default intensity, which is based on your hardware and operating environment. Under Windows, the default background intensity is high-intensity. The default value is “0”.
1	The runtime uses low-intensity.
2	The runtime uses high-intensity.

There are two important exceptions:

- The runtime always assigns low-intensity to the background if the background color is black. Using high-intensity would cause the background to be dark gray, which tends to make the screen look muddy.
- Many devices do not support a background intensity independent from the foreground intensity (most terminals, for example). When that is the case, the runtime declares the background intensity to be low-intensity.

BELL

When set to “0” (off, false, no), this variable will inhibit all bells generated by ACCEPT and DISPLAY statements. Note that this will override explicit WITH BELL clauses as well as implicit bells. The default setting is “1” (on, true, yes).

BOXED_FLOATING_WINDOWS

When this variable is set to “1” (on, true, yes) all floating windows displayed on character-based hosts are drawn with a border (box). If this variable is set to “0” (off, false, no), floating windows are drawn with a border only when the BOXED phrase appears in the statement that creates the window. The default value for this variable is “0” (off, false, no). This variable has an affect only on character-based host systems.

BTRV_MASS_UPDATE

When this variable is set to “1” (on, true, yes), a Btrieve file is opened in exclusive mode. No other processes may open the file at the same time. When this variable is set to “0” (off, false, no), a Btrieve file is opened in accelerated mode, and other processes may open the file.

BTRV_NOWRITE_WAIT

When a user tries to write to a locked file, the Btrieve interface performs a 15-second “wait and retry” operation before it reports an error condition (99) to the runtime. Setting the BTRV_NOWRITE_WAIT configuration variable to “TRUE” (the default) prevents this operation from occurring, and the error condition is reported immediately. Setting BTRV_NOWRITE_WAIT to “FALSE” causes the interface to perform the wait and retry operation.

BTRV_USE_REPEAT_DUPS

This variable controls whether duplicate keys are created as LINKED duplicates (the Btrieve default) or REPEATING duplicates. When set to the default value of “FALSE”, the Btrieve interface creates all duplicate keys as LINKED duplicates. When set to “TRUE”, the Btrieve interface creates all duplicate keys as REPEATING duplicates.

In cases where a large number of users are accessing files, you may experience better performance if you set this variable to “TRUE”. See the Pervasive documentation for information on REPEATING duplicates and why you may want to use them.

BUFFERED_SCREEN

This variable controls how the Terminal Manager should buffer its output on UNIX systems. Normally, all queued output is sent to the screen after each DISPLAY statement. If this value is set to “1” (on, true, yes), then output is sent only when the internal buffer is full, an ACCEPT statement is executed, or an internal 1-second timer expires. This can speed up output on some systems by reducing the number of times the operating system is called. It will also cause a short delay before messages are seen. We recommend keeping this setting at the default “0” (off, false, no) unless you are experiencing poor screen performance.

CALL_HASH_SIZE

The setting of this variable controls the size of the hash table that tracks CALL statements to COBOL subprograms. Each CALL statement tracks its last resolution (target object, entry point, and owning thread). When the resolution is unchanged in a subsequent execution of the CALL statement, the CALL uses the saved information, contributing to improved performance. Each program contains its own copy of this table, so the size should generally be set to a small value.

The default value for CALL_HASH_SIZE is "31". The only reason to change this setting is if your programs contain hundreds of individual CALL statements that target distinct objects. In this case, you may see a small performance improvement by setting CALL_HASH_SIZE to a larger value. You can disable the tracking of these CALL statements by setting the value of CALL_HASH_SIZE to "0".

Note that this mechanism consumes a small amount of memory for each CALL statement. This memory is recovered when the calling object is removed from memory. The amount is machine-specific, but is normally well under 100 bytes per CALL.

CANCEL_ALL_DLLS

This variable is used to change the default behavior of a CANCEL ALL statement. The default behavior is for CANCEL ALL to free all DLLs and UNIX/Linux shared object libraries loaded with a prior CALL statement. Setting CANCEL_ALL_DLLS to "0" (off, false, no) indicates that CANCEL ALL should not free any DLLs or shared object libraries. If you want to free a particular DLL or shared library when CANCEL_ALL_DLLS is set to "0", you must specify the DLL's name in a CANCEL statement.

The default value of CANCEL_ALL_DLLS is "1" (on, true, yes).

CARRIAGE_CONTROL_FILTER

The value of this variable affects how carriage control characters are treated when found in LINE SEQUENTIAL data files.

RM/COBOL version 2 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.

You can retain any or all of these characters in the input record by setting CARRIAGE_CONTROL_FILTER to a value as follows:

1	form-feed characters are retained
2	carriage return characters are retained
4	line-feed characters are retained

You can specify two or three characters to be retained by adding the appropriate values together. For example, a value of “6” retains carriage returns and line feeds (2 plus 4). Setting the variable to “0” causes the default action of removing all three characters.

The default value is “0”.

Note: 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.

CBLHELP

Define the CBLHELP configuration variable to the location of the “cblhelp” debugger help file. The definition must include the path and filename. For example:

```
CBLHELP /home/acucobol8/etc/cblhelp
```

CGI_AUTO_HEADER

This variable is used when you are writing a Common Gateway Interface (CGI) program in COBOL. It allows you to suppress the output of the HTML header.

Set CGI_AUTO_HEADER to "0" (off, false, no) if you want to suppress the output of the HTML header. This can be useful when you want to execute a CGI program and include its output into an existing flow of HTML text. For example, with server-side includes, or SSI, you can instruct the Web server to execute a subprogram in the manner of CGI and then incorporate its output right into the HTML document before sending it to the requesting client. The default value is "1" (on, true, yes).

For information about writing a CGI program in COBOL, refer to Chapter 4 in *A Programmer's Guide to the Internet*.

CGI_CLEAR_MISSING_VALUES

This variable is used when you are writing a Common Gateway Interface (CGI) program in COBOL. It allows you to control the behavior of the ACCEPT statement when CGI variables do not exist in the CGI input data.

By default, ACCEPT sets the value of numeric data items to zero and non-numeric data items to spaces if a CGI variable does not exist. Set the CGI_CLEAR_MISSING_VALUES configuration variable to "0" (off, false, no) if you do not want ACCEPT to change the value of the data item if the corresponding CGI variable is missing from the CGI input data.

CGI_CONTENT_TYPE

By default, the output generated by your CGI program is mapped as HTML content. To associate your CGI output with a MIME content type other than "text/html", use the CGI_CONTENT_TYPE configuration variable. This variable lets you control the content type information in the header of output files created by ACUCOBOL-GT. Such information informs recipients of the type of content that they are about to receive.

Using this variable, you can configure your CGI program for many types of output, including eXtensible Markup Language (XML) or Wireless Markup Language (WML) for Wireless Application Protocol (WAP) devices like mobile phones.

Whichever format you choose, the US-ASCII character set is applied to the output by default. If you want the CGI output to be mapped to an alternate character set such as ISO-8859-I (Western European), then you can specify the character encoding set to use with the variable as well.

Include this variable in your runtime configuration file as follows:

```
CGI_CONTENT_TYPE contenttype; charset=encoding_set
```

Where *contenttype* is the MIME content type of the generated output, and *encoding_set* is the preferred character encoding set to use.

For example, the WML content type for WAP mobile phones is “text/vnd.wap.wml”. To associate your CGI output with WML, include the following in your configuration file:

```
    CGI_CONTENT_TYPE text/vnd.wap.wml
```

If you want your WML output to be mapped to the Western European character set, include the following:

```
    CGI_CONTENT_TYPE text/vnd.wap.wml; charset=iso-8859-I
```

The content type for eXtensible Markup Language (XML) documents is “text/xml”. If your program generates XML data, include the following:

```
    CGI_CONTENT_TYPE text/xml
```

Caution: To avoid overriding other Content-Type associations, we suggest that you create a different configuration file for each of the MIME Content-Type associations that you make in your Web server setup.

Please note that if you use this variable, the external forms indicated in your program’s DISPLAY syntax *must* contain the appropriate content. In other words, if you associate your program with the “text/xml” content type, the forms must be “.xml” documents with XML syntax. If you associate it with “text/vnd.wap.wml”, the forms must be “.wml” documents with WML syntax. Your program can DISPLAY virtually any type of data, as long as the Content-Type ID corresponds to the external form file that you provide.

Be aware that if you do not use the proper file extension for your external form documents, the Web server will interpret the data as HTML and display the wrong data. WML and XML are also more sensitive to syntax errors than HTML.

In addition, note that the capabilities of the configuration entry CGI_NO_CACHE may be affected by the content type that you choose.

For information about writing a CGI program in COBOL, refer to Chapter 4 in *A Programmer's Guide to the Internet*.

CGI_NO_CACHE

This variable allows you to choose whether the HTML output of your Common Gateway Interface (CGI) program will be cached by the requesting client.

The default value is "1" (on, true, yes), which means there is no caching. By default, the runtime generates "Pragma: no-cache" in the HTML response header that gets sent to the standard output stream. If you set CGI_NO_CACHE to "0" (off, false, no), the runtime suppresses this line of the response header, and the requesting client caches the output.

For information about writing a CGI program in COBOL, refer to Chapter 4 in *A Programmer's Guide to the Internet*.

CGI_STRIP_CR

When this variable is set to "1" (on, true, yes), the runtime automatically removes carriage return characters from data entered in HTML TEXTAREAS (multiple line entry-fields). Stripping the carriage returns from this kind of input prevents double-spacing problems, as well as conflicts that may arise if the data is used in a context that does not expect a carriage return character to precede each line feed character. Some browsers send a carriage return line feed sequence to the CGI program, and when this sequence is written to a file on operating systems that terminate text lines with line feed characters only, the file may appear to be double spaced. The default value for this variable is "0" (off, false, no).

For example, if you enter the following three lines in a TEXTAREA for a field called "thetext":

```
Sometext line 1
Sometext line 2
Sometext line 3
```

The browser sends the following to the CGI program:

```
thetext=Sometext+line+1%0D%0ASometext+line+2%0D%0ASometext+line+3%0D%0A
```

If the CGI_STRIP_CR is set to "1" (on, true, yes), the runtime strips the carriage return characters so that the input line is the following:

```
thetext=Sometext+line+1%0ASometext+line+2%0ASometext+line+3%0A
```

For information about writing a CGI program in COBOL, refer to Chapter 4 in *A Programmer's Guide to the Internet*.

CHAIN_MENUS

When this variable is set to "1" (on, true, yes), the runtime system automatically destroys any menu displayed by a program performing a CHAIN or CALL PROGRAM. This destruction is accomplished with the WMENU-DESTROY-DELAYED operation of the W$MENU library routine. The effect is that the menu is not actually destroyed until the chained-to program displays a new menu. Setting this variable to "0" (off, false, no) inhibits the destruction of the menu. The default value is "off".

CHECK_USING

When this value is "1" (on, true, yes), the runtime system tests each use of a LINKAGE data item to make sure 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.

Setting this value to "0" (off, false, no) inhibits the parameters size matching test. It also inhibits the runtime test that verifies that all parameters of a subprogram are passed by the caller.

The default value is “1”. If you set this value to “0”, you should test your programs carefully to avoid corrupting memory.

Note: It is common for programs in some OLTP environments to specify a data item length as a negative value. By default, this produces a runtime error. Set CHECK_USING to “0” to override the default behavior.

CISAM_COMPRESS_KEYS

This variable allows you to turn off key compression in C-ISAM files. By default, the ACUCOBOL-GT interface to C-ISAM uses the key compression feature of C-ISAM. But some C-ISAM emulators do not understand the compressed keys and cannot read the files created. This variable allows you to turn off the compression.

When the variable is set to “0” (off, false, no), key compression is not used. When it’s set to the default of “1” (on, true, yes), key compression is used. Note that this value is examined each time a file is created, so its setting can be changed for each file. The setting is meaningful only when the file is created. After that, the file retains its compression mode.

CLOSE_ON_EXIT

When set to “1” (on, true, yes), this variable enables the automatic closing of all files except print files when a program performs an EXIT PROGRAM statement. When set to “2” it enables the automatic closing of all files when a program exits. When set to “0” (off, false, no), no files will be automatically closed. For more information, see the *ACUCOBOL-GT User’s Guide*, section 2.7.5, “File Handling Options.” The default value is “0”.

COBLPFORM

This configuration variable is used to define and print to printer channels C01-C12. Specify the line numbers for each channel with the COBLPFORM configuration variable. Null entries are ignored. Those channels that have line number zero, function-names S01-S052, CSP, or are undefined, are set to line 1.

Example 1

```
COBLPFORM 1:3:5:7:9:11:13:15:17:19:21:23
```

In this example C01 equals 1, C02 equals 3, and so on.

Example 2

```
COBLPFORM :3::5: :9
```

In this example, C01 equals 3, C02 equals 5, C03 equals 1, and C04 equals 9. You can specify only a single line number for each channel.

In example 2 above, channels C05 - C12 are undefined. If a print statement specifies channel C05 - C12, the line is printed at line 1. In addition, in the example shown, C03 equals 1 because its value is a space and therefore undefined.

Any WRITE BEFORE/AFTER PAGE statements cause positioning to be at line 1. Each line advance increases the line number by one. A request to skip to a line number less than or equal to the current line causes a new page to begin. The appropriate number of line feeds are then generated.

CODE_CASE

This configuration variable allows you to adjust the case of an object file name that is specified in a CALL statement. It has five possible values:

NONE	or	“0”	(the default) object file names are not translated
LOWER	or	“1”	object file names are translated to lower case, including directory (path) elements

UPPER	or	“2”	object file names are translated to upper case, including directory (path) elements
LOWER_BASE	or	“3”	object file names are translated to lower case, excluding directory (path) elements
UPPER_BASE	or	“4”	object file names are translated to upper case, excluding directory (path) elements

Translation occurs before the CODE_SUFFIX and CODE_PREFIX configuration options are applied. You should make sure that those variables specify the correct case. For a complete description of the runtime CALL handling procedure, see section 2.9.1 in Book 1, *ACUCOBOL-GT User's Guide*.

CODE_MAPPING

This configuration variable allows you to modify CALL, CHAIN, and CANCEL names at runtime. This can be particularly useful if you are using AcuServer or AcuConnect. When this variable is set to “1” (on, true, yes), every CALL, CHAIN, and CANCEL statement checks the current configuration for a name that matches the CALL name. This is handled in the same way that file name processing is done (the environment is checked for an uppercase version of the name, with any hyphens treated as underscores). If a matching name is found, its value is substituted. This is done recursively until no more matching names are found.

After this substitution occurs, the CALL name handling proceeds normally (and includes any effects of CODE_PATH, CODE_SUFFIX, and CODE_CASE).

For example, with CODE_MAPPING set to “1”, if your configuration file had the following entry:

```
    MYPROG  @sun:/app/myprog
```

Then CALL “MYPROG” would act the same as CALL “@sun:/app/myprog”.

Thin client applications may find the CODE_MAPPING mechanism useful for automatically adding the "@[DISPLAY]:" prefix to the name of the DLL to run on the display host. For example, if your configuration file includes the entry:

```
mylib.dll  @[DISPLAY]:mylib.dll
```

Then the statement

```
CALL  "mylib.dll"
```

is interpreted as

```
CALL "@[DISPLAY]:mylib.dll"
```

causing "mylib.dll" to run on the display host.

Those wanting to specify the DLL calling conventions will also find CODE_MAPPING useful. For example, if you use the following configuration entries:

```
funcA=funcA@__stdcall
funcB=funcB@__cdecl
```

then the statement

```
CALL "funcA"
```

calls funcA using the stdcall calling convention and

```
CALL "funcB"
```

calls funcB using the cdecl convention.

For more information about calling DLLs from thin client applications, see section 7.2.6 of the *AcuConnect User's Guide*. For information on calling DLLs in general, refer to Chapter 3 of *A Guide to Interoperating with ACUCOBOL-GT.*

The default value for this variable is "0" (off, false, no).

CODE_PREFIX

This variable defines a set of directories that the runtime searches to locate a program object file. The default value is "." (current working directory). Code and data file search paths are described in more detail in section 2.7.2 of the *ACUCOBOL-GT User's Guide*.

Directories can be a mix of relative and absolute paths. Entries are separated by spaces. A space is a valid separator on all systems. Alternatively, on UNIX systems you can also separate entries with a colon. On Windows systems a semicolon can be used. On VMS systems a comma can be used.

Include a "^" (carat) to specify the directory containing the calling program. For example:

```
CODE_PREFIX . /cobbin ^
```

causes the runtime to search the current working directory, followed by the "cobbin" root directory, followed by the directory containing the calling program.

You can specify a directory path that contains embedded spaces if you surround the path with quotation marks. For example:

```
CODE_PREFIX C:\“program files” C:\Customers
```

Remote name notation is allowed if your runtime is *client-enabled*. See *User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

Up to 4096 characters can be specified for the value of this variable.

CODE_SUFFIX

The value of this variable is automatically appended to the end of program filenames when those names do not contain explicit suffixes. A suffix is the portion of a filename that follows a period. For example, if CODE__SUFFIX is set to "COB", then CALL "PGMFILE" causes the runtime to look for the file "PGMFILE.COB". The default value is empty.

CODE_SYSTEM

The runtime configuration variable CODE_SYSTEM tells the runtime if double-byte character data is being accepted or displayed, and which code system (that is, which standard for encoding Japanese and other Asian character sets, for example) is being used. Each code system has a range of values that it allows within each byte of a two-byte character, so identifying the code system allows the runtime to recognize character boundaries when it is processing double-byte data for ACCEPT and DISPLAY statements.

Setting CODE_SYSTEM to the proper value allows your COBOL applications to handle input and display of double-byte character data without source program changes. The syntax is:

```
CODE_SYSTEM  setting
```

The table below shows the possible settings of the CODE_SYSTEM variable, the code system to which each setting refers, and some examples of operating systems to which the particular code system applies:

Setting	**Code System**	**Op. System Examples**
BIG5	Big Five (Taiwan)	Chinese DOS, Windows
DBC	ACUCOBOL-GT Generic Double-byte Coding Scheme	other double-byte machines
EUC	Extended UNIX	Most UNIX machines
GB	Code of Chinese Graphic Character Set (People's Republic of China)	Chinese DOS, Windows
KSC	Korean Character Standard	Korean DOS
SJC	Shift JIS Code (Japanese Industrial Standard)	DOS/V, Windows, some UNIX machines

The default "0" means ASCII or EBCDIC single-byte characters.

The following table shows the decimal values that the respective code systems allow for each byte of the two-byte character:

Code System Setting	**1st byte**	**2nd byte**
BIG5	161 - 254	64 - 126
(second format)	161 - 254	161 - 254
DBC	128 - 255	128 - 255
EUC	142	161 - 223
(second format)	161 - 254	161 - 254
GB and KSC	161- 254	161 - 254
SJC	129 - 159	64 - 252 (not 127)
(second format)	224 - 239	64 - 252 (not 127)

Note: The first and second byte values are co-dependent; that is, both values must fall within the respective ranges shown in the table. If either value is not within its allowable range, then each byte will be treated as a single character.

COLOR_MAP

This variable can be used to assign colors to programs that do not contain explicit color settings. This is described in section 4.4.1 of *ACUCOBOL-GT User's Guide*. The default value is empty.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

COLOR_MODEL

This variable is typically used when a character-based application is moved to a graphical environment. Use the COLOR_MODEL setting to perform uniform changes to your program's color scheme. These changes are represented by rules that act on your colors. An example of a rule is "exchange the foreground and background colors". Use COLOR_MODEL to change your color scheme in a global way.

The default color model is model "0". It causes no changes to occur to your color scheme. The remaining 10 models are "1" through "10".

- The odd-numbered models transform only those parts of your program that are entirely black and white. Any character position that contains any color is left unchanged.
- The even-numbered models apply the changes regardless of color. When selecting a COLOR_MODEL, you can ignore the even-numbered models if you are satisfied with the color portions of your program.

Each color model is actually a composite; it's the equivalent of two or more configuration file variable settings:

COLOR_MODEL	**Equivalent Configuration File Variable Settings**
"1"	COLOR_TRANS "5" INTENSITY_FLAGS "34" BACKGROUND_INTENSITY "1"
"2"	COLOR_TRANS "4" INTENSITY_FLAGS "34" BACKGROUND_INTENSITY "1"
"3"	COLOR_TRANS "3" INTENSITY_FLAGS "34"
"4"	COLOR_TRANS "1" INTENSITY_FLAGS "34"
"5"	COLOR_TRANS "1" INTENSITY_FLAGS "129"
"6"	COLOR_TRANS "1" INTENSITY_FLAGS "129" BACKGROUND_INTENSITY "2"
"7"	COLOR_TRANS "3" INTENSITY_FLAGS "161"

COLOR_MODEL	**Equivalent Configuration File Variable Settings**
“8”	COLOR_TRANS “1” INTENSITY_FLAGS “161”
“9”	COLOR_TRANS “3” INTENSITY_FLAGS “193”
“10”	COLOR_TRANS “1” INTENSITY_FLAGS “193”

For more information, see Chapter 9 in Book 2, *ACUCOBOL-GT User Interface Programming*.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

COLOR_TABLE

This variable is typically used when a character-based application is moved to a graphical environment. Use the COLOR_TABLE variable to cause transformations of individual color combinations. For example, a COLOR_TABLE entry might cause a red foreground on a black background to be translated to a white foreground on a blue background.

Follow the word COLOR_TABLE with the original foreground and background numbers, separated by a comma. Follow these by an equals sign and then the new foreground and background numbers, separated by a comma.

For example, to transform the color combination of foreground 5 on background 2, to foreground 13 on background 2, you would use:

```
COLOR_TABLE   5, 2 = 13, 2
```

These are the possible values for foreground and background settings:

Color	**Color value**
low-intensity Black	1

Color	**Color value**
low-intensity Blue	2
low-intensity Green	3
low-intensity Cyan	4
low-intensity Red	5
low-intensity Magenta	6
low-intensity Brown	7
low-intensity White	8
high-intensity Black	9
high-intensity Blue	10
high-intensity Green	11
high-intensity Cyan	12
high-intensity Red	13
high-intensity Magenta	14
high-intensity Brown	15
high-intensity White	16

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

COLOR_TRANS

This variable is typically used when a character-based application is moved to a graphical environment. It determines how the initial colors in an application are transformed. By default, it is set to “0”, which causes no transformation. It may be set to any of these values:

1	This mode causes the foreground and background colors to be exchanged for each other. This is equivalent to running the entire program in reverse-video.
2	This causes white to be exchanged for black and black to be exchanged for white. The foreground and background colors are transformed independently. For example, a green foreground on a black background would turn into a green foreground on a white background. This setting usually has the effect of transforming a black background into white while maintaining the general color scheme of the application.
3	The foreground and background colors are exchanged for each other, but only if they are both black or white. If either the foreground or background contains a color other than black or white, then nothing happens. This is equivalent to running the monochrome parts of your program in reverse-video while maintaining the color portions unchanged.
4	The foreground and background colors are exchanged for each other, but only if the background is black. This mode ensures that you never have a black background.
5	If the colors are foreground white and background black, they are exchanged for each other. Otherwise, nothing happens.

Generally speaking, you could use the COLOR_TRANS variable as a starting point in converting an application to appear more natural under Windows. (It’s easier to start with COLOR_MODEL instead.) Note that if your application is entirely black-and-white, then the first three COLOR_TRANS options are essentially identical. See Chapter 9 in Book 2, *ACUCOBOL-GT User Interface Programming* for color mapping suggestions.

COLUMN_SEPARATION

This configuration variable sets the default separation distance between columns in a list box. The value is expressed in 10ths of characters. For example, to place a 1/2 character space between list box columns, you would assign a value of "5". See the description of the list box SEPARATION property for more information. The default value of COLUMN_SEPARATION is "5".

COMPRESS_FACTOR

This variable is used to define the compression factor that is applied to indexed files (if the indexed file system supports compression; Vision does). COMPRESS_FACTOR is applied when a file is created with the WITH COMPRESSION phrase in the ASSIGN clause of the file's SELECT and the COMPRESSION CONTROL VALUE phrase is either omitted or specifies a value of "1". If the COMPRESS CONTROL VALUE phrase specifies a value other than one, that value is used and the value of COMPRESS_FACTOR is ignored.

COMPRESS_FACTOR can be set to any value within the range zero to 100. Zero specifies no compression. Values from 2-100 are treated as a percentage that specifies how much of the space saved by file compression is removed from the compressed records. A value of 1, the default, is a special case that causes the standard default compression factor of 70 to be applied. Note that a file's compression factor is set when the file is created and cannot later be changed except by recreating the file or rebuilding the file with **vutil**. For more information about Vision record compression, see Book 1, *ACUCOBOL-GT User's Guide,* section 6.1.6.1, "Compression."

COMPRESS_FILES

Setting this configuration variable to "1" (on, true, yes) causes ACUCOBOL-GT to treat all indexed files as if they had the WITH COMPRESSION phrase specified for them. This affects the status of newly created files only. When the configuration variable is set to the default value of "0" (off, false, no), only those files with the WITH COMPRESSION phrase specified will be compressed. You can specify the amount of compression with the **COMPRESS_FACTOR** configuration variable.

CONTROL_CREATION_EVENTS

This variable applies to those using ActiveX controls in their ACUCOBOL-GT programs. Use it if you want to allow events during the creation of an ActiveX control. By default, the runtime ignores events from all controls while it is creating an ActiveX control. If it did not, subsequent operations on the ActiveX control could fail.

If you are using a control that delivers significant information using events and you don't want to miss those events while you are creating a new control, set the CONTROL_CREATION_EVENTS variable to "1" (On, True, Yes). Alternatively, you could avoid creating an ActiveX control when you are expecting an event.

By default, this variable is set to "0" (Off, False, No).

CURRENCY

This configuration variable can be used to set the desired currency character at runtime. It is followed with the desired character. The default is to use the character specified in the source program's CURRENCY phrase (or "$" if the CURRENCY phrase is absent).

CURSOR_MODE

This configuration variable determines when the cursor should be visible. It has three values:

1	always visible
2	always invisible
3	invisible except during ACCEPT statements, then visible

The default value is “3”. Note that a change to the value does not take effect until the next ACCEPT or DISPLAY statement. The sample program MENUBAR.CBL contains examples of how to modify the cursor from within a program. The cursor is always set to Normal, Visible when the runtime exits or when the SYSTEM library routine is called.

CURSOR_TYPE

This configuration variable determines the way the cursor looks on character-based systems. It can be set to one of the following values (“3” is the default):

1	normal cursor (usually underscore)
2	bright cursor (usually block)
3	normal cursor except when in insert mode, then bright
4	vertical bar (when available)

DEBUG_NEWCOPY

This variable determines whether a new copy of a COBOL program being debugged is loaded from disk whenever the debugger is active. By default, DEBUG_NEWCOPY is “True” so that you can continue to use the logical cancel and code caching feature while the debugger is active.

Set DEBUG_NEWCOPY to “False” if you want to keep caching enabled and have the debugger use the copy of the program in the cache instead of reading a new copy from disk. You must then do one of two things:

- Start the debugger before the first execution of the program in the current process
- In a transaction processing system, use the CICS command, CEMT SET PROGRAM(*program_name*) NEWCOPY, to load a new copy of the program to be debugged.

Note: The ACUCOBOL-GT debugger periodically reads source code from the object file on disk. When the program is cached (as the result of a logical cancel), the object file is closed and could be replaced or deleted. For the debugger to function correctly, it must keep the object file open and ensure that the object code in the disk file is identical to the code in memory. Therefore, if the program has been cached (using LOGICAL_CANCELS and DYNAMIC_MEMORY_LIMIT), the debugger unloads the program from the cache, reopens the object file, and reloads the object code from memory. For more information, see section 6.3, “Memory Management,” in Book 1, *ACUCOBOL-GT User’s Guide*.

DECIMAL_POINT

This configuration variable sets the character to be used as the program’s decimal point. Follow it with the desired character. If you use this variable to set the decimal point to a comma, then the place and function of the decimal point and comma are reversed (just like the phrase DECIMAL_POINT IS COMMA). The default is to use the decimal point specified by the program’s source.

Note: You do not have to change the value of DECIMAL_POINT to match the decimal point used by floating point values received from external components. The runtime automatically makes the correct adjustment.

DEFAULT_FILESYSTEM

This variable determines the file system to be used if no *filename*_FILESYSTEM variable is set for a file and none of the other file system variables are set for the file type. The other variables you can use to specify a different file system for indexed, relative, or sequential files, are:

DEFAULT_IDX_FILESYSTEM
DEFAULT_REL_FILESYSTEM
DEFAULT_SEQ_FILESYSTEM

For example, setting:

```
DEFAULT_IDX_FILESYSTEM EXTFH
```

causes all indexed files to go through the EXTH interface. Unless another file system is specified, ACUCOBOL-GT uses its native file handler for relative and sequential files.

Note: The DEFAULT_IDX_FILESYSTEM variable is a synonym for the existing configuration variable, DEFAULT_HOST.

By default, all file access is handled by the ACUCOBOL-GT native file handler. For those file types you want to access using an EXTFH library, you need to set one or more of these configuration variables to “EXTFH”.

For example, to use the DB2 library to access indexed files, you would set the following two configuration variables:

```
A_EXTFH_LIB=/usr/lpp/cics/lib/libxfhdb2sa.a(libxfhdb2_shr.o)
DEFAULT_IDX_FILESYSTEM=EXTFH
```

For information on specifying EXTFH library and function names to use with the EXTFH interface, see section 11.6, “Working With an EXTFH Interface,” in *A Guide to Interoperating with ACUCOBOL-GT.*

DEFAULT_FONT

This variable defines which font to use for the DEFAULT_FONT (for a description of this font, see Format 3, ACCEPT FROM OBJECT in section 6.6 in Book 3, *ACUCOBOL-GT Reference Manual*). When DEFAULT_FONT is set to “0” (the normal setting), the font used depends on the host system as follows:

System	**Font Used**
Graphical system	MEDIUM-FONT
Non-graphical system	FIXED-FONT

You can set DEFAULT_FONT to one of the following values to use a different font. The following words are valid settings:

Setting	**Font Used**
TRADITIONAL	TRADITIONAL-FONT
FIXED	FIXED-FONT
LARGE	LARGE-FONT
MEDIUM	MEDIUM-FONT
SMALL	SMALL-FONT

Due to the way the runtime initializes the windowing subsystem, the DEFAULT_FONT setting is effective only when it is placed in the configuration file or the host system's environment. Setting DEFAULT_FONT from inside a COBOL program has no effect.

DEFAULT_HOST

When the application program is opening an existing file or creating a new file, you need to tell the runtime which file system to use. You accomplish this with one of two configuration variables: DEFAULT_HOST or *filename*_HOST.

```
DEFAULT_HOST filesystem
```

designates the file system to be used for files that are not individually assigned. If this variable is not given a value, and if you have not individually assigned a file system (with *filename*_HOST), the Vision file system is used.

Note: The DEFAULT_IDX_FILESYSTEM variable is a synonym for DEFAULT_HOST.

DEFAULT_MAP_FILE

Use this variable to point to the character map file used for translating international character sets between machines that use differing character codes. The map file is a simple text file that you create with an editor of your choice. Each line in the map file must contain two values in either decimal or hexadecimal: the character code of the character on the client machine, and the character code of the same character on the remote machine. Use a # sign to indicate a comment.

The runtime first searches for the configuration variable *server*_MAP_FILE and, if it is found, uses that setting to locate the map file. If that variable is not set, the runtime searches for DEFAULT_MAP_FILE. If this variable is not set, then no character translation is done.

Example:

```
    DEFAULT_MAP_FILE = c:\etc\pc_iso.txt
```

DEFAULT_PROGRAM

Use this variable to specify the name of the program to be run by default if no program name is specified on the command line. The name you give here is treated exactly as it would be if you had typed it on the command line. The default is "cbl.out".

Remote name notation is allowed for this variable if your runtime is client-enabled. See *ACUCOBOL-GT User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

DEFAULT_TIMEOUT

This variable is used by the runtime and Web Runtime to define the length of time, in seconds, that they will wait for a response from **acuserve** before timing out. The default value for this variable is 25 seconds. Some networks have long connect times and the default value may not be long enough to allow the application to connect. For example, to change the timeout default of 25 seconds to one minute, you would set the following:

```
DEFAULT_TIMEOUT = 60
```

If the runtime or Web Runtime receives an error before the specified time, they will time out immediately. This variable only works with AcuServer client runtimes and AcuServer client Web Runtimes.

DISABLED_CONTROL_COLOR

This variable allows character-based hosts to use color and video attributes to distinguish disabled screen controls from enabled controls. It can be set to a variety of numeric values that express combinations of attributes. When it is set to “0” (off, false, no), disabled controls appear the same as enabled ones. See section 6.4.9, “Common Screen Options, COLOR phrase” in Book 3, *ACUCOBOL-GT Reference Manual,* for a description of other numeric values that can be used.

DISPLAY_SWITCH_PERIOD

This variable helps to determine how frequently the program’s threads will switch control. After a thread performs the value of DISPLAY_SWITCH_PERIOD display operations, the runtime switches control to another thread (if one exists). Note that because a single DISPLAY statement can compile into multiple “display operations,” and because thread switching is also affected by other program operations (such as file I/O), it is impossible to predict or control when a thread will change control based on the presence of DISPLAY statements in the source.

By setting DISPLAY_SWITCH_PERIOD to lower values, you cause windows that are updated by multiple threads to update more uniformly, but more time will be spent in the thread switching code. Setting DISPLAY_SWITCH_PERIOD to higher values will decrease the switching overhead, but will also cause the windows to update in blocks. In most cases, applications that use threads will run well with the default setting of “10”.

DLL_CONVENTION

This variable allows you to specify the calling convention used to call DLLs. When this variable is set to “0”, the cdecl (standard C) interface is used. When this variable is set to “1”, the stdcall (Pascal/WINAPI) interface is used. The default for this variable is “0”.

Note that there are a few ways to override the DLL_CONVENTION setting:

- You can specify a list of DLL names and calling conventions in the SHARED_LIBRARY_LIST configuration variable. This variable can be set in the environment, in the runtime configuration file, or programmatically with the SET ENVIRONMENT statement.
- You can specify the calling convention for individual library functions in the COBOL CALL statement.
- You can set the CODE_MAPPING variable to “1”, then use configuration entries to specify the calling convention for individual functions.
- You can specify a list of DLL names and calling conventions using the “-y” runtime option.

In all of these cases, the runtime uses the specified calling convention and ignores the value of the DLL_CONVENTION configuration variable. See Chapter 3 in *A Guide to Interoperating with ACUCOBOL-GT* for more details about calling DLLs.

DLL_SUB_INTERFACE

This variable identifies the routine to be used as the “sub” interface routine within a DLL. It applies only to Windows systems. Set DLL_SUB_INTERFACE to the name of the routine you want to use. This name may be “sub” or any name you choose. The runtime checks DLL_SUB_INTERFACE when a DLL is loaded. You may change its value afterwards without any effect on DLLs that have already been loaded.

If DLL_SUB_INTERFACE is empty (default), the runtime does not look for a “sub” interface routine in a called DLL.

DLL_USE_SYSTEM_DIR

When a program calls an unloaded DLL, the value of this variable determines whether the runtime attempts to find the DLL in the Windows and System folders. When set to the default value "1" (on, true, yes), the runtime looks in the Windows and System folders. When set to "0" (off, false, no) the runtime does not look in the Windows and System folders. See Chapter 3 in *A Guide to Interoperating with ACUCOBOL-GT* for more details about calling DLLs.

DOS_BOX_CHARS

This variable allows you to redefine the line drawing characters used with the Windows console (DOS-box) runtime. The value of DOS_BOX_CHARS is a list of characters that draw the line segments. It should be a list of 13 space-delimited characters that correspond, in order, to the line segments as listed below. To redefine the DOS line drawing characters, specify the characters you want in the following order:

1. horizontal line
2. vertical line
3. upper left corner
4. upper right corner
5. lower left corner
6. lower right corner

Four three-way intersections -

7. missing bottom line
8. missing left line
9. missing top line
10. missing right line

and -

11. the four-way intersection
12. upper-half block
13. lower-half block

These line drawing characters may also be specified by decimal value. Characters that are not available on a particular machine should be specified with the decimal value “0”.

The default value for DOS_BOX_CHARS depends on the CODE_SYSTEM configuration variable. If CODE_SYSTEM is not set, or is set to “0” (or ASCII or EBCDIC), the default is:

```
DOS_BOX_CHARS 196 179 218 191 192 217 193 195 194 180 197 223 220
```

If CODE_SYSTEM is set to a non-zero value, which is the case in the ACUCOBOL-GT JPN version, the default is:

```
DOS_BOX_CHARS 6 5 1 2 3 4 21 25 22 23 16 0 0
```

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

DOS_SYS_EMULATE

When set to “1” (on, true, yes), this variable causes a program running in the Windows console runtime to run as if it’s in a DOS environment. One of its effects is that it prevents such a program from attempting to display GUI screens. It is set to “0” (off, false, no) by default. This variable has meaning only in Windows environments.

DOUBLE_CLICK_TIME

This variable has meaning only on systems that support a mouse. It controls the “double-click” rate on systems that do not control it themselves.

Specify the maximum time (in hundredths of a second) allowed between two clicks that are to be interpreted as a double-click. For example, if DOUBLE_CLICK_TIME were set to “75” (three-quarters of a second), then any two clicks that occur at least that close together would be considered a double-click rather than two single clicks.

The default value is “50” (one-half of a second).

DUPLICATES_LOG

This variable is used during bulk addition of Vision files. It causes Vision to write files rejected for having illegal duplicate keys to a log file. Set DUPLICATES_LOG to the name of a file in which to store the records. If this log file already exists, it is overwritten. You should 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, as follows:

```
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 no duplicate records are found, the log file is removed when the Vision file is closed. If DUPLICATES_LOG has not been set, or is set to spaces, no log file is created.

Note: 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.

See section 6.1.6.3, “Bulk Addition Mode for Vision,” in Book 1, *ACUCOBOL-GT User’s Guide,* for instructions on how to read the log file.

DYNAMIC_FUNCTION_CALLS

This variable allows you to specify a list of functions or function name prefixes that the runtime treats as dynamic functions and therefore searches first, before searching the disk for COBOL programs. This speeds the resolution of calls to functions in the current process or in a shared library.

The runtime checks call names for matches in the list specified in the variable. If a match is found, the runtime attempts to call the routine directly in the current process and in each of the loaded shared libraries. If these attempts fail, the runtime attempts to load a COBOL program with the specified name.

Set DYNAMIC_FUNCTION_CALLS to a space- or comma-delimited list of names of frequently called functions that are linked into the current process or in one of the loaded shared libraries.

The asterisk "*" character can be appended to the end of a name as a wild card. In this case, the characters before the asterisk are treated as a prefix and match any call name that begins with that prefix. A value of asterisk ("*") alone matches all function names. Use this to cause the runtime to treat all names as dynamic functions first before searching the disk or memory for a COBOL program with a matching name.

The value of DYNAMIC_FUNCTION_CALLS is case insensitive. The default value is empty.

DYNAMIC_FUNCTION_CALLS can be set in the environment, configuration file, or programmatically with the SET verb. Set it to spaces to clear the list.

When DYNAMIC_FUNCTION_CALLS is set in the configuration file, there is no limit on the number of function names or overall size of the value of the configuration variable. To specify a configuration file value on multiple lines, you must prepend each line after the first with "-". For example:

```
DYNAMIC_FUNCTION_CALLS =
-                          func1,
-                          func2,
-                          func3
```

The line continuation processing removes all leading and trailing spaces so in this case you must separate the values with a comma (that is, append a comma to each line).

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

DYNAMIC_MEMORY_LIMIT

The value of this variable indicates the maximum number of bytes of dynamic memory that the ACUCOBOL-GT runtime will use to cache canceled programs when the logical cancel mechanism is enabled. When the total amount of memory exceeds the value of DYNAMIC_MEMORY_LIMIT, the runtime releases all memory held by programs that have been logically canceled.

Valid values are:

-1	(the default) no memory limit. In transaction processing systems, memory used by programs that have been logically canceled is released only by the CICS transaction, CEMT SET PROGRAM(*program_name*) NEWCOPY
0	all cancels are physical; program memory is not cached
1 to 2147483647	the maximum number of bytes of dynamic memory

A discussion of memory management and physical and logical cancels is located in section 6.3, “Memory Management,” in Book 1. DYNAMIC_MEMORY_LIMIT is used in conjunction with the LOGICAL_CANCELS configuration variable. See its entry in this appendix.

ECN-3699

This variable relates to read-only entry fields. As of Version 8.1, read-only entry fields were changed to conform to standard Windows behavior in that the background color is always gray (regardless of the COBOL program's Color setting). If you need the ability to change the color of read-only entry fields, set the runtime configuration variable "ECN_3699" to "0".

The default behavior will not allow color changes and matches Windows behavior: read-only entry fields have gray backgrounds.

EDIT_MODE

This is an obsolete entry that has been replaced by the KEYSTROKE configuration variable. Its setting is ignored.

EF_UPPER_WIDE

This variable determines which font measure is used to compute the width of an entry field with the UPPER style. If the value is “1” (on, true, yes), the entry field is sized with the wide font measure. See section 5.9 in Book 2, *ACUCOBOL-GT User Interface Programming* for a description of how entry fields are measured. The default value of EF_UPPER_WIDE is “1”.

EF_WIDE_SIZE

This variable sets the boundary size that determines whether an entry field is sized with the standard or wide font measure. An entry field that has a specified width greater than the value of EF_WIDE_SIZE is always sized with the standard font measure. Entry fields that are both non-numeric and not larger than EF_WIDE_SIZE are sized with the wide font measure. See section 5.9 in Book 2, *ACUCOBOL-GT User Interface Programming* for a description of how entry fields are sized. The default value of EF_WIDE_SIZE is “5”. Setting this variable to “0” causes all entry fields to be sized with the standard font measure (exception: see EF_UPPER_WIDE above). Note that setting the value of this variable to a number larger than your largest entry field causes all entry fields to use the wide font measure.

EOF_ABORTS

This configuration variable can be used to handle two unexpected loop conditions:

1. a loop that results when the runtime has been started with “-i” and an input file terminates prematurely.
2. a loop that results when a terminal emulator disconnects unexpectedly.

If the runtime is started with the “-i” option and a loop occurs when an input file terminates prematurely, you can set EOF_ABORTS to “1” (on, true, yes) to cause the runtime to shut down when an ACCEPT statement detects an end-of-file condition.

On UNIX/Linux systems, if the runtime enters a loop due to an unexpected disconnect from a terminal emulator, you can set EOF_ABORTS to a value of “2” to cause the runtime to generate a hangup signal (SIGHUP) when it detects an EOF on standard input (stdin).

The default value is “0” (off, false, no).

EOL_CHAR

This configuration variable determines the character that is used to mark the end of each line when a pre-existing line sequential file is read. This should be set to the ASCII value of the desired character. The default value is “10” (line-feed). This option may be useful if you must process a line-oriented file that has an unusual line terminator. This configuration variable has no effect under VMS (RMS does not support it).

ERRORS_OK

Normally, if a file error occurs and there is no AT END, INVALID KEY, or Declarative statement to handle it, the runtime system prints an error message and halts. You can cause the runtime to ignore file errors and continue processing by setting ERRORS_OK to either “1” (on, true, yes) or “2” (FILESTATUS).

By default, ERRORS_OK is set to “0” (off, false, no).

When ERRORS_OK is set to “1”, if a file error occurs the runtime continues as if no error occurred.

When ERRORS_OK is set to “2”, if a file error occurs and there are no Declaratives but a file status variable is defined, the runtime ignores the error and continues processing. However, if a file error occurs and there are no Declaratives and a file status variable is *not* defined, the runtime halts.

Note: In general, it is not recommended that you configure the runtime to ignore file errors.

EXIT_CURSOR

When a STOP RUN is executed, the ACUCOBOL-GT runtime system normally places the cursor on the last line of the screen and then scrolls the screen one line. This allows the operating system prompt to appear on a new blank line at the bottom of the screen. To inhibit this behavior, set EXIT_CURSOR to “0” (off, false, no). This causes the runtime system to leave the cursor in its current location when the program exits. The default value “1” (on, true, yes) causes the standard ACUCOBOL-GT cursor positioning.

This variable has no effect on Windows systems.

EXPAND_ENV_VARS

Setting this variable to “1” (on, true, yes) causes the runtime to expand environment variables in filename specifications. This is the last step of file name interpretation process (see section 2.8, “File Name Interpretation,” in Book 1, *ACUCOBOL-GT User’s Guide*). 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 example, if the program attempts to open “$mydir/myfile”, the environment and configuration file are searched for the variable “mydir”. If found, its value is substituted. If not found, the replacement is null. Referring to the preceding example, if “mydir” is not defined, the runtime attempts to open “/myfile”.

The default value is “0” (off, false, no).

Note: The “$” character is a valid filename character in many file systems, including NTFS and most UNIX file systems. If you want to use dollar signs in your file names, you should not enable this option. In particular, if a user chooses the name of the file, you should keep this option disabled.

If you also use **FILE_ALIAS_PREFIX**, note that when EXPAND_ENV_VARS is set to “1”, FILE_ALIAS_PREFIX treats “$FILE1” and “FILE1” the same.

EXTEND_CREATES

Setting this configuration variable to “1” (on, true, yes), causes OPEN EXTEND statements to create a new file when the file being opened is not present. The default value is “0” (off, false, no).

EXTFH_KEEP_TRAILING_SPACES

An EXTFH_KEEP_TRAILING_SPACES configuration variable allows you to preserve trailing spaces in line sequential file records when using our EXTFH module with EXTSM. Set this variable to “1” (on, true, yes) to retain the trailing spaces, which is the runtime’s default behavior. With a default value of “0” (off, false, no), trailing spaces are removed.

Note that a related configuration variable is the **STRIP_TRAILING_SPACES** variable.

EXTERNAL_SIZE

ACUCOBOL-GT manages external data items by allocating them in *pools*. The minimum size of each pool is set by the EXTERNAL_SIZE variable. When a new external data item is needed, it is allocated from an existing pool. If it doesn’t 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 configuration variable. Using this larger pool reduces memory fragmentation. Because external data items remain allocated after programs are canceled, it's best to allocate the external data items together so they don't break up the memory space. The default value for EXTERNAL_SIZE is "8192". The maximum value is "32767".

EXTRA_KEYS_OK

This configuration variable allows you to open an indexed file without specifying all of that file's alternate keys. When it is set to "1" (on, true, yes), you may open an indexed file that contains more keys than are described by your program, and no file error will occur. However, you will still receive a file error if you open a file that does not contain all of the keys described in your program. EXTRA_KEYS_OK is useful when you are adding new alternate keys to an existing file because you do not need to rework your existing programs. This configuration variable is ignored if you use a Version 1.4 or earlier ACUCOBOL-85 object file.

The default value is "0" (off, false, no).

F10_IS_MENU

By convention, the F10 key is used by Windows and Windows NT to activate program menus. This action is controlled automatically by the program. The F10_IS_MENU configuration variable allows you to set the runtime to handle the F10 key as a user defined-key. The default setting is "1" (on, true, yes). When you change the setting to "0" (off, false, no) you inhibit the menu activation capability. For example, action of Shift-Ctl-F10 may only be defined by the user if F10_IS_MENU is set to "0", otherwise this key combination activates context menus. This variable does not affect the behavior of the mouse. However, the mouse continues to work with the menu.

FAST_ESCAPE

This configuration variable determines how long the runtime will wait after receiving an escape key before deciding that the key is actually intended as an escape key, and *not* as the start of a function key sequence. (Increasing the number causes the runtime to wait longer.) The default setting varies with the machine. It is generally between 20 and 100.

This variable has no effect on Windows systems.

FAST_SIGN_DECODE

Version 7.3 introduced a new algorithm to decode sign bytes in USAGE DISPLAY (sign incorporated) data items. The new algorithm is 3-4 times faster than the previous one, but produces unexpected results for undefined or non-initialized numeric data items.

This configuration variable turns on this faster algorithm introduced in Version 7.3.

To use the Version 7.3 algorithm, set the FAST_SIGN_DECODE configuration variable to "TRUE". If using this algorithm, numeric data items must be defined/initialized, otherwise incorrect results may occur.

The default value is "FALSE" which means the pre-7.3 version of the algorithm is used. The pre-7.3 version does not require numeric data items to be defined/initialized.

FIELDS_UNBOXED

On most GUI systems, including Microsoft Windows, entry fields are boxed by default. This can cause problems when you are converting applications that have fairly full screen displays, because the box adds roughly 50% to the height of the field. This can make it difficult to fit all the existing fields onto the user’s screen.

FIELDS_UNBOXED provides a global method of removing boxes on entry fields. If this field is set to “1” (on, true, yes), the system does not display a box around entry fields. Technically this has three effects:

1. If it is set when the entry field is initially created, the NO-BOX property is automatically implied.
2. If it is set when a floating window is initially created, the window’s LABEL-OFFSET property is given a default value of “0”.
3. When an entry field is measured by the CELL phrase of the DISPLAY FLOATING WINDOW statement, its height is measured without the box.

On character-based systems, setting this variable to “1” (on, true, yes) eliminates the display of the left and right delimiting symbols used in the textual emulation of entry fields. (See GUI_CHARS for more information about these delimiting symbols). Eliminating these symbols affects the location at which the entry fields are displayed on character-based systems.

The default value for this option is “0” (off, false, no).

This variable can be overridden for individual entry fields in the program with the BOXED style in the entry field definition.

FILE_ALIAS_PREFIX

This variable allows you to specify a list of strings to prefix to a file name before searching for that name in the configuration file or environment. Data and code file search paths are described in more detail in section 2.7.2 of the *ACUCOBOL-GT User’s Guide*.

When searching for a file alias:

1. The runtime constructs the file alias name by prepending the first string listed in FILE_ALIAS_PREFIX to the file name and searches for that name in the environment or configuration file.
2. If the name is not found, the runtime constructs a new name by prepending the second string in FILE_ALIAS_PREFIX to the file name and searches for that alias.

This 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.

For example, with:

```
SELECT file1-name ASSIGN TO "FILE1".
```

by default, the runtime looks for a configuration or environment variable named “FILE1” and, if found, substitutes its value for the file name. If you specify:

```
FILE_ALIAS_PREFIX "":DD_
```

the runtime first looks for “FILE1” and, if not found, looks for “DD_FILE1”.

The default value of FILE_ALIAS_PREFIX is an empty string (“”). Specifying an empty string as an entry in FILE_ALIAS_PREFIX causes the runtime to search for the file name itself as an alias name. Up to 4096 characters can be specified for the value of this variable.

Note: Separate strings by one or more spaces. A space is a valid separator on all systems. On UNIX systems, you can also separate entries with a colon. On Windows systems, a semicolon can be used and on VMS systems, a comma can be used. Strings can be enclosed in quotation marks. You can specify an empty string using two consecutive quotation marks.

Note on using with EXPAND_ENV_VARS:

If you use the **EXPAND_ENV_VARS** configuration variable and the file name includes a dollar sign ($), the FILE_ALIAS_PREFIX logic is applied to the environment variable name. For example, if EXPAND_ENV_VARS is set to”1” (on, true, yes), “$FILE1” and “FILE1” are treated the same.

For example, with:

```
EXPAND_ENV_VARS=1
FILE_ALIAS_PREFIX=DD_
```

the following statement,

```
SELECT file1-name ASSIGN TO "DIR1/$DIR2/FILE1".
```

causes the runtime to search for an environment or configuration variable named “DD_DIR2” (instead of “DIR2”) and, if found, substitute its value for “$DIR2”.

FILE_CASE

This configuration variable allows you to adjust the case of data file names. Possible values include:

NONE or “0”	(the default) data file names are not translated
LOWER or “1”	data file names are translated to lower case, including directory (path) elements
UPPER or “2”	data file names are translated to upper case, including directory (path) elements
LOWER_BASE or “3”	data file names are translated to lower case, excluding directory (path) elements
UPPER_BASE or “4”	data file names are translated to upper case, excluding directory (path) elements

Translation occurs before the FILE_PREFIX and FILE_SUFFIX configuration options are applied. You should make sure that those variables specify the correct case.

File name translation does not occur if the file name starts with -F, -D, or -P. (See *ACUCOBOL-GT User’s Guide*, section 2.8, “File Name Interpretation.”)

FILE_CONDITION

This configuration variable can be used to alter the File Status value of an individual file status condition. We recommend that you use one of the four pre-defined file status code sets instead. If you need to change an individual status code, contact our Technical Support for assistance.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

FILE_IO_PEEKS_MESSAGES

This configuration variable tells the Windows runtime to automatically call the Windows PeekMessage() API function between file operations. When the FILE_IO_PEEKS_MESSAGES configuration variable is set to "1" (on, true, yes), the runtime calls PeekMessage() with flags that tell it to simply check for messages without removing them from the message queue. This operation tells Windows that the application is alive and responding. The default value of the variable is "0" (off, false, no).

FILE_IO_PROCESSES_MESSAGES

This configuration variable can be used to control whether the runtime processes system messages while performing file I/O operations. When it is set to "1" (on, true, yes), the runtime will process system messages while doing file I/O operations. This was the default behavior *prior* to Version 3.2. Note that the processing of system messages during file I/O should only be enabled under special conditions, as described below.

To understand when it is appropriate to set this configuration variable, it is important to be familiar with *system messages* and how the ACUCOBOL-GT runtime and your program respond to them. For the purposes of this discussion, system messages are the mechanism used by graphical systems, such as Windows, to communicate with your program. They are what the operating system uses to facilitate the communication of user and system activity to the program. They are similar to ACUCOBOL-GT's *events*. Prior to Version 3.2, the runtime automatically processed system events during file operations. This allows the user to manipulate an application window (for example, minimizing it) while file I/O operations are performed. If the application suspends the processing of system messages, the system appears to the user to be frozen.

Starting with Version 3.2, this feature is turned off by default. This is because the processing of messages outside of an ACCEPT statement can cause flaws in a program that uses multithreading or modeless windows. It also creates a state where event procedures can be called at unexpected times. In addition, the controls of the application are not actually functional, though they appear to be working to the user.

Generally speaking, setting this variable is useful only when the application does not use multithreading, modeless windows, or event procedures.

Note: The proper way to process system messages while performing other operations is to start a second thread that performs an ACCEPT statement while the main thread continues with the work. This allows the system to process messages under control of an ACCEPT, which provides a well-defined point in your program from which event procedures can be called.

FILE_PREFIX

This variable defines a set of directories that the runtime searches to locate a data file. The default value is "." (current working directory). Data and code file search paths are described in more detail in section 2.7.2 of the *ACUCOBOL-GT User's Guide*.

Directories can be a mix of relative and absolute paths. Entries are separated by one or more spaces. A space is a valid separator on all systems. Alternatively, on UNIX systems you can also separate entries with a colon. On Windows systems a semicolon can be used. On VMS systems a comma can be used.

You can specify a directory path that contains embedded spaces if you surround the path with quotation marks. For example:

```
FILE_PREFIX  C:\“Sales Data”  C:\“Customers”
```

Remote name notation is allowed for the FILE_PREFIX variable if your runtime is client-enabled (for indexed files, remote name notation requires the Vision file system). See *ACUCOBOL-GT User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

Up to 4096 characters can be specified for the value of this variable.

FILE_STATUS_CODES

This variable determines which set of file status codes to use. For details, see the *ACUCOBOL-GT User's Guide,* section 2.7.3, "File Status Codes." The default value is "85".

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

FILE_SUFFIX

The value of this variable is automatically appended to data file names that do not contain an explicit suffix. A suffix is the portion of a file name that follows a period. For example, if FILE_SUFFIX is set to "DAT", then opening a file called "EMPFILE" would actually open the file called "EMPFILE.DAT". The default value is empty.

FILE_TRACE

This variable allows you to start file tracing without opening the debugger. Set this variable to a non-zero value to save information about all file OPENs, READs, and WRITES in the error file. This is equivalent to specifying "tf *n*" from the debugger (where *n* is an integer). The default is "0." See section 3.1.4 of the *ACUCOBOL-GT User's Guide* for more information about the file trace feature.

FILE_TRACE_FLUSH

Set this variable to “1” (on, true, yes) to flush the error file after every WRITE statement. This is equivalent to using “t flush” from the debugger. The default is “0” (off, false, no). See section 3.1.4 of the *ACUCOBOL-GT User’s Guide* for more information about the file trace feature.

FILE_TRACE_TIMESTAMP

Set this variable to “1” (on, true, yes) to cause file trace timestamp information to be recorded in the error file. When this variable is enabled, a timestamp is placed at the beginning of every line in the trace file. The format of the timestamp is: HH:MM:SS.mmmmmm, where “mmmmmm” is the finest resolution that the runtime can obtain from the system. The default setting of this variable is “0” (off, false, no).

Timestamp information is included only when file trace information is directed to a file. Timestamp output can add significant I/O overhead and may have a noticeable impact on performance.

filename

This configuration variable allows you to map Vision 4 and 5 files to a different directory. Vision examines the name of each physical file it attempts to open to determine if the file should be mapped to a different directory. The configuration variable used is constructed from the file’s base name and extension, with all letters converted to upper case and all non-alphanumeric characters converted to underscores.

For example, assume you open “/usr/data/custfile.dat”, and a configuration variable “CUSTFILE_DAT” has the value “/usr2/data/custfile.dat”. Vision treats this value as the actual file name to open, and “custfile.dat” ends up in the “/usr2/data” directory rather than “/usr/data”.

Because the extension is included in the configuration variable name, you can place different parts of a multi-segment file in different directories. If no name is found for a particular segment, then the segment name is used unchanged. Note that you can move parts of a file around by simply moving the segment and adding/modifying its corresponding configuration name. Name mapping is done directly by Vision (as opposed to, for example, FILE_PREFIX, which is handled by the runtime). As a result, all programs that use Vision (such as **vutil** and **vio**) use this variable when present. For programs other than the runtime, the variable must be set in the environment rather than the configuration file.

Two configuration variables can affect the value of this variable. They are: V_BASENAME_TRANSLATION and V_STRIP_DOT_EXTENSION. See their respective entries in this appendix for details.

This variable is similar to the *filename*_VERSION configuration variable described later in this appendix.

Note: The filename translation performed by this configuration variable is performed by Vision itself. The runtime can also perform filename translation. See Book 1, *ACUCOBOL-GT User's Guide*, section 2.7.1 for more information.

---

*filename*_DATA_FMT

This configuration variable specifies a format for naming the data segments of Vision 4 and 5 files. (See **filename_INDEX_FMT** for details about naming the index segments, as both variables should be set to corresponding patterns). The configuration variable used is constructed from the file's base name and extension, with all letters converted to upper case and all non-alphanumeric characters converted to underscores, followed by the "_DATA_FMT" string. Note that by design, this variable does not modify the first specified data segment. The first data segment retains the originally specified name. The filenames of the additional segments of a Vision file are generated from the name of the initial data segment. The *filename*_DATA_FMT variable allows you to change the way the names of the following data segments are formed, but the names still originate from the name of the initial data segment. As long as the names are as expected (and you have set filename_DATA_FMT and *filename*_INDEX_FMT accordingly) the segments will be found properly.

Suppose that the regular name of your COBOL file is “/usr1/gl.dat”. The variable you would use to set the data segment naming format for this file is GL_DAT_DATA_FMT.

The variable 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 extension for each segment. Part of this pattern is a special escape sequence (such as %d) that specifies how the segment number should be represented. Choices include %d (decimal segment numbers), %x (lowercase hexadecimal numbers), %X (uppercase hexadecimal numbers), and %o (octal numbers).

For example, setting the variable GL_DAT_DATA_FMT=gl%d.dat 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.

Escape sequence definitions:

The %d in the value of the *filename*_DATA_FMT 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: While the runtime checks for this segment naming variable in the runtime configuration file as well as in the environment, utilities such as **vutil** and **vio** check only the environment. Therefore, if you are using this variable with the runtime and **vio** or **vutil**, you must set the variable in the environment and not in the configuration file.

Two configuration variables affect the value of this variable: V_BASENAME_TRANSLATION and V_STRIP_DOT_EXTENSION. See their respective entries in this appendix for details.

Note: The filename translation performed by this configuration variable is performed by Vision itself. The runtime can also perform filename translation. See Book 1, *ACUCOBOL-GT User's Guide*, section 2.7.1 for more information.

---

*filename*_FILESYSTEM

*filename*_FILESYSTEM is a synonym for *filename*_HOST. See the entry for **filename_HOST**.

*filename*_HOST

Note: *filename*_FILESYSTEM is a synonym for *filename*_HOST.

If the program opens an existing file or creates a new one, you can tell the runtime the file system to use with that file. The Vision file interface is used by default. You specify the file system with one of two configuration variables: DEFAULT_FILESYSTEM, or *filename*_HOST (syn. *filename*_FILESYSTEM). DEFAULT_FILESYSTEM specifies the default file system for all files (see the entry for **DEFAULT_FILESYSTEM**). *filename*_HOST specifies the file system for an individual file. For example,

```
    filename_HOST filesystem
```

assigns the specified data file to the named file system. Any file so assigned uses the designated file system and not the one specified by DEFAULT_FILESYSTEM. *Filename* must be the base name of the file and cannot include the path or the file extension (any part of the name that follows the first dot (".")). For example, to specify that data file "IDX1.DAT" be handled by the EXTFH interface, you would specify:

```
IDX1_HOST EXTFH
```

or

```
IDX1_FILESYSTEM EXTFH
```

You must specify only the base name of the file in *filename*.

To specify that the data file "DXML1.DAT" be handled by the XML interface, you could specify:

```
DXML1_HOST XML
```

Note that XML can be specified only with sequential files.

*filename*_INDEX_FMT

This configuration variable specifies a format for naming the index segments of Vision 4 and 5 files. (See **filename_DATA_FMT** for details about naming the data segments, as both variables should be set to corresponding patterns). The configuration variable used is constructed from the file's base name and extension, with all letters converted to upper case and all non-alphanumeric characters converted to underscores, followed by the "_INDEX_FMT" string. Note that by design, this variable does not modify the first specified index segment. The first index segment retains the originally specified name. The filenames of the additional segments of a Vision file are generated from the name of the initial index segment. The *filename*_INDEX_FMT variable allows you to change the way the names of the following data segments are formed, but the names still originate from the name of the initial index segment. As long as the names are as expected (and you have set filename_DATA_FMT and *filename*_INDEX_FMT accordingly) the segments will be found properly.

Suppose that the regular name of your COBOL file is "/usr1/gl.dat". The variable you would use to set the format for naming the file's index segments is GL_DAT_INDEX_FMT.

The variable 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 (lowercase hexadecimal numbers), %X (uppercase hexadecimal numbers), and %o (octal numbers).

For example, setting the variable GL_DAT_INDEX_FMT=gl%d.idx would result in index segments named /usr1/gl0.idx, /usr1/gl1.idx, /usr1/gl2.idx, and so forth.

Escape sequence definitions:

The %d in the value of the *filename*_INDEX_FMT 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: While the runtime checks for this segment naming variable in the runtime configuration file as well as in the environment, utilities such as **vutil** and **vio** check only the environment. Therefore, if you are using this variable with the runtime and **vio** or **vutil**, you must set the variable in the environment and not in the configuration file.

Two configuration variables affect the value of this variable: V_BASENAME_TRANSLATION and V_STRIP_DOT_EXTENSION. See their respective entries in this appendix for details.

Note: The filename translation performed by this configuration variable is performed by Vision itself. The runtime can also perform filename translation. See Book 1, *ACUCOBOL-GT User's Guide*, section 2.7.1 for more information.

---

*filename*_LOG

This configuration variable specifies individual log files to be used by the transaction logging system. The format of the variable is:

```
        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 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 with the specified name. Note that *logfilename* can have remote name notation.

FILENAME_SPACES

When this configuration variable is set to "1" (on, true, yes), filenames may contain embedded spaces and the runtime considers the last non-space character as the end of the name. The default is "1". When this configuration variable is set to "0" (off, false, no), then filenames may not contain embedded spaces and the name terminates at the first space character. For example:

```
C:\temp dir\my file name
```

is read as:

```
C:\temp
```

This affects the behavior of the library routines that take a filename as an argument:

C$CHDIR
C$COPY
C$DELETE
C$FILEINFO
C$FULLNAME
C$MAKEDIR
C$RESOURCE
CBL_COPY_FILE
CBL_CREATE_DIR
CBL_DELETE_DIR
CBL_DELETE_FILE
I$IO
RENAME
W$BITMAP
W$KEYBUF
$WINHELP
WIN$PLAYSOUND

*filename*_VERSION

This configuration variable sets the Vision file format on a file-by-file basis. The *filename* is replaced by the base name of the file (the filename minus directory and extension). The meaning of the variable is the same as for “V_VERSION”. This variable is useful if you want to have all your Vision files in one format, with a few exceptions. For example, you might want to maintain most of your files in Vision Version 3 format to conserve file handles, but have a few files in Version 5 format to take advantage of the larger file size. Note that this variable only affects the file format when it is created. You can always rebuild the file in another format later.

This variable (and the "V_VERSION" variable) is most helpful when you are using transaction management. The transaction system does not record the format of the created file if an OPEN OUTPUT is done during a transaction, because the transaction system is not tied to any particular file system. If you need to recover a transaction, the system will recreate the OPEN OUTPUT files using the settings of the "VERSION" variables.

The behavior of this variable is affected by the settings of the configuration variables V_STRIP_DOT_EXTENSION and V_BASENAME_TRANSLATION.

- If V_STRIP_DOT_EXTENSION is set to "0" (off, false, no), Vision does not remove any dot extension when replacing the base name of the file. This can be useful if you have two files that share a common name before their dot extension.

- If V_BASENAME_TRANSLATION is set to "0" (off, false, no), Vision includes the entire path of the file in the base name. This can be useful if you have files with the same names stored in different directories.

*filesystem*_DETACH

This configuration variable detaches any file system from the runtime. The syntax is:

```
filesystem_DETACH n
```

where *filesystem* corresponds to the first five letters of the file system name and *n* is a non-zero value. Examples of file systems that may be detached using this feature are:

Btrieve	BTRIE_DETACH
C-ISAM	C_ISA_DETACH
Informix	INFOR_DETACH
SQL Server	MSSQL_DETACH
ODBC	ODBC_DETACH
Oracle	ORACL_DETACH
RMS	RMS_DETACH

Sybase	SYBAS_DETACH
Vision	VISIO_DETACH

The file systems may be detached only when the runtime is started, not during execution. If you detach *all* file systems, the runtime will terminate with an error message. For example, if you detach Vision with VISIO_DETACH on a standard runtime, the runtime will terminate with this message to std err: No file system available.

This feature automatically supports new file systems added to the runtime.

FLUSH_ALL

This configuration variable can be used to control the flushing of file buffers to disk. It is one of several variables that control buffer flushing. See the other entries in this appendix that begin with "FLUSH".

This variable can take a combination of the following values:

1 (on, true, yes, all)
0 (off, false, no)
MASS_UPDATE
REMOTE

When this variable is set to "1", files opened for MASS-UPDATE are flushed along with other files. This means that the local cache used to hold the MASS-UPDATE buffers is flushed whenever the operating system cache is flushed.

When this variable is set to the default value of "0", files opened for MASS-UPDATE are not flushed.

Setting this variable to MASS_UPDATE causes the runtime to flush local files, including files opened with MASS-UPDATE.

Setting this variable to REMOTE causes the runtime to flush local files not opened with MASS-UPDATE, as well as remote files.

You can also set this variable to a combination of values. For example,

```
    FLUSH_ALL  MASS_UPDATE  REMOTE
```

causes the runtime to flush all local files, including those opened with MASS-UPDATE, as well as remote files.

Note on bitmask integer values:

Internally, the value of this configuration variable is converted to a bitmask, and its bitmask integer value is determined by the keywords used to set it. Keywords translate into integer values as follows:

FALSE, NO and OFF are equivalent to “0”

MASS_UPDATE is equivalent to “1”

REMOTE is equivalent to “2”

ALL, TRUE, YES and ON are equivalent to “–1”

When the runtime is started with the “-l” and “-e *errfile*” arguments, only the integer value of FLUSH_ALL is recorded in the error file.

FLUSH_COUNT

This configuration variable allows you to flush the disk buffers after a certain number of file updates has occurred. For example, if you set this configuration variable to “10”, then the buffers will be flushed after every ten updates to disk files. Only indexed files are counted. When the buffers are flushed, the exact action depends on the operating system:

Windows	Buffers are written to disk and the file’s directory information is updated. This is roughly equivalent to the action that occurs when a file is closed.
UNIX	The “sync” system routine is called. This causes all of UNIX’s cache to be written to disk. This operation is only scheduled--it occurs when the system finds time to do it. Because the system does this every 30 seconds anyway, probably the only reason to request a call to “sync” is if you have unreliable power.
VMS	VMS does not have a system cache, so this configuration variable has no effect.

Note: Setting this variable to a low non-zero value will improve the chances of recovering a file after a power failure, but will decrease performance. If FLUSH_COUNT is set to “0”, then the system buffers are flushed only when a file is closed. The default setting is “0”.

FLUSH_ON_ACCEPT

This configuration variable causes the system’s buffers to be flushed whenever a Format 1 or Format 2 ACCEPT statement is executed. You turn on this configuration variable by setting its value to “1” (on, true, yes). The default setting is “0” (off, false, no). Note that this variable is not recommended for multi-user systems because of the performance penalty.

FLUSH_ON_CLOSE

This configuration variable applies only to Windows systems. When this variable is set to “1” (on, true, yes), the cache buffers will be flushed to disk when the file is closed. Versions prior to 4.3.1 flushed the cache buffers for safety reasons. This, however, reduced system performance, significantly in some programs. This feature is turned “off” by default.

FLUSH_ON_COMMIT

When this configuration variable is set to “0” (off, false, no), the COMMIT verb will not request the host operating system to flush all buffers to disk.

If flushing is prevented, COMMIT and UNLOCK ALL have the same effect. The default setting is “0” (off, false, no).

FLUSH_ON_OPEN

This variable causes the system’s buffers to be flushed on the first WRITE, REWRITE, or DELETE that occurs after an indexed file is opened for I/O. The purpose of this is to update the “user count” field in the file’s header to keep it accurate. When this configuration variable is set to “1” (on, true, yes), this feature is turned on; when set to “0” (off, false, no), it’s turned off. The default setting is “0”.

FONT

This variable has meaning only on graphical systems such as Windows. You determine the font used for accepting and displaying data on screen by setting the configuration variable FONT to one of these values:

1	Use the graphical font (default). The character set for this font is referred to as the “ANSI” set.
2	Use the “OEM” character set (MS-DOS font).

This setting must be made in the configuration file before you execute the program. Altering the value of FONT from inside your program has no effect.

By default, data is stored on disk using the same character set that was used when the data was entered. Thus, if you use the graphical font to accept data, that data is stored in the ANSI character set. If you use the MS-DOS font, then data is stored in the OEM character set.

Data moved into a graphical environment from MS-DOS applications was originally stored in the OEM character set. What happens if you now choose the graphical font? As long as your application uses only standard ASCII characters, the underlying representation is the same, and so the data is completely interchangeable. See the variable TRANSLATE_TO_ANSI if you are using non-ASCII characters.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

FONT_AUTO_ADJUST

This variable allows you to disable an automatic font adjustment that is applied on Windows machines. The runtime attempts to adjust automatically for differences in the relative proportions between “small fonts” and “large fonts.” You can inhibit the adjustment by setting this variable to “0” (off, false, no).

The adjustment is provided because the internal scaling of fonts under Windows changes between “small” and “large” fonts. Under small fonts, digits are slightly wider than the average character. Under large fonts, digits have the same width as the average character. ACUCOBOL-GT uses the size of the font’s width for many calculations. Thus, the change in relative proportion within a single font can cause problems for screens designed for the small fonts. For example, a frame may not be big enough to hold its contents. To prevent this problem, the runtime ensures that the “standard font measure” is always a bit larger than the average character width in a font. To disable this adjustment, set this variable to “0”.

Note: This variable computes the width of a printer font in the same way that the width of a screen font is computed. You can suppress this behavior by compiling with the “-C43” option.

FONT_SIZE_ADJUST

This variable allows you to adjust the size of the standard font measurement that is computed for graphical controls (applies to variable-pitch fonts only). The value of FONT_SIZE_ADJUST is added directly to the computed standard font size. For example, a setting of “1” adds one pixel to the computed width of the font. Because the standard font measure is used to compute the width of nearly all controls, any adjustment made by this variable will have a significant impact on the layout of your screens.

The adjustment to the standard font measure is made after the wide font measure is computed (this is important to note because the wide font measure depends on the standard font measure; to change the wide font measure, use the FONT_WIDE_SIZE_ADJUST configuration variable).

After applying the adjustment, the runtime checks and ensures that the computed font measure is not less than one (1) or greater than the widest character in the font. If you find that the default size of most controls is slightly smaller than you prefer, you might try setting FONT_SIZE_ADJUST to a small value (typically 1).

Generally, it is recommended that FONT_SIZE_ADJUST (and FONT_WIDE_SIZE_ADJUST) be left at its default value of "0". You can also use negative values, but there is rarely a need to do so.

To optimize performance, the runtime computes the font sizes only occasionally. Although you can change the variable dynamically at runtime, the exact time when the new setting will take effect is difficult to predict. For this reason, we recommend that you either set it in your program prior to constructing your initial screen, or directly in the configuration file.

Note: This variable computes the width of a printer font in the same way that the width of a screen font is computed. You can suppress this behavior by compiling with the "-C43" option.

FONT_WIDE_SIZE_ADJUST

This variable allows you to adjust the size of the wide font measurement (applies to variable-pitch fonts only). The wide font measure is normally used when the runtime is measuring small or upper-case entry fields. The value of FONT_WIDE_SIZE_ADJUST is added directly to the computed wide font size.

After applying the adjustment, the runtime checks and ensures that the computed wide font measure is not smaller than the (adjusted) standard font measure or larger than the widest character in the font. If your upper-case fields are not quite as wide as you prefer, try setting this variable to a small value (typically 1 or 2).

Generally, it is recommended that FONT_WIDE_SIZE_ADJUST (and FONT_SIZE_ADJUST) be left at its default value of "0". You can also use negative values, but there is rarely a need to do so.

Note: In order to improve performance, the runtime computes the font sizes only occasionally. Although you can change the variable dynamically at runtime, the exact time when the new setting will take effect is difficult to predict. For this reason, we recommend that you either set it in your program prior to constructing your initial screen, or directly in the configuration file.

FOREGROUND_INTENSITY

Use this variable to set the default foreground intensity.

0	The runtime uses the default intensity for the output device. For Windows the default is low-intensity.
1	The runtime uses low-intensity.
2	The runtime uses high-intensity.

If your program specifies a default intensity, then the runtime will never assign high-intensity if the foreground is black. As with the background, we do this to prevent a washed-out appearance. There's one exception to this rule. The runtime will assign high-intensity to a black foreground if the output device does not support independent background intensities. In this case, the device will typically show the background in high-intensity and keep the foreground black. Note that if your program explicitly sets high-intensity, then that will be used regardless of the foreground color. The default value for this variable is "0".

FREEZE_AX_EVENTS

This configuration variable applies only in a thin client environment. During the processing of an ActiveX event, the Windows and thin client runtimes attempt to suspend subsequent ActiveX events until the first event has completed. By default, the thin client runtime also attempts to suspend ActiveX events whenever the application is not processing an ACCEPT statement. To suspend and resume events, the runtime calls the ActiveX function IOleControl::FreezeEvents().

You might want to disable calls to “FreezeEvents” for ActiveX controls that discard events while in a “FreezeEvents” state. For example, if a user double-clicks in an ActiveX control, the control might generate three events: mouse-down, mouse-up, and double-click. If the COBOL program terminates an ACCEPT statement in response to the mouse-down event, the runtime calls FreezeEvents(), and the ActiveX control might discard the mouse-up and double-click events.

You can disable the FreezeEvents() logic by setting the FREEZE_AX_EVENTS runtime configuration variable to “0” (off, false, no) in the configuration file or programmatically with the SET verb. The default value of FREEZE_AX_EVENTS is “1” (on, true, yes).

Note: The FreezeEvents() logic protects against unexpected nesting of ActiveX events and against event procedures running unexpectedly during a CREATE, DISPLAY, MODIFY, INQUIRE, or other operation that waits for results from the thin client. Turning this feature off can cause unexpected behavior.

For more information about ActiveX controls in a thin client environment, refer to the *AcuConnect User’s Guide*.

FULL_BOXES

This variable applies only in text-mode environments. When FULL_BOXES is set to “1” (on, true, yes) a full, four-sided box is drawn around *boxed* entry fields. By default, to save screen space on character-based systems, only the left and right edges of a box are drawn around boxed entry fields. The default value of FULL_BOXES is “0” (off, false, no).

Note: This variable requires that the boxed entry field be defined as MULTILINE.

GRID_BUTTONS_CAUSE_GOTO

This variable applies to graphical programs that include one or more paged grid controls. When GRID_BUTTONS_CAUSE_GOTO is set to “1” (on, true, yes) and a user clicks a scroll button on the side of a paged grid, the runtime checks to see if the grid has focus. If the grid does not have focus, the runtime gives the grid the focus, generating any events that result normally from that focus change. This usually means that a CMD-GOTO event is sent to the COBOL program. The default value for this variable is “0” (off, false, no).

GRID_NO_CELL_DRAG

This variable applies to graphical programs that include grid controls and sets the NO-CELL-DRAG style as the default behavior for all grid controls, as opposed to specifying NO-CELL-DRAG style individually for each grid control. To configure the NO-CELL-DRAG style as the default setting, set the GRID_NO_CELL_DRAG configuration variable to “1” (on, true, yes). The default value is “0” (off, false, no) and will enable the user to drag a cell in a GRID control unless you specify a NO-CELL-DRAG style on that particular grid control.

GUI_CHARS

This variable configures a character-based host runtime to substitute some specific characters when it is performing textual emulation of graphical controls. If the GF-GUI-MAP terminal database entry doesn’t exist or has the character “0” (zero) in a particular character’s position, the runtime examines the GUI_CHARS variable to determine what character to display. GUI_CHARS is used only by character-based host runtimes.

The value of GUI_CHARS is a list of 14 space delimited characters strictly ordered to correspond with the following graphical elements (defaults are in parentheses):

1.	System menu button	(*)
2.	Title left corner	(+)
3.	Title right corner	(+)

4.	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 slider	(#)
13.	Left entry field box	([)
14.	Right entry field box	(])

The characters may be specified in ASCII or decimal form. To specify an ASCII value, precede the value with a space. For example, to use "-" in place of "=" for the title fill character, make the following entry in the runtime configuration file:

```
GUI_CHARS  0 0 0 - 0 0 0 0 0 0 0 0 0 0
```

or

```
GUI_CHARS  0 0 0 -
```

The presence of a "0" (zero) following a space, causes the default character to be used.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

HELP_PROGRAM

If the program uses help automation (see Book 2, *ACUCOBOL-GT User Interface Programming*, Chapter 10), this variable should be assigned the name of the help processor program. The help processor's entry point is always a COBOL program. The program can be the help processor itself, or a shell to some other help processor, such as Windows Help. The default value of HELP_PROGRAM is "AcuHelp".

HINTS_OFF

Controls how long the pop-up hint is displayed before being erased. See section 3.7.4 in Book 2, *ACUCOBOL-GT User Interface Programming* for a description of pop-up hints. Set this variable to the number of hundredths of seconds to display the hint. The default value is "400" (four seconds). If you set this value to "0", the hint will remain displayed until some other event (such as using the button or typing) causes it to disappear.

HINTS_ON

Controls how long the mouse must remain stationary over a bitmap button before displaying its pop-up hint. For a description of pop-up hints, see section 3.7.4 in Book 2, *ACUCOBOL-GT User Interface Programming.* Set this variable to the number of hundredths of a second that the mouse must be stationary. The default value is "75" (three-quarters of a second). If you set this variable to "0", pop-up hints will not be displayed. Setting this variable to "1" or "2" is not recommended because it may result in the pop-up hint being displayed while the mouse is moving across the button face (because the mouse motion events may occur less than 0.02 seconds apart).

HOT_KEY

ACUCOBOL-GT offers two methods for assigning hot keys--the HOT_KEY variable, described here, and the KEYSTROKE hot-key format described in the *ACUCOBOL-GT User's Guide,* section 4.3.2.2.

Using the HOT_KEY variable described below, you can easily assign a whole range of keys to a single hot-key program and determine which key activated the program. This lets you write a single program that handles an entire menu. Each menu item can act as a "hot key" to call this program.

This HOT_KEY format differs from the KEYSTROKE hot-key described in the *User's Guide* in three ways:

- You assign a hot key by referencing its *exception value* instead of referencing its *key code*. Thus, if you assign the same exception value to several individual keys, you can associate these keys with the same hot-key program by making one COBOL configuration file entry.

Similarly, menu items and individual keys can be assigned the same exception value, and then associated with the same hot-key program in a single configuration file entry.

- You may assign a range of exception values to activate the same program. You could use this to write a menu handler by assigning all of your menu items to a unique range and then assigning that range to a single hot-key program.
- A hot-key program activated using the HOT_KEY format is passed an additional parameter. This third parameter contains the value of the exception key that activated the program. This is passed as a COMP-1 data item.

Use this variable to associate an exception value, or range of values, with a program. HOT_KEY has the following format:

```
HOT_KEY  program = value1 [, value2]
```

where program is the name of the program to run, value1 is the lower (or only) exception value that activates the program, and value2 is the upper value of the activation range. Value2 may be omitted; if it's used it must include the separating comma. You must place program in single or double quotes if you require a lower-case program name.

For example, to assign a program called "mymenu" to exception values 100 through 200, use the following entry:

```
HOT_KEY  "mymenu" = 100, 200
```

A special exception value named TIMEOUT may be specified as the first exception value. When this value is used as the first exception value for a HOT_KEY program, the runtime will execute the named program whenever an ACCEPT BEFORE TIME times out. When that occurs, the second exception value is ignored.

Remote name notation is allowed for the HOT_KEY variable if your runtime is client-enabled. See *ACUCOBOL-GT User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

Multiple HOT_KEY entries may reference the same program. This allows you to specify noncontiguous activation ranges. (Be aware that no more than 16 hot-key entries can be included in the COBOL configuration file. Using a contiguous range of exception values assigns many keys while counting as only one entry towards the limit.)

If you specify a *value1* value of “0”, then all hot-key references to *program* are removed. Within a given run unit, this is the only way to remove the assignment of an exception value to a hot-key program after it has been assigned. You will probably use SET ENVIRONMENT in your source code to do this.

If you assign multiple hot-key programs to the same exception value, the results are undefined.

You may assign different hot keys using both the HOT_KEY variable, described here, and the KEYSTROKE hot-key format described in the *ACUCOBOL-GT User’s Guide*, section 4.3.2.2. 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.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

HP_TERMINAL_ATTRIBUTE_HANDLING

When set to “1” (on, true, yes), if the previous character was written to a line other than the current one this variable causes the runtime to set the attribute for the character to be written even if the attribute has not changed. When set to “0” (off, false, no), this variable causes the runtime to behave as it always has. For example, if the terminal attribute has not changed then it will not be set again. The default setting is “0” (off, false, no).

HTML_TEMPLATE_PREFIX

This variable is used to specify a series of directories for locating HTML template files. This variable is similar to FILE_PREFIX and CODE_PREFIX. It specifies a series of one or more directories to be searched for the desired HTML template file. 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). This is the default.

Note: Remote name notation is *not* allowed for the HTML_TEMPLATE_PREFIX variable, even if your runtime is client-enabled.

ICOBOL_FILE_SEMANTICS

This variable affects the behavior of indexed and relative files when reversing direction after reading past the beginning or end of a file. Normally, if you perform a series of READ NEXTs that reach to the end of the file (returning file status “10”), a subsequent READ PREVIOUS will return the last record in the file. The file pointer’s position after each READ NEXT is just past the end of the last record. Similarly, reading past the beginning of the file and then doing a READ NEXT will return the first record in the file.

Under ICOBOL, these conditions produce different results. The record returned by the READ PREVIOUS is the second-to-last record in the file, and the record returned by the READ NEXT is the second record in the file. Essentially, when a series of READs passes either end of the file, the record pointer remains on the first or last record.

Setting ICOBOL_FILE_SEMANTICS to “1” (on, true, yes) will cause the runtime to emulate ICOBOL’s handling. This is useful when porting ICOBOL programs to ACUCOBOL-GT. This option is effective only in programs that have been compiled for ACUCOBOL-85 2.0, or later. The default value is “0” (off, false, no).

ICON

This variable has meaning only on graphical systems such as Windows. Use this variable to designate a program's *minimized* icon. (By default, it uses the non-debugger icon provided with ACUCOBOL-GT.) Set ICON to the name of the file that contains the icon. This file should be an ".ICO" file that contains a 16-color icon. (Although the file may contain icons in other formats, the runtime accesses only the 16-color icon.) The icon should be created using an icon editor like the one in the Windows Software Development Kit.

For example, if your custom icon were contained in the file "PAYROLL.ICO" in the directory \ACCT\ICONS, you would add the following line to your COBOL configuration file:

```
    ICON  \ACCT\ICONS\PAYROLL.ICO
```

The maximum icon size that can be used with the ICON variable is 32 x 32 bits. The runtime uses the first icon it finds in the icon file that fits its maximum. If that icon is larger than the maximum, a memory access violation may occur.

Note: The ICON configuration variable determines the icon used only when your application is minimized. It does not determine the icon displayed by the Program Manager.

IMPORT_USES_CELL_SIZE

This variable is used when importing graphical screens into the AcuBench Screen Designer using the screen import utility. IMPORT_USES_CELL_SIZE allows you to choose whether fields are measured using the actual cell size of the imported screen or measured in 10-pixel by 10-pixel cells. The runtime checks this variable only if you are importing screens. When IMPORT_USES_CELL_SIZE is set to the default value of "1" (on, true, yes), the screen import utility captures the actual cell size used to create windows. If this variable is set to "0" (off, false, no), the screen import utility outputs information based on 10-pixel by 10-pixel cells. Note that there is no need to set this variable when importing character screens, which should always import with a cell size of 10-pixel by 10-pixel cells. See the AcuBench documentation for more information on importing screens.

INACTIVE_BORDER_COLOR

This variable is used on character-based hosts to specify the color and video attributes of the characters that form the border (box) around an inactive floating window. INACTIVE_BORDER_COLOR can be set to a variety of numeric values that express combinations of color and video attributes. See the documentation for the COLOR phrase in the “Common Screen Options” section of the *ACUCOBOL-GT Reference Manual* (section 6.4.9).

If INACTIVE_BORDER_COLOR is set to “0”, the inactive window’s border is drawn with the colors and video attributes specified in the COBOL program when the window is first created. The default value is “0”.

INCLUDE_PGM_INFO

This variable causes additional program information to be added to the string passed to COBOL error procedures.

When INCLUDE_PGM_INFO is set to “1” (on, true, yes) the string passed to COBOL error procedures is prepended with the current program name and the address of the program failure. The address may not be exactly the same as that in the COBOL listing, but it will be very close. (The given address is the actual current program counter, which is typically slightly advanced from the line on which the fault occurred.) When INCLUDE_PGM_INFO is set to the default, “0” (off, false, no), the string contains only the text of the intermediate error message.

For more about COBOL error procedures, see the entry for the CBL_ERROR_PROC in Appendix I of the *ACUCOBOL-GT Appendices Manual*.

INPUT_STATUS_DEFAULT

The value of this variable is returned when an ACCEPT FROM INPUT STATUS statement is executed on a machine that cannot determine the input status of the terminal. This can be used to make a running program behave correctly on a new machine. The value must be a single digit. The default value is “0”.

If the input is redirected (not attached to a terminal), the **SCRIPT_STATUS** configuration variable determines whether ACCEPT FROM INPUT STATUS returns the value of INPUT_STATUS_DEFAULT or returns the actual status of the input file.

INSERT_MODE

This variable determines whether or not keystrokes are inserted in front of any existing text when the user types an entry. Set this variable to “1” (on, true, yes) to enable insertion. The default value is “0” (off, false, no), which causes typing to replace existing text.

The user can change the state of INSERT_MODE with various key actions. For example, pressing <insert> can enable insertion. This variable has no affect on Windows controls.

INTENSITY_FLAGS

This variable takes effect only if you use COLOR_TRANS and only if it changes your color scheme. After any color transformation is completed, the runtime system then transforms the foreground and background intensities according to the setting of INTENSITY_FLAGS. The value for this variable is actually the sum of the values you choose from the list below. The default value is “0”.

Set INTENSITY_FLAGS to a combination of the following options by adding their values together:

1	Exchanges the foreground and background intensities for each other. This is useful if you are swapping a black background into the foreground and want to assign the foreground’s intensity to the background.
2	Causes the foreground intensity to be inverted. That is, if the foreground is high-intensity, it becomes low-intensity. Otherwise, it becomes high-intensity. This is useful if you are transforming the background to white and the foreground to black. Setting this will cause your low-intensity foreground to be shown as gray while your high-intensity item will show as black.

4	Forces the foreground to high-intensity. This will not be applied to a black foreground.
8	Forces the foreground to low-intensity. This may not be used if “4” is used.
16	Causes the “4” or “8” setting to be used even if the COLOR_TRANS setting had no effect. This is an override switch that you can use to cause all foreground intensities to be set to high or low.
32	Forces the background to high-intensity. This will not be applied to a black background.
64	Forces the background to low-intensity. This may not be used if “32” is used.
128	Forces the background to high-intensity, but only if it is black. This may be used in conjunction with setting “32” or “64” for special effects.
256	Causes the “32”, “64”, or “128” setting to be used even if the COLOR_TRANS setting had no effect.

These transformations are performed in the order listed above. After this variable is applied, the COLOR_TABLE setting is applied to the program.

IO_CREATES

Setting this configuration variable to a “1” (on, true, yes) causes the runtime system to create a relative or indexed file when the program attempts to open a nonexistent file for I-O. This is provided for compatibility with some other COBOL systems. The default value is “0” (off, false, no).

IO_FLUSH_COUNT

Use this variable to specify how often the runtime should flush pending screen output during file operations. When set to a positive value, the variable indicates the number of file operations to perform between each screen flush. By default, IO_FLUSH_COUNT is set to 20.

For optimal performance, set IO_FLUSH_COUNT to zero (“0”). When IO_FLUSH_COUNT is set to zero, COBOL file verbs will not automatically flush pending screen output.

To reduce unexpected screen behavior, however, leave this variable at its default setting. The overhead at the default setting is small.

IO_READ_LOCK_TEST

When this variable is set to “1” (on, true, yes), the runtime will cause a read with no lock to fail if the file is opened for I-O and the record is locked. This setting will work with Vision files only. The setting is provided for compatibility with some other COBOL systems. The default value is “0” (off, false, no). The default behavior is to allow the read to succeed.

IO_SWITCH_PERIOD

The value of this variable affects the frequency with which the program’s threads change control based on file IO activity. After a thread performs the value of IO_SWITCH_PERIOD operations, the runtime switches control to another thread (if one exists). Note that because thread switching is also affected by other program operations (such as display I/O), it is impossible to predict or absolutely control when a thread will change control.

The default value of IO_SWITCH_PERIOD is “10”. This value will provide good results with most applications. To produce behavior that more closely imitates that of Versions 6.1 and earlier, set IO_SWITCH_PERIOD to “1”. Zero and negative values are invalid and will result in undefined behavior.

ISOLATE_FILE_CREATES

It is possible to experience unexpected file errors when trying to create a file if another process is simultaneously creating or removing the same file name. Setting ISOLATE_FILE_CREATES to “1” (on, true, yes) causes files to be created with temporary names and then renamed when they are fully formed. This prevents another process from interfering with the creation. This option is effective only with Vision, and has undefined effects when used with other file systems. We recommend that you use this option only if you are experiencing unexpected errors when trying to create a file.

JAVA_LIBRARY_NAME

This variable is designed for those calling a Java program from COBOL via the C$JAVA library routine. In this variable, specify the name of the DLL that exports the Java Native Interface (JNI) API for loading the JVM. In the case of JRE 1.4.2_04, the file is called “jvm.dll” or “libjvm.so”. If the path to the DLL is in the Path environment variable, the filename is sufficient here; otherwise, this name should be fully qualified with the path.

For details on the C$JAVA routine, see Appendix I. For information on calling Java from COBOL using C$JAVA, see section 2.3.1 in *A Guide to Interoperating with ACUCOBOL-GT.*

JAVA_OPTIONS

This variable is designed for those calling a Java program from COBOL via the C$JAVA library routine. In this variable, specify the command-line options that you want passed to the JVM when it is started.

Note that both CLASSPATH (java.class.path system property) and the java.library.path must be configured in order for C$JAVA to locate the Java class to run. The CLASSPATH is the location of “.jar” or “.class” files. The java.library.path is the location DLLs or shared objects that are required either by the runtime or by the Java Virtual Machine (JVM).

If these properties are not set in the environment, use JAVA_OPTIONS to set CLASSPATH and java.library.path. For example:

```
JAVA_OPTIONS=″-Djava.library.path="c:\usr\lib" -Xms128m
-Xmx128m -classpath /java/MyClasses/myclasses.jar″
```

JUSTIFY_NUM_FIELDS

When this variable is set to “1” (on, true, yes), all entry fields that have a numeric or numeric-edited VALUE data item associated with them are right justified (the same as if the RIGHT style were specified). You can inhibit this for a given field by specifying the LEFT style for the entry field. Note that this variable is examined only when each entry field is created and has no further effect. The default value is “0” (off, false, no).

KBD

These variables can be used in conjunction with the KEYBOARD variable to set global terminal attributes. For details, see the *ACUCOBOL-GT User's Guide*, section 4.3.2.1, "KEYBOARD variable."

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

KEY_MAP

This is an obsolete variable that has been replaced by the KEYSTROKE configuration variable. Its setting is ignored.

KEYBOARD

This variable sets global terminal attributes. For details, see the *ACUCOBOL-GT User's Guide*, section 4.3.2.1, "KEYBOARD variable."

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

KEYSTROKE

This variable redefines the action of a particular keystroke. It can also control mouse handling. For details on redefinition of keystrokes, see section 4.3.2.2 in Book 1, *ACUCOBOL-GT User's Guide*. Information on mouse handling is provided in Chapter 7 in Book 2, *ACUCOBOL-GT User Interface Programming*.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

LC_ALL

This variable supports the transfer of double-byte character variables and string literals and has meaning only on 32-bit Windows systems that support double-byte characters (e.g., Asian Windows machines). It should not be set in other environments and is needed only if you are passing data to a COBOL program from another language, such as Visual Basic, and are using C$GETVARIANT or C$SETVARIANT. By using the LC_ALL configuration variable, you cause the runtime to set the locale to a particular value. You do not need to set this variable on Japanese machines. The runtime automatically detects Japanese versions of Windows and automatically sets the locale, LC_ALL, to "Japanese_Japan.932".

The default value for this variable is "C". The C locale assumes that all characters are 1 byte and that their value is always less than 256. The value of LC_ALL is in the format:

```
language[_country[.code_page]]
```

or

```
.code_page
```

where "language" is one of the supported language strings, "country" is one of the supported country or region strings, and "code_page" is the Windows code page setting for the language and country. "country" and "code_page" are optional. For example, the following are all equivalent:

```
LC_ALL Japanese
LC_ALL Japanese_Japan
LC_ALL Japanese_Japan.932
LC_ALL .932
```

For Korean double-byte character support under Windows use:

```
LC_ALL Korean
```

For Chinese use:

```
LC_ALL Chinese
```

or

```
LC_ALL Chinese-simplified
```

or

```
    LC_ALL Chinese-traditional
```

The following are the supported language strings:

Chinese	Chinese	“chinese”
Chinese	Chinese (simplified)	“chinese-simplified” or “chs”
Chinese	Chinese (traditional)	“chinese-traditional” or “cht”
Czech	Czech	“csy” or “czech”
Danish	Danish	“dan” or “danish”
Dutch	Dutch (Belgian)	“belgian”, “dutch-belgian”, or “nlb”
Dutch	Dutch (default)	“dutch” or “nld”
English	English (Australian)	“australian”, “ena”, or “english-aus”
English	English (Canadian)	“canadian”, “enc”, or “english-can”
English	English (default)	“english”
English	English (New Zealand)	“english-nz” or “enz”
English	English (UK)	“eng”, “english-uk”, or “uk”
English	English (USA)	“american”, “american english”, “american-english”, “english-american”, “english-us”, “english-usa”, “enu”, “us”, or “usa”
Finnish	Finnish	“fin” or “finnish”
French	French (Belgian)	“frb” or “french-belgian”
French	French (Canadian)	“frc” or “french-canadian”
French	French (default)	“fra” or “french”
French	French (Swiss)	“french-swiss” or “frs”
German	German (Austrian)	“dea” or “german-austrian”
German	German (default)	“deu” or “german”
German	German (Swiss)	“des”, “german-swiss”, or “swiss”
Greek	Greek	“ell” or “greek”

Hungarian	Hungarian	“hun” or “hungarian”
Icelandic	Icelandic	“icelandic” or “isl”
Italian	Italian (default)	“ita” or “italian”
Italian	Italian (Swiss)	“italian-swiss” or “its”
Japanese	Japanese	“japanese” or “jpn”
Korean	Korean	“kor” or “korean”
Norwegian	Norwegian (Bokmal)	“nor” or “norwegian-bokmal”
Norwegian	Norwegian (default)	“norwegian”
Norwegian	Norwegian (Nynorsk)	“non” or “norwegian-nynorsk”
Polish	Polish	“plk” or “polish”
Portuguese	Portuguese (Brazilian)	“portuguese-brazilian” or “ptb”
Portuguese	Portuguese (default)	“portuguese” or “ptg”
Russian	Russian (default)	“rus” or “russian”
Slovak	Slovak	“sky” or “slovak”
Spanish	Spanish (default)	“esp” or “spanish”
Spanish	Spanish (Mexican)	“esm” or “spanish-mexican”
Spanish	Spanish (Modern)	“esn” or “spanish-modern”
Swedish	Swedish	“sve” or “swedish”
Turkish	Turkish	“trk” or “turkish”

The following are the supported country/region strings:

Australia	“aus” or “australia”
Austria	“austria” or “aut”
Belgium	“bel” or “belgium”
Brazil	“bra” or “brazil”
Canada	“can” or “canada”
Czech Republic	“cze” or “czech”
Denmark	“denmark” or “dnk”
Finland	“fin” or “finland”

France	“fra” or “france”
Germany	“deu” or “germany”
Greece	“grc” or “greece”
Hong Kong	“hkg”, “hong kong”, or “hong-kong”
Hungary	“hun” or “hungary”
Iceland	“iceland” or “isl”
Ireland	“ireland” or “irl”
Italy	“ita” or “italy”
Japan	“japan” or “jpn”
Mexico	“mex” or “mexico”
Netherlands	“nld”, “holland”, or “netherlands”
New Zealand	“new zealand”, “new-zealand”, “nz”, or “nzl”
Norway	“nor” or “norway”
Peoples Republic of China	“china”, “chn”, “pr china”, or “pr-china”
Poland	“pol” or “poland”
Portugal	“prt” or “portugal”
Russia	“rus” or “russia”
Singapore	“sgp” or “singapore”
Slovak Republic	“svk” or “slovak”
South Korea	“kor”, “korea”, “south korea”, or “south-korea”
Spain	“esp” or “spain”
Sweden	“swe” or “sweden”
Switzerland	“che” or “switzerland”
Taiwan	“taiwan” or “twn”
Turkey	“tur” or “turkey”
United Kingdom	“britain”, “england”, “gbr”, “great britain”,”uk”, “united kingdom”, or “united-kingdom”
United States of America	“america”, “united states”, “united-states”, “us”, or “usa”

LICENSE_ERROR_MESSAGE_BOX

This configuration variable prevents **acushare** licensing errors from appearing in a message box, which requires a response from the user. The error messages will instead go to the error output (stderr, or an error file if one is specified). Set LICENSE_ERROR_MESSAGE_BOX to 0 to prevent these messages from appearing in a message box. The default value is 1, which allows these messages to appear in a message box.

LISTS_UNBOXED

Meaningful only in character-based environments, this variable indicates whether list boxes should be boxed (set to a value of “0”, off, false or no) or unboxed (set to a value of “1”, on, true or yes). The default setting is “0”.

LITERAL_ENTRY

This variable can be used to loosen the literal match restrictions of ENTRY point name matching logic. By default, the matching logic is case sensitive and distinguishes between hyphens and underscores. Setting LITERAL_ENTRY to “0” (off, false, no) causes the runtime to handle ENTRY point name matching with case insensitivity (upper and lower case equivalent) and to treat hyphens and underscores (“-”, “_”) as equivalent. The default value is “1” (on, true, yes).

LOCK_DIR

When set, this controls automatic device locking on UNIX systems. This is described in section 6.1.5 of the *ACUCOBOL-GT User’s Guide*. The default value is empty.

LOCK_OUTPUT

When set to a “1” (on, true, yes), this configuration variable causes all files open for OUTPUT to be locked for exclusive use. Setting this can dramatically improve performance on VMS machines. Some other COBOL systems lock files that are open for OUTPUT. The default value is “0” (off, false, no).

LOCK_SORT

When this configuration variable is set to a “1” (on, true, yes), input files to the SORT verb are opened for INPUT ALLOWING READERS. This can improve the performance of the SORT verb slightly and also ensure that the data being sorted is not modified. The default value is “0” (off, false, no).

LOCKING_RETRIES

This configuration variable is designed for Windows 98 systems. It gives you some control over situations where a user must wait for access to a shared file. The runtime will try repeatedly to acquire the file lock, up to 400 times by default. Set this variable to the number of attempts you would like the runtime to make to acquire the file lock.

LOCKS_PER_FILE

This value determines the maximum number of record locks that can be held on a file by a single process. This value affects only the files that are maintaining multiple record locks. The default setting is “10”. The maximum value is 32767 for Vision files. Setting this variable to its maximum value can waste resources and is not recommended.

Note: If you increase the value of LOCKS_PER_FILE, you may wish to increase the value of **MAX_LOCKS** as well.

LOG_BUFFER_SIZE

This sets the maximum buffer size, in bytes, for the transaction log file. Acceptable values are from “0” to “32767”. LOG_BUFFER_SIZE is examined before each write to the log file. Its default value is “512”. If LOG_BUFFER_SIZE is set to “0”, then writes to the log file are synchronous (unbuffered). This value can also be set inside a COBOL program with the SET ENVIRONMENT verb.

LOG_DEVICE

Setting this value to “1” (on, true, yes) causes the transaction management system to assume that the log file is actually a device, rather than a file. This means that a special device locking method will be used on the log file (see the *ACUCOBOL-GT User’s Guide* section 6.1.5, “Device Locking Under UNIX”). It also guarantees that the log file will be opened “append” and that no seeks will be performed on it. This allows for the use of a tape device for the log file on many systems. The default setting is “0” (off, false, no).

LOG_DIR

This variable allows you to specify a directory to be used for holding the temporary files generated by the transaction management system. The value of LOG_DIR is treated as a prefix, much like FILE_PREFIX. If no directory is specified, temporary files are placed in the current directory.

Note: In general, you should not use remote name notation in the LOG_DIR variable. Although remote name notation is allowed for the LOG_DIR variable, it is not advisable to place temporary files on a remote server.

LOG_ENCRYPTION

If this value is set to “1” (on, true, yes), record images are encrypted before they are written to the transaction log file. The default setting is “0” (off, false, no).

LOG_FILE

This identifies the name of the default log file for transaction management. The default log file is opened at the beginning of the first transaction unless NO_LOG_FILE_OK is set to “1.” If it does not exist, it is created. This variable can be set programmatically with the SET ENVIRONMENT verb. The default setting is empty. Unless you have set NO_LOG_FILE_OK to “1”, you must set the LOG_FILE variable if you want to use transaction management.

Remote name notation is allowed for the LOG_FILE variable if your runtime is client-enabled. See *ACUCOBOL-GT User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

LOGGING

Setting this variable to “0” (off, false, no) disables the logging of file updates to the log file. This means that the data file recovery process (using the library routine C$RECOVER) is impossible. However, because rollback information is maintained internally by the runtime, the program can still use the transaction management system to START, COMMIT, and ROLLBACK transactions. This variable can be set programmatically with the SET ENVIRONMENT verb. The default setting is “1” (on, true, yes).

LOGICAL_CANCELS

This variable is used to enable *logical cancels*. Logical cancels reduce CALL overhead and can, as a result, improve performance. Cancels, both logical and physical (the default), are initiated by the CANCEL verb or through a function of the C interface. A discussion of memory management and physical and logical cancels is located in section 6.3, “Memory Management,” in Book 1. A description of the CANCEL verb is located in section 6.6 of Book 3.

LOGICAL_CANCELS can be set to the following values:

-1	(default) all cancels are physical cancels except for programs called from CICS that have the "Resident" attribute set to TRUE.
0	all cancels are physical cancels. Cancels of programs called with the C interface are treated as physical cancels even if the "cache" field is set to "1".
1	all cancels are logical cancels. Cancels of programs called with the C interface are treated as logical cancels even if the "cache" field is set to "0".

LOGICAL_CANCELS is used in conjunction with the **DYNAMIC_MEMORY_LIMIT** configuration variable, which specifies the size of the dynamic memory pool for programs. See its entry in this appendix.

MAKE_ZERO

When set to a "1" (on, true, yes), this configuration variable causes a numeric data item that contains non-numeric data to be treated as zero when that item is used in a numeric statement. Setting the value to "0" (off, false, no) causes the item to be treated "as is" with whatever effects that will have. The default value is "1" (on, true, yes).

MASS_UPDATE

When set to "1" (on, true, yes), this variable causes all OPEN...WITH LOCK statements to be treated as if they were written OPEN...WITH MASS_UPDATE. This does not apply to OPEN INPUT, however. Setting this configuration variable will improve file performance for applications that lock files, but may lead to file corruption if the program is killed before it completes. For more information on this topic, see the *ACUCOBOL-GT User's Guide,* section 6.1.6 "Indexed File Considerations." The default value is "0" (off, false, no).

MAX_ERROR_AND_EXIT_PROCS

This variable sets the maximum number of error and exit procedures that a program can install or call. The default is 64. If the limit is exceeded, program execution is aborted. For more information about error and exit procedures, see Error and Exit Procedures, CBL_ERROR_PROC, CBL_EXIT_PROC, and CBL_GET_EXIT_INFO in Appendix I of the *ACUCOBOL-GT Appendices Manual*.

MAX_ERROR_LINES

This variable sets the maximum number of lines that can be included in the error file. It's especially useful when you are using the file trace function of the debugger. When the size of the error file reaches the number of lines specified in this variable, the error file is *rewound* to its beginning, and subsequent lines of output overwrite existing lines in the file. For example, if you set the maximum to 300, then lines 301 and 302 would overwrite lines 1 and 2 in the file. Note that there is an upper limit of 32767, setting a larger number causes odd behavior. The default value is "0", which means do not limit the size of the file.

When using the "-l" runtime option (which causes the configuration settings to be logged in the trace file) the trace file wraps to the line below the MAX_ERROR_LINES entry. For this reason, MAX_ERROR_LINES should be the last entry in the runtime configuration file

To help you locate the end of the file, the message "*** End of log ***" is output whenever the runtime shuts down. Line numbers are included in each line of the file, in columns one through seven.

This variable is not available on VMS systems.

MAX_FILES

This variable sets the maximum number of files that can be opened by the runtime system. The default value is “32”. Keeping this value small conserves memory. Many operating systems limit the number of files that can be opened by a single process, so you may have to make some adjustments there too. The maximum value of MAX_FILES is 32767.

MAX_LOCKS

This value sets the maximum number of record locks that can be held by the runtime system for all of the files together. The default value matches the setting of MAX_FILES. Many operating systems also have limits on the number of record locks that can be held, so you may have to make adjustments there too. The maximum value is 32767 for Vision files. Setting this variable to its maximum value can waste resources and is not recommended.

Note: If you change the value of MAX_LOCKS, you should consider changing the value of **LOCKS_PER_FILE** as well.

MENU_ITEM

This variable affects the behavior of pull-down menus.

The default action of a menu item is to return an exception value equal to the item’s ID. You can change the default action of a particular item by using MENU_ITEM.

Use MENU_ITEM in the same fashion as the KEYSTROKE variable, except that the last entry on the line is the menu’s ID, not the key code. For example, to cause a menu item whose ID is “200” to act the same as the Delete key, use the following:

```
MENU_ITEM  Edit=Delete  200
```

Alternately, you could cause menu item “200” to call the “notepad” sample program by using:

```
    MENU_ITEM  Hot_Key=“notepad” 200
```

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

MESSAGE_BOX_COLOR

This variable is used on character-based hosts to specify the color and video attributes of characters displayed in a message box window. MESSAGE_BOX_COLOR can be set to a variety of numeric values that express combinations of color and video attributes. See the documentation for the COLOR phrase in the “Common Screen Options” section of the *ACUCOBOL-GT Reference Manual* (section 6.4.9).

If MESSAGE_BOX_COLOR is set to “0” (the default value), the message box is displayed with the default window colors and attributes.

MESSAGE_QUEUE_SIZE

This variable sets the initial size of the message queue, in bytes. The message queue is dynamically resized, as needed, to hold large messages. However, it is not resized to hold multiple messages (instead, the sending threads wait until the queue empties). Setting the value larger than the default will allow more messages to be queued. Setting it to a smaller value will allow fewer messages to be queued and conserves memory. The default size is 32767.

MIN_REC_SIZE

This configuration variable sets the minimum record size for print records, and for line sequential files for which trailing space removal has been specified. The default value is “1”. If set to “0”, then a record will be reduced to zero size (except for the line delimiter) if the line is blank. You may also set MIN_REC_SIZE to higher values to establish a minimum record size other than one.

MONOCHROME

When set to “1” (on, true, yes), this configuration variable disables color output for machines with graphics video cards. The default value is “0” (off, false, no).

ACUCOBOL-GT assumes that machines with graphics video cards are color machines. If you have a monochrome monitor attached to such a machine, some program screens may be difficult to see. You tell ACUCOBOL-GT to disable color output for these machines through the MONOCHROME configuration variable. When this variable is set to a “1”, ACUCOBOL-GT will use only black and white.

MOUSE

This variable has meaning only on systems with a mouse. When the user selects a field in the Screen Section, the exact behavior depends on the field’s underlying type. The runtime distinguishes between three classes of fields: numeric, numeric-edited, and all others. These are referred to respectively as NUMERIC, EDITED, and ALPHA.

You can control the behavior of the mouse with regard to each of these field types with the MOUSE configuration variable. This variable takes as its arguments one of the field-type names and two keywords. The first keyword defines how the field is selected when the user presses the left button. The second keyword indicates the shape that the mouse pointer should take while in the field. The first keyword can be one of the following:

None	Indicates that this type of field may not be selected with the mouse. When this keyword is used, then the second keyword (which defines the mouse’s shape) is ignored. The mouse adopts the shape used for areas of the screen that are not part of any field.
Field	Indicates that pressing the left button anywhere in the field will cause the cursor to be positioned at the beginning of the field.
Character	Indicates that pressing the left button in the field will position the cursor at the character pointed to by the mouse. If this is past the last non-prompt character in the field, the cursor will be placed just after the last non-prompt character.

The second keyword indicates the shape that the mouse pointer should take while in the field. It can be one of the following:

Arrow	The mouse pointer appears in the default arrow shape.
Bar	The mouse appears as a vertical bar. This is the “I-Bar” shape typically used to indicate that the mouse can be positioned at a particular character.
Cross	The mouse appears as cross-hairs.

You may also define the shape that the mouse will take when it is used in the *current field*. Because the action of the mouse is the same for all field types once they become the current field, the mouse shape is the same for all three types. You set the desired shape using the Current keyword in the MOUSE configuration variable. The default shape is the Bar shape.

Configuring the MOUSE variable

Depending on where you are setting the MOUSE variable, there are three methods of setting its configuration:

1. If you want to implement this variable in a configuration file, the variable can be set without using the equals sign. For example:

```
MOUSE_NUMERIC_SHAPE     Bar
```

2. If you are setting the variable in the Windows environment, the variable would look this:

```
SET MOUSE_NUMERIC_SHAPE=Bar
```

3. If you are setting the variable in your program using COBOL syntax, the variable would look like this:

```
SET ENVIRONMENT “MOUSE_NUMERIC_SHAPE” TO “Bar”
```

The default configuration is as follows:

MOUSE_ALPHA_CHARACTER	Bar
MOUSE_NUMERIC_FIELD	Arrow
MOUSE_EDITED_FIELD	Arrow
MOUSE_CURRENT	Bar

You may place multiple entries on the MOUSE configuration line, but you are not required to do so.

The following configuration variables can also be used to set the behavior of the mouse:

To set field selection:

MOUSE_ALPHA_SELECT

MOUSE_EDITED_SELECT

MOUSE_NUMERIC_SELECT

To set cursor shape:

MOUSE_ALPHA_SHAPE

MOUSE_EDITED_SHAPE

MOUSE_NUMERIC_SHAPE

MOUSE_CURRENT_SHAPE

With these variables, you need to set the first and second keywords separately. For example, to change the defaults shown above for a numeric field, you would enter:

```
MOUSE_NUMERIC_SELECT  character
MOUSE_NUMERIC_SHAPE  bar
```

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

MOUSE_FLAGS

This variable has meaning only on systems with a mouse. Indicate which mouse actions will return an exception value to your program by setting the value of the configuration variable MOUSE_FLAGS. Mouse actions that you don't want to deal with will be ignored. The value you set is actually one or more values added together. The possible values are:

1	Causes ACUCOBOL-GT to use its automatic mouse handling facility. (default)
2	Enables the "left button pushed" action.
4	Enables the "left button released" action.
8	Enables the "left button double-clicked" action.
16	Enables the "middle button pushed" action.
32	Enables the "middle button released" action.
64	Enables the "middle button double-clicked" action.
128	Enables the "right button pushed" action.
256	Enables the "right button released" action.
512	Enables the "right button double-clicked" action.
1024	Enables the "mouse moved" action.
2048	Forces the mouse pointer always to be the default arrow shape when you are using automatic mouse handling. If this is not set, then the shape of the mouse pointer varies depending on various other configuration options. See MOUSE above.
16384	This causes all enabled mouse actions that occur within your application's window to return an exception value. If this is not set, then only mouse actions that occur within the current ACUCOBOL-GT window return a value. (The current ACUCOBOL-GT window is a window created by your program with the DISPLAY WINDOW verb.)

For example, if you wanted your program to receive an exception value whenever the user pressed either the left or right button, you would set:

```
    MOUSE_FLAGS  130
```

NO_CONSOLE

This variable has meaning only on graphical systems that create an application window, such as Windows. Set this variable to "1" (on, true, yes) to indicate that you've built your own user interface entirely in C or that you are using an interface created by a code-generating tool. This is equivalent to executing the runtime system with the "-b" command-line option. When this variable is set to "1", the runtime won't create its own application window. Instead, your C code must build its own window. When you provide your own user interface, you may not use ACCEPT or DISPLAY verbs in your COBOL program (except for those that don't interact with the screen or keyboard).

The default value is "0" (off, false, no).

NO_LOG_FILE_OK

Setting this variable to "1" (on, true, yes) eliminates the need to specify a default transaction log file with the LOG_FILE variable. When this variable is set, the runtime will write transaction recovery information only to the log files specified via the *filename*_LOG variables.

The default value is "0" (off, false, no).

NO_TRANSACTIONS

This variable allows you to disable ACUCOBOL-GT's built-in transaction management system. When NO_TRANSACTIONS is set to "1" (on, true, yes), all calls to ACUCOBOL-GT's built-in transaction management system return without doing any work. This affects all file systems in use. The value of NO_TRANSACTIONS is checked once, the first time a BEGIN, COMMIT, or ROLLBACK is attempted and it is not checked again. Therefore, although the variable can be set in the program, the effective setting cannot be changed after the first transaction management action has been attempted. The default value is "0" (off, false, no), meaning that the built-in transaction management facility is enabled.

NT_OPP_LOCK_STATUS

This configuration variable controls how files on a shared drive are opened if you are working in the Windows opportunistic locking mode. NT_OPP_LOCK_STATUS can take one of four values: CREATEFILE, SAFE, GETFILETYPE, or FAST. The default value is “SAFE”, which is a synonym for “CREATEFILE”.

If you set this variable to “GETFILETYPE” or “FAST” (synonyms for each other), the runtime uses the fast method of opening files.

Note: If your Windows installations are not completely up to date with all available patches from Microsoft, particularly those related to opportunistic locking, the GETFILETYPE or FAST setting may cause file corruption (error 98).

NESTED_AX_EVENTS

When an application dialog contains an ActiveX control that is assigned an event procedure, the event handler sometimes triggers additional ActiveX events. This variable determines whether or not the event procedure will be nested.

Set this variable to “1” (on, true, yes) if you want the event procedure to be nested. (This is the default). When NESTED_AX_EVENTS is set to “1”, the runtime allows events to trigger while it is processing other events. It is your responsibility to know when the event procedure is busy and reject events when this is the case, or to look for specific events and properly handle them. For example, consider this code:

```
AX-EVENT.
MOVE 1 TO MY-LOOP.
PERFORM UNTIL MY-LOOP = 10
* Do some stuff
ADD 1 TO MY-LOOP
END-PERFORM
```

When NESTED-AX-EVENTS is set to “1”, it is possible that when your code is inside the event, possibly executing the loop for the fifth time, a new event triggers, setting MY-LOOP back to “1”. The perform loop could execute simultaneously in two threads on the same data, and the runtime could crash. When you do not have reentrant events, MY-LOOP can only become “1” one time. This is the case when NESTED-AX-EVENTS are set to “0”.

Set NESTED_AX_EVENTs to “0” (off, false, no) if you do not want the event procedure to be nested. Be aware, however, that this option may cause you to lose certain events (typically events triggered by modifications made in the event procedure).

When NESTED_AX_EVENTS is set to “0”, once a program has entered an ActiveX control’s event procedure, new events are ignored. This prevents the runtime from executing the same code, at the same time. However, events that are imperative for the code execution may be refused.

Note: NESTED_AX_EVENTS applies only to the local runtime and has no effect in thin client scenarios.

NO_BARE_KEY_LETTERS

This variable is related to the terminal manager KEYSTROKE EDIT=ALT method of requiring users to press the Alt key along with the key letter to move to a new control. In character mode and when accepting a push-button, check box, or radial button, the runtime’s default behavior is to terminate the accept if the key letter of a control is pressed, and move to that control.

This behavior can be changed by setting NO-BARE-KEY-LETTERS to “TRUE”. When set to “TRUE”, in order to move to these types of controls, users will be required to press the key set by the KEYSTROKE setting EDIT=ALT before pressing a key letter of that control. The default setting is “FALSE”, which uses the behavior described in the previous paragraph.

NUMERIC_VALIDATION

If this configuration variable is set to “1” (on, true, yes), the runtime checks for proper format when data is converted to a numeric type (via a MOVE, for example). When NUMERIC_VALIDATION is set to the default “0” (off, false, no), numeric conversion checking does not occur.

OLD_ARIAL_DIMENSIONS

The Arial font shipped with Windows 98 Version 2 has a character width of 35 pixels, while the Arial font shipped with earlier versions of Windows and Windows NT has a width of 23 pixels. This might cause field overlap or screen distortions in programs that rely on the smaller version of the Arial font. Setting this variable to “1” (on, true, yes) causes the runtime to use the 23-pixel measurement for fields, regardless of which version of Arial (35 or 23-pixel) is being used.

The default value of “0” (off, false, no) will cause fields to be sized according to the version of Arial used.

Note: Because the 35-pixel version of Arial is wider, uppercase characters may be truncated when their size is computed with the 23-pixel measurement. Use of this variable may not compensate for all possible character width sizing issues. Some reprogramming of your screens may be required.

OPEN_FILES_ONCE

This variable allows different logical files that access the same physical file to open the physical file only once. The default for this variable is “1” (on, true, yes). This variable is valid only for UNIX runtimes.

OPTIMIZE_CONTROL_RESIZE

This configuration variable determines how the runtime optimizes control resize requests. Prior to Version 5.2, the runtime would optimize away requests to resize a screen control if the new size and position matched the control's current size and position on the screen. In Version 5.2, or later, the runtime optimizes the control resize request using the SIZE and LINES indicated (or implied) by the program. Setting OPTIMIZE_CONTROL_RESIZE to "0" (off, false, no) prevents any optimization of control resizing operations. The default of "1" (on, true, yes) enables the new behavior. See Appendix C: Changes Affecting Previous Versions, in the *ACUCOBOL-GT Appendices Manual* for more details.

OPTIMIZE_INDIVIDUAL_LINKAGE

This variable enables the runtime to perform address optimizations on each Linkage item individually. In versions prior to 8.1 this optimization was done either for all Linkage items or for none of them, which could result in secenarios where optimizations could have occured on some items, but did not.

The default and recommended setting is "1" (on) because the main effect is improved CALL performance in a greater number of scenarios. Usually, the only reason to turn this variable off is if a flaw is suspected in the optimization.

PAGE_EJECT_ON_CLOSE

When set to "1" (on, true, yes), this variable will cause print files to print a page advance record when the file is closed, unless the close contains the NO REWIND phrase. This is provided for compatibility with RM/COBOL version 2. The default value is "0" (off, false, no).

PAGED_LIST_SCROLL_BAR

This variable applies only in text-mode environments. PAGED_LIST_SCROLL_BAR can be set to “-1”, “0”, or “1”. The default value is “-1”. When set to “-1”, the vertical scroll bar is displayed to the right of a paged list box if the user interface configuration supports a mouse. Otherwise, the right border appears just like the left border. The appearance depends on whether the NO-BOX style is set and the values of the FULL-BOXES and LISTS-UNBOXED configuration variable settings.

The runtime internally calls the W$MOUSE library routine with the TEST-MOUSE-PRESENCE op-code to determine whether the user interface supports a mouse. Note that mouse support is available for X terminals only if the a_termcap entry includes the “km” function. (See the *AcuCOBOL-GT User’s Guide*, section 4.6.8, “Mouse Support for X Terminals.”)

When PAGED_LIST_SCROLL_BAR is set to “1”, the vertical scroll bar is always displayed to the right of a paged list box. When set to “0”, the vertical scroll bar is never displayed to the right of a paged list box.

PARAGRAPH_TRACE

This variable is used for debugging purposes and turns on paragraph tracing. Set this variable to “1” (on, true, yes) to turn on paragraph tracing from within the configuration file or the COBOL program. This is equivalent to the debugger “tp” command. The COBOL program must be compiled with symbols (“-Gs”, or anything that implies that option) to get any error output.

PERFORM_STACK

This variable sets the depth to which PERFORM statements can be nested at runtime when the “-Zr” compile-time option is used. The default value is “128”. The maximum value is “10916”. Setting this variable to its maximum value can waste resources and is not recommended.

PRELOAD_JAVA_LIBRARY

This variable is designed for those calling a Java program from COBOL via the C$JAVA library routine. By default, the Java Virtual Machine (JVM) is loaded by the runtime the first time it executes a CALL C$JAVA statement. If you want to load the JVM when the runtime is started, set this configuration variable to “1”. If you do not want to preload the JVM, set the variable to “0”.

For details on the C$JAVA routine, see Appendix I. For information on calling Java from COBOL using C$JAVA, see section 2.3.1 in *A Guide to Interoperating with ACUCOBOL-GT.*

PROFILE_TYPE

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.

PROMPTING

This variable is used on character-based hosts to turn ENTRY-FIELD prompting off or on. When PROMPTING is set to “0” (off, false, no) prompting is not performed. The default value of PROMPTING is “1” (on, true, yes).

QUEUE_READERS

This configuration variable evens out user access by modifying the rules Vision uses when several users are accessing a file. This variable applies to UNIX machines. Because of restrictions, it is recommended only for sites that are experiencing performance problems with updaters.

By default, the runtime allows multiple readers to access a file simultaneously, while updaters require exclusive access to the file. When a file has many readers, an updater can get blocked out of the file for a period of time while the runtime waits for a moment when there are no active readers. While this allows processes that read the file to have nearly immediate access, updaters may need to wait for a noticeable amount of time.

The QUEUE_READERS configuration variable lets you request that the runtime service each user in turn. This means a reader will have the same priority for accessing a file as an updater does. Each user is processed in turn so that access to files is evenly balanced among all the users.

By default, QUEUE_READERS is set to “0” (off, false, no). Set it to “1” (on, true, yes) to force the readers to take turns instead of having immediate access.

Because of technical limitations in the UNIX file system, if you use this configuration variable you must provide read-write access to all indexed and relative files that the runtime uses. This is true even for files that are open only for input--UNIX requires that the runtime have write access to the files in order to place the kind of lock that causes each user to take turns.

QUIT_MODE

This variable has meaning only on graphical systems such as Windows. It gives you control over the Close action that appears on the System menu in a graphical environment. You may use the QUIT_MODE variable with only the main application window. All other windows return the CMD-CLOSE event when they are closed. QUIT_MODE has no affect on windows created with the NO-CLOSE phrase (see formats 11 and 12 of the DISPLAY statement, in Book 3, *ACUCOBOL-GT Reference Manual*, section 6.6).

Many COBOL programs should not be shut down in an uncontrolled manner. This is especially true of any application that updates several files in a row. If the program is halted after updating the first file, but before updating the last, the files are left in an inconsistent state. For this reason, ACUCOBOL-GT allows you to control the Close action.

To do this, you set QUIT_MODE to a non-zero value. The value that you specify affects the Close action as follows:

-2	Disable Close: disables the Close action entirely. The Close menu item will appear gray on the System menu, and the user will not be able to select it.
-1	Close only on input: the runtime disables the Close action except when it is waiting for user input. This prevents the user from stopping the runtime in the middle of a series of file operations, but still allows the user to quit the application any time that the application is waiting for input.
0	Always Close: the runtime halts the program whenever Close is selected from the system menu.
>0	Program controlled Close: when a positive value is used, the Close item becomes a standard menu item with an ID equal to the value of QUIT_MODE. You may then handle the Close item just like any other menu item.

For example, if you set QUIT_MODE to “100”, then your program will receive exception value 100 when the user selects the Close item. If you wanted to call a special shutdown program when the user selected Close, you could assign the Close action to a hot-key program:

```
MENU_ITEM  Hot_Key =“shutdown”  100
```

In this example, the “shutdown” program might pop up a small window to confirm that the user wanted to exit and, if so, do a STOP RUN.

If you start your program in “safe” mode with the “-s” runtime option, then QUIT_MODE will be initialized to “-2” instead of “0”. This prevents the user from using the Close menu item. A QUIT_MODE entry in the configuration file takes precedence over the default handling of “-s”.

If a user attempts to end the Windows session when it is not allowed, a pop-up message box asks the user to terminate the application first. You can customize the message that appears in the box by setting the TEXT configuration variable, message number 18.

Note: The QUIT_MODE setting affects only the main application window. All other windows always return the event CMD-CLOSE when the window is closed.

QUIT_ON_FATAL_ERROR

This configuration variable applies only when running in HP COBOL compatibility mode (with the “-Cp” compiler option). The QUIT_ON_FATAL_ERROR configuration variable causes the runtime to call the MPE QUIT intrinsic when an error occurs. The MPE job control word (JCW) is then set, and the MPE environment can determine if the program terminated with a fatal error. When set to “1” (on, true, yes), QUIT_ON_FATAL_ERROR calls the MPE QUIT intrinsic. The default setting is “0” (off, false, no).

QUIT_TO_EXIT

When this variable is set to “1” (on, true, yes), the user must press the close button on the title bar (or an alternate close mechanism provided by the window) after the program executes a STOP RUN. The default value is “0”.

RECURSION

ACUCOBOL-GT allows a program to call itself, directly or indirectly. A CALL statement that attempts to call an active program is termed a *recursive call*.

To use recursive calls, you must set the configuration variable RECURSION to “1” (on, true, yes). The default setting for RECURSION is “0” (off, false, no), which disallows recursive calls.

When you allow recursive calls, an active program may be called again. This causes a new copy of the program to be loaded into memory and executed, as if it were the first call of that program. Files and data described in that program are local to each copy of the program.

More specifically, the runtime assigns a *recursion level* to each recursively called program. The first time a program is called, it is assigned a recursion level of “0”. If that program is still active and it is called again, it receives a recursion level of “1”. The recursion level is incremented by 1 for each active copy of the same program.

When you call a program at a specific recursion level for the first time, it is freshly loaded from disk and its Working-Storage data items are given their initial values as defined by their VALUE clauses. Subsequent calls to a program at the same recursion level will find the files and data left in the same state that the program had when it last exited.

Files and data items are distinct between different recursion levels.

When you CANCEL a recursively called program, all of its inactive copies are removed from memory. Active copies are left alone. Subsequent calls to any of the canceled recursion levels will reload the program from disk and reinitialize the files and data items.

If you need to share data between different active copies of the same program, you can pass this data through the Linkage Section. Alternatively, you can share both files and data items by declaring them as EXTERNAL items. Yet another option is the **RECURSION_DATA_GLOBAL** configuration option described below.

The runtime system shares the program code for recursively called programs. Thus, while each recursion level has its own set of data, there is only one copy of the Procedure Division code in memory, regardless of how many active copies of the program there are. The runtime system does not, however, share overlays. Each copy of the program in memory has its own overlay area.

RECURSION_DATA_GLOBAL

This configuration variable allows you to configure the runtime so that each instance of a recursively called program shares the same data as the original instance of the program. The primary reason for configuring the runtime in this manner is if you are migrating code that relies on this behavior from another COBOL system, such as HP COBOL.

When RECURSION_DATA_GLOBAL is set to "1" (on, true, yes), files and data items in a recursive call of a program refer to the identical items in the original call of that same program. This is true regardless of the entry point into the program. Changes to data or file state made in any recursive instance of the program are seen by all other instances of that program in the same run unit.

Note that to use this feature, you must not only set the configuration variable RECURSION_DATA_GLOBAL to "1", you must also set the **RECURSION** configuration variable to "1" to allow recursion (which is not allowed under standard ANSI-85 COBOL rules).

The default is "0" (off, false, no).

REL_DELETED_VALUE

This configuration variable is helpful when you use relative files and need to have a valid record that contains binary zeros. However, because binary zeros are used as the deleted record marker, you have to be able to change this marker. REL_DELETED_VALUE can hold the ASCII character value for the new deleted record marker.

REL_LOCK_READ_THROUGH

This variable allows you to "read through" relative file record locks in Windows environments. This means that READs that do not assert a lock are allowed to READ a relative file record even if it is locked by another process. To turn on the "read through" capability, all processes accessing the relative file must set the configuration variable REL_LOCK_READ_THROUGH to "1" (on, true, yes). When this variable is turned on, the Vision library uses an alternate location for relative file record locks that does not block other processes from reading the records. This is necessary because Windows has "enforced" file locks that preclude all other access to the locked region. Failure to set the configuration variable to the same value on all processes accessing the relative file results in undefined behavior. This variable has no effect on UNIX platforms. For more information about relative files and record locking, refer to section 6.1, "Handling Files," of the *ACUCOBOL-GT User's Guide*.

RENEW_TIMEOUT

When set to "1" (on, true, yes), this variable restarts the timeout used by ACCEPT BEFORE TIME after each keystroke that the user makes. The default setting is "0" (off, false, no), which means that the timeout is canceled as soon as the user starts typing.

RESIZE_FRAMES

This variable is used to turn off the automatic resizing of frames that is performed on character-based hosts. By default, a character-based host runtime automatically resizes frames to visually surround all controls whose home coordinates are bounded by the frame. This makes it easier to maintain applications that run on both character and graphical systems. To turn automatic resizing off, set RESIZE_FRAMES to "0" (off, false, no). The default value is "1" (on, true, yes).

RESIZE_FREELY

Normally, under a graphical host (such as Windows), the runtime will not allow you to resize an AUTO-RESIZE window larger than its logical size (as determined by the number of rows and columns the program requested when it created the window). When the RESIZE_FREELY variable is set to "1" (on, true, yes), the user can resize the window to any size. Maximizing the window will cause the window to occupy the entire available screen. The region of the window that lies outside of the logical area will not be accessible by the program and will be shown as the background color of the logical window (defined as the last background color to be applied to the entire window). Setting this option is essentially a cosmetic change to the way your application looks when maximized. The default value is "0" (off, false, no).

RESTRICTED_VIDEO_MODE

This value determines which rules will be followed when the program is positioning attribute characters for “magic cookie” terminals (terminals whose attributes occupy screen positions). For more details, see the *ACUCOBOL-GT User’s Guide,* section 4.5, “Restricted Attribute Handling.” The default is “0”.

RMS_NATIVE_KEYS

This variable is for use only on VAX/VMS systems. When it is set to “1” (on, true, yes), it causes the runtime to specify a key type for numeric keys. In order to make use of this variable, you must also create XFD files at compile-time (“-Fx”), and you must set the configuration variable XFD_DIRECTORY to point to the directory containing those XFD files.

When these steps have been taken, the runtime will create RMS files with keys having either the packed decimal or integer attribute, under certain conditions dictated by the RMS file system. In particular, a key will have the packed decimal or integer attribute only if it is a single-segment key and only if there is a single field in the key. In this case, a USAGE COMP-3 data item in the key will receive the packed decimal attribute, and a USAGE COMP-5 data item (if it is 2, 4, or 8 bytes long) will receive the integer attribute.

One effect of having this attribute set on key fields is that the order of the data is changed. Without this attribute, a file that has records with keys -3, -2, -1, 0, 1, 2, 3 would have those records ordered in this way: 0, 1, -1, 2, -2, 3, -3. With this attribute, those records would be ordered in this way: -3, -2, -1, 0, 1, 2, 3. The default setting for this variable is “0” (off, false, no).

SCREEN

This configuration variable controls a variety of screen configuration options. For details, see the *ACUCOBOL-GT User’s Guide*, section 4.4.2, “The SCREEN Option.”

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

SCREEN_COL_PLUS_BASE

This variable allows you to configure the COLUMN PLUS phrase in the Screen Section. This capability is provided to improve compatibility with other COBOLs. SCREEN_COL_PLUS_BASE allows you to choose the behavior that works best for your program. It takes the following values:

-1	(default) This value causes the runtime to determine the behavior of the COLUMN PLUS phrase based on whether ICOBOL compatibility has been specified with the "-Ci" compile option. If so, "COLUMN + 1" produces a space between items, and "COLUMN + 0" creates adjacent items. If ICOBOL compatibility has not been specified, "COLUMN + 1" produces adjacent items. This matches the prior behavior of ACUCOBOL-GT.
0	This value causes column adjustments to start counting at zero. In this case, "COLUMN + 0" produces adjacent items, and "COLUMN + 1" puts a space between items.
1	This value causes column adjustments to start counting at one. In this case, "COLUMN + 1" produces adjacent items, and "COLUMN + 2" puts a space between items.

SCREEN_TRACE

This variable is used for debugging and turns on screen tracing. Set this variable to "1" (on, true, yes) to turn on screen tracing from within the configuration file or the COBOL program. This is equivalent to the debugger "ts" command.

SCRIPT_STATUS

This variable controls the behavior of ACCEPT FROM INPUT STATUS when the input is not attached to a terminal. If SCRIPT_STATUS has its default setting of "0" (off, false, no), an ACCEPT FROM INPUT STATUS statement will return a fixed value when the program has redirected input. The value returned is the value of the **INPUT_STATUS_DEFAULT** configuration variable.

When SCRIPT_STATUS is not “0”, and input is redirected, then ACCEPT FROM INPUT STATUS will return the actual status of the script file (i.e., it will return “1” (on, true, yes) unless the script file has been exhausted).

SCRN

This variable can be used in conjunction with the SCREEN variable to control many attributes of the video sub-system. For details, see the *ACUCOBOL-GT User’s Guide*, section 4.4.2, “The SCREEN Option.”

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

SCROLL

This variable affects when screen scrolling will occur. When it is set to “1” (on, true, yes), scrolling and cursor positioning occur normally. When it is set to “0” (off, false, no), screen scrolling will occur only as the result of a SCROLL phrase in an ACCEPT or DISPLAY statement, and any DISPLAY statement that references a line past the bottom of the current window will be ignored. ACCEPT statements that reference a line past the bottom of the current window will be placed in the home position of the window. The default setting is “1”.

*server*_MAP_FILE

This variable is used to point to the character map file used for translating international character sets between a client machine and a specific server that uses different character codes. The map file is a simple text file that you create with an editor of your choice. Each line in the map file must contain two values in either decimal or hexadecimal: the character code of the character on the client machine, and the character code of the same character on the server. Use a # sign to indicate a comment.

The map file may be stored on either the client machine or the server machine.

The server specified in the configuration variable name must match the server specified in the remote name notation that points to the data files. 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, create a map file and 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
```

If the map file is located on a server, you must have the AcuServer product on that server, to enable client access.

The runtime first searches for the configuration variable *server*_MAP_FILE and, if it is found, uses that setting to locate the map file. If that variable is not set, the runtime searches for DEFAULT_MAP_FILE. If that variable is also not set, then no character translation is done.

*server*_PASSWORD

Designed to be defined in the environment (rather than in the configuration file), *server*_PASSWORD and its mate *server_port*_PASSWORD make working with AcuServer easier when the compiler and **cblutil** are called repeatedly from the AcuBench integrated development environment. In this scenario, when one of these variable is used, the user never has to enter a password. When these variables are not used, if a password is required the user must provide it repeatedly.

Set *server*_PASSWORD to the value of the password. For example:

```
MERCURY_PASSWORD=we1rneB
```

where *server* is replaced by the name of the server.

The compiler checks the variable *server_port*_PASSWORD first. If it isn't defined, *server*_PASSWORD is checked. If *server*_PASSWORD is not defined, the user is prompted for a password. If either variable is defined, but the value does not match the value in the AcuAccess file, the connection attempt fails.

*server_port*_PASSWORD

Designed to be defined in the environment (rather than in the configuration file), *server_port*_PASSWORD and its mate *server*_PASSWORD make working with AcuServer easier when the compiler and **cblutil** are called repeatedly from the AcuBench integrated development environment. In this scenario, when one of these variable is used, the user never has to enter a password. When these variables are not used, if a password is required the user must provide it repeatedly.

Set *server_port*_PASSWORD when you want to connect to a server on a particular port. For example, to set a password to connect to a server named "MERCURY" that is listening on port 4330, you can set the following:

```
MERCURY_4330_PASSWORD=we1rneB
```

where *server* is replaced by the name of the server, and *port* is replaced by the port number that AcuServer is using.

The compiler checks the variable *server_port*_PASSWORD first. If it isn't defined, *server*_PASSWORD is checked. If *server*_PASSWORD is not defined, the user is prompted for a password. If either variable is defined, but the value does not match the value in the AcuAccess file, the connection attempt fails.

SHARED_CODE

For many UNIX machines, ACUCOBOL-GT supports the ability to have multiple users share the same copy of a COBOL program's object code in memory. This configuration variable indicates which programs you want to share code. Use of shared memory is recommended only if you have a problem with limited memory and excessive swapping. In this case, the advantage of reduced swapping will usually more than make up for the overhead added by sharing memory. To use shared code for all of your programs on UNIX, add the following line:

```
SHARED_CODE   1
```

This will cause all programs to attempt to share code. Every code segment loaded into memory will be placed into shared memory until shared memory is full. Further code segments will then be placed in conventional memory. 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.

Since shared memory is a limited resource under UNIX, you will usually want to restrict the use of shared code to those programs where it will be most beneficial. This will ensure that other programs do not use up all of the available shared memory first. To do this, specify each program 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 use this method, “Program1”, “Program2”, and so forth are the PROGRAM-IDs from the programs’ Identification Divisions (note that a program’s object file name is *not* used). If you use this method, then setting SHARED_CODE to “1” will have no effect.

For additional information, see the *ACUCOBOL-GT User’s Guide*, section 2.11, “The acushare Utility Program.”

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

SHARED_LIBRARY_EXTENSION

This variable allows you to define the filename extension for UNIX/Linux shared object libraries. The default value is ".so". This variable has meaning only on systems that support UNIX shared libraries.

SHARED_LIBRARY_LIST

This variable allows you to specify the names of UNIX/Linux shared object libraries or Windows DLLs.

SHARED_LIBRARY_LIST can be set in one of three ways:

1. In the environment
2. In the runtime configuration file
3. Programmatically with the SET ENVIRONMENT statement

The runtime loads the listed objects on program startup or as the result of a SET ENVIRONMENT statement. Names must be delimited by spaces, colons (UNIX/Linux), or semicolons (Windows).

With DLLs, you can specify both the name of the DLL and the calling convention to use. Any calling convention specified this way overrides the DLL_CONVENTION variable setting. For information about specifying DLLs and calling conventions, see section 3.3.2, "Loading DLLs with Configuration Variables," in *A Guide to Interoperating with ACUCOBOL-GT.*

You can also list objects without path information and use the SHARED_LIBRARY_PREFIX configuration variable to specify a set of directories that the runtime will search when attempting to load a shared library.

Once loaded, functions exported by these libraries can be called directly.

The SET ENVIRONMENT statement can be used to set SHARED_LIBRARY_LIST any number of times during program execution. Each time it is set, the runtime loads the libraries listed. Previously loaded libraries remain loaded. Libraries loaded with SHARED_LIBRARY_LIST remain in memory until the process exits. The CANCEL statement cannot be used to unload the library.

On some systems, such as AIX, if the shared module is a member of an archive, you must specify the name of the member in parentheses after the name of the archive. For example:

```
SHARED_LIBRARY_LIST="/usr/opt/db2_08_01/lib/libdb2.a(shr.o)"
```

SHARED_LIBRARY_LIST is like the runtime "-y" option, except that it does not require setting the SHARED_LIBRARY_EXTENSION variable, and unlike "-y", you can mix ".a" and ".so" libraries in the list.

Note: The SHARED_LIBRARY_LIST configuration variable 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.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

SHARED_LIBRARY_PREFIX

This variable allows you to specify a set of directories that the runtime will search when attempting to locate a shared library. The format of the value is the same as that allowed for **FILE_PREFIX**. You can set SHARED_LIBRARY_PREFIX in the configuration file, environment, or programmatically with the SET verb.

The default value on Windows systems is empty.

The default value on UNIX and Linux systems is "/opt/acucorp/810/lib /opt/acu/lib". This helps the runtime find "libclnt.so" (or "libclnt.sl") when the operating system's shared library environment variable (e.g., LIBPATH, LD_LIBRARY_PATH, SHLIB_PATH, etc.) is not set.

SHUTDOWN_MESSAGE_BOX

This variable allows you to specify whether or not you want the runtime's shutdown message to be displayed in a message box. If this variable is set to "0" (off, false, no), the runtime will display the shutdown message to the screen without a message box. The default value is "1" (on, true, yes).

SORT_DIR

This variable allows you to place temporary files used by the SORT verb in another directory. 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, much like FILE_PREFIX. You can improve the performance of the SORT verb by placing the temporary files on a fast device. Take care, however, that the device has enough free space to hold twice the size of the data to be sorted.

You may not use the SORT_DIR variable with AcuServer.

Remote name notation is allowed for the SORT_DIR variable if your runtime is client-enabled. See *ACUCOBOL-GT User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

SORT_FILES

This configuration variable sets the number of temporary files used by SORT. The acceptable range is from 4 to 64. The default value is "8".

Increasing the number of files used will usually improve SORT performance, particularly for large sorts. Note that you must have enough available file handles to open all of the temporary files concurrently. In general, the number of files available is more important than the amount of memory used. If you are experiencing long sorts, try increasing the number of files before you increase the amount of sort memory.

The SORT verb removes all of its temporary files, except for one, prior to beginning its output phase.

SORT_MEMORY

This variable specifies the number of 64 KB blocks of memory that the SORT verb will try to allocate when it executes. The acceptable range is from 1 to 16384. The default value is “32”. Using a value lower than the default can be useful if memory is tight on the host machine. Using a higher value may enhance SORT performance.

Take care, when increasing the SORT_MEMORY setting, to ensure that you do not assign too much memory to the runtime. For most operating systems, the memory used by SORT is not returned to the system. While the runtime may use the memory for other purposes, this memory is not available to other programs until the runtime exits.

The SORT verb will attempt to allocate the amount of memory specified in SORT_MEMORY. If the requested amount is not available, the runtime will return an out of memory error.

SPACES_ZERO

This configuration variable applies only to object files generated with ACUCOBOL-85 Version 1.5 and earlier. For later object files, use the “-Zz” compiler option. When SPACES_ZERO is “1” (on, true, yes), it alters the method in which USAGE DISPLAY data items are used by the runtime system. The main effect is that, in most cases, a data item containing spaces will be treated as if it contained zeros. Note that this may not occur in all instances because the ACUCOBOL-GT compiler may construct code that directly acts on a data item without first converting it to a number. The default value is “0” (off, false, no).

SPOOL_FILE

This configuration variable allows you to hold a *pipe* open when you close the named file with the CLOSE WITH NO REWIND verb. This enables you to gather multiple reports into a single job for the print spooler. For additional details about pipes and file name interpretation, see the *ACUCOBOL-GT User’s Guide*, section 2.8, “File Name Interpretation.”

The value given to the SPOOL_FILE variable must be the ASSIGN name of a sequential file that has been attached to a pipe. The pipe must be attached to the ASSIGN name in the COBOL configuration file via the "-P" option. For example, suppose you have a file defined as follows:

```
SELECT PRINT_FILE
ASSIGN TO PRINT "PRINTER"
```

and that you have the following pipe defined in the COBOL configuration file:

```
PRINTER  -P  lp  -s
```

Then, to specify that you want to keep the pipe open when the file is closed WITH NO REWIND, you would add the following line to the COBOL configuration file:

```
SPOOL_FILE PRINTER
```

The name specified for SPOOL_FILE is processed in the same way as the external file name specified in the file's ASSIGN clause.

When the corresponding file is closed with a NO REWIND option, the pipe is not closed. Instead, if the file is later opened again, the same pipe is used. The pipe is not closed until a CLOSE verb without the NO REWIND option is executed on that file, or until the run unit finishes. Only one pipe can be held open in this manner.

STD_FIXED_FONT

This configuration variable allows you to set the standard font used by the Windows version of the runtime. It can be set in the configuration file to one of the following values:

-1	(default) The web runtime uses ANSI_FIXED_FONT. Other instances of the runtime use SYSTEM_FIXED_FONT.
0	All runtimes use ANSI_FIXED_FONT for the standard font.
1	All runtimes use SYSTEM_FIXED_FONT for the standard font.
2	All runtimes use OEM_FIXED_FONT for the standard font.

If this variable has not been set, or has an invalid value, it will default to "-1".

STOP_RUN_ROLLBACK

When this variable is set to “1” (on, true, yes), the system performs an implied ROLLBACK rather than a COMMIT after a STOP RUN.

With a “0” (off, false, no) setting for this variable, the system performs an implied COMMIT after a STOP RUN. The default value for this variable is “0”.

STRIP_TRAILING_SPACES

This variable provides an alternate method for determining which LINE SEQUENTIAL files will have trailing spaces removed from records written to them. At the time a LINE SEQUENTIAL file is opened, the value of this variable is examined. If this variable is “1” (on, true, yes), then automatic space suppression is applied to this file.

Otherwise, the file is processed according to the normal rules, as described in the *ACUCOBOL-GT User's Guide*, section 6.1.1, “Sequential files.” The default value for this variable is “0” (off, false, no).

Note that a related configuration variable is the **EXTFH_KEEP_TRAILING_SPACES** variable.

SWITCH_PERIOD

This variable helps determine how frequently threads switch control. When a thread executes SWITCH_PERIOD number of selected operations, the threads switch control. The selected operations are generally comparisons. Comparison operations are used to cause compute-bound threads to switch.

Setting the value of SWITCH_PERIOD lower will increase the overhead spent switching threads, but increase the uniformity of thread execution. Setting the value very low can significantly hurt performance. The default value is “100”.

SYSINTR_NAME

This variable defines the location of the SYSINTR file that may be used with MPE emulation software. This variable must be specified with HFS syntax and set to the full path of the SYSINTR file. For example:

```
SYSINTR_NAME /opt/mpux/etc/sysintr.txt
```

TC_AUTO_UPDATE_FAILED_MESSAGE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. If the thin client automatic update process fails for any reason, a message box may appear informing the user of the failure. The TC_AUTO_UPDATE_FAILED_MESSAGE configuration variable lets you specify the text in this message box. Its default value is

```
ACUCOBOL-GT Thin Client: Automatic update was
unsuccessful
```

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User's Guide.*

TC_AUTO_UPDATE_FAILED_TITLE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. If the thin client automatic update process fails for any reason, a message box may appear informing the user of the failure. The TC_AUTO_UPDATE_FAILED_TITLE configuration variable lets you set the title bar text for this message box. Its default value is

```
ACUCOBOL-GT Thin Client
```

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User's Guide.*

TC_AUTO_UPDATE_NOTIFY_FAIL

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. If the thin client automatic update process fails for any reason, a message box may appear informing the user of the failure. If you do not want the thin client to inform the user that the automatic update has failed, set the TC_AUTO_UPDATE_NOTIFY_FAIL configuration variable to “false” (0, off, no). The default value of this variable is “true” (1, on, yes).

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_AUTO_UPDATE_QUERY

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. When an event triggers the update process, the thin client displays a message box informing the user that an upgrade is required. The default setting of “1” (on, true, yes) for the TC_AUTO_UPDATE_QUERY configuration variable enables the display of that message box. Setting this variable to “0” (off, false, no) prevents the message box from appearing.

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_AUTO_UPDATE_QUERY_MESSAGE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. When an event triggers the update process, the thin client displays a message box informing the user that an upgrade is required. The value of the TC_AUTO_UPDATE_QUERY_MESSAGE configuration variable determines the message displayed in that message box. The default value of the variable depends on the circumstances that triggered the automatic update. For example, if the automatic update is initiated by a version or protocol number mismatch, the default message displayed is:

```
Incompatible server version
Server version: <srvvers>, client <clntvers>
Server protocol: <srvproto>, client <clntproto>
Press OK to automatically correct this problem
```

where <*srvvers*>, <*clntvers*>, <*srvproto*>, and <*clntproto*> are replaced by the server version, client version, server protocol number, and client protocol number, respectively.

For detailed information about other default values for this configuration variable and about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_AUTO_UPDATE_QUERY_TITLE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. When an event triggers the update process, the thin client displays a message box informing the user that an upgrade is required. You use the TC_AUTO_UPDATE_QUERY_TITLE configuration variable to specify the title bar text in that message box. The default value of this variable is

```
ACUCOBOL-GT Thin Client
```

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_AX_EVENT_LIST

In thin client deployments, this variable lets you control which events your program receives, giving you more control over the volume of network traffic. It must be set in the configuration file and cannot be changed programmatically with the SET verb. It contains the numeric value of a single .NET or ActiveX event type or a list of .NET or ActiveX event types separated by non-numeric characters like spaces or commas. Whether your program receives these events depends on the value of TC_EXCLUDE_EVENT_LIST. If its value is “0”, then your program receives the events listed in TC_AX_EVENT_LIST. If TC_EXCLUDE_EVENT_LIST is set to “1”, then the events listed in TC_AX_EVENT_LIST are not sent to your program.

TC_CHECK_ALIVE_INTERVAL

This variable allows you to set a time interval in seconds (a value between “1” and “32767”) during which the server runtime checks for thin client activity. This activity can be either regular thin client user interaction or, if the user interface is inactive, two “ping” messages sent by the thin client during the defined interval. If no thin client activity of any kind occurs during a particular interval, the server runtime process exits. Setting this variable to “0” disables the client activity check feature. The default value is “300” (5 minutes).

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TC_CHECK_INSTALLER_TIMESTAMP

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. The value of the TC_CHECK_INSTALLER_TIMESTAMP configuration variable determines whether the thin client compares the modification times of the installer files on the client and on the server. If this variable is set to “1” (on, true, yes) and the modification time of the client file is older than the time of the server file, the automatic update process is initiated. If the installer file does not exist on the client, then the comparison is made with the modification time of the thin client executable (**acuthin**) currently running. The default value for this variable is “0” (off, false, no).

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_CONTINUITY_WINDOW

If this configuration variable is set to “1” (on, true, yes), the Thin Client creates an invisible window after the next window is created by the COBOL application. This invisible window remains until the Thin Client shuts down.

This variable must be set in the application's configuration file or in the initialization code of the application so that it is applied to the initial window created by the application.

The invisible window is needed because the GetMessage API routine behaves differently under Windows 2000 and Windows XP than in older versions of Windows. If an application has no open windows and it calls GetMessage, the focus is transferred to another application. Once transferred, the application cannot force the focus to return to the original application, even if it subsequently creates a window to receive the focus. This situation arises under the Thin Client if the COBOL application destroys all of its windows before creating a new one.

Since this variable applies to a very specific situation, the default setting for TC_CONTINUITY_WINDOW is “0” (off, false, no). You must set it explicitly if you want to use this feature.

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TC_CONTROL_SYNC_LEVEL

This variable determines which VALUE data items in a Screen Section are updated when a BEFORE, AFTER or EXCEPTION procedure executes. (This variable only affects BEFORE, AFTER and EXCEPTION procedures. The values of all variables are made current anytime an ACCEPT terminates.) The possible values for TC_CONTROL_SYNC_LEVEL are:

1	(default) Only the VALUE data item associated with the current field is updated when its AFTER or EXCEPTION procedure executes.
2	Only the VALUE data item associated with the current field is updated when its BEFORE, AFTER or EXCEPTION procedure executes.
3	All VALUE data items are updated when executing any BEFORE, AFTER or EXCEPTION procedure.

For best performance, we recommend leaving this variable at its default setting of “1” unless that causes your program to perform incorrectly. In which case, you can increase the setting of TC_CONTROL_SYNC_LEVEL to “2” or “3” to adjust for problems in the application behavior.

Note: Alternatively, you can directly INQUIRE the value of a control in an embedded procedure. This allows you to tune application performance more precisely than TC_CONTROL_SYNC_LEVEL will allow.

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TC_DELAY_ACTIVATE

This variable determines precisely when the thin client sends CMD-ACTIVATE events to the server. Under the default setting of “1” (on, true, yes), the client delays sending the event until after the Windows notification routine receiving the event has completed. However, ActiveX events are never delayed. The alternate setting of “0” (off, false, no) sends the event to the server immediately when it is generated on the client.

We recommend leaving this variable at its default setting because the Windows API occasionally alters actions taken by the program when they occur within the scope of an activation notification. (For example, Windows will sometimes override a “set focus” call.) Delaying the COBOL program's response to the activation until after the Windows notification routine is complete avoids these alterations.

If you experience an unexplained difference in window activation when running under the thin client, try setting this variable to “0”. If this produces the desired behavior, the handling of the CMD-ACTIVATE events by the program is unusual and may not be performing as intended. For example, the EVENT procedure that handles the CMD-ACTIVATE event may be destroying an unrelated window instead of transferring control to the window issuing the CMD-ACTIVATE event.

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TC_DELAY_PRE_EVENT_OPS

This configuration variable applies only to the ACUCOBOL-GT Thin Client. Using this variable, you can direct the thin client to buffer some requests received from the server and process them later. When you set this variable to “1”, the thin client buffers the requests received between the time that the client sends an event to the server and the time that the server informs the client that it has started the related event procedure. The events are processed only after the event procedure starts in order to prevent the thin client from processing requests that generate more events before the first event procedure has started. The default value of TC_DELAY_PRE_EVENT_OPS is “0”.

Note: The buffering behavior described for this configuration variable was introduced as the default behavior in Version 6.1. Beginning with Version 7.2, the buffering behavior is turned off by default.

TC_DISABLE_AUTO_UPDATE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. You can disable the automatic update process by setting the TC_DISABLE_AUTO_UPDATE configuration variable to “1” (on, true, yes). The default value of this variable is “0” (off, false, no).

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_DISABLE_SERVER_LOG

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. If the thin client automatic update process fails for any reason, a log file may be created on the server. This file contains a log of the update operations and details about the failure. To prevent the creation of this log file, set the TC_DISABLE_SERVER_LOG configuration variable to “true” (1, on, yes). The default value of this variable is “false” (0, off, no).

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_DOWNLOAD_CANCEL_MESSAGE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. During the automatic update installer file download process, a progress dialog appears. You can cancel the download at any time from this dialog box. Use the TC_DOWNLOAD_CANCEL_MESSAGE configuration variable to specify the message that appears when the download is cancelled. The default value for this variable is

```
Please wait while the download is being cancelled . . .
```

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_DOWNLOAD_DESCRIPTION

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. During the automatic update installer file download process, a progress dialog appears. You use the TC_DOWNLOAD_DESCRIPTION configuration variable to specify the text that appears in the middle of the download progress dialog. Its default value is

```
Downloading installation file. . .
```

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_DOWNLOAD_DIALOG

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. During the automatic update installer file download process, a progress dialog appears. The default value of "1" (on, true, yes) for the TC_DOWNLOAD_DIALOG configuration variable allows the appearance of this dialog box. If you set this variable to "0" (off, false, no), the progress dialog does not appear.

For detailed information about the thin client automatic update process, refer to section 7.4, "Thin Client Automatic Update," in the *AcuConnect User's Guide.*

TC_DOWNLOAD_DIALOG_TITLE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. During the automatic update installer file download process, a progress dialog appears. The TC_DOWNLOAD_DIALOG_TITLE configuration variable is used to specify the title bar text in this dialog box. The default value of this variable is

```
ACUCOBOL-GT Thin Client Automatic Update
```

For detailed information about the thin client automatic update process, refer to section 7.4, "Thin Client Automatic Update," in the *AcuConnect User's Guide.*

TC_EVENT_LIST

This configuration variable lets you control which events your program receives, giving you more control over the rate of network traffic. It must be set in the configuration file and cannot be changed programmatically with the SET verb. It contains the numeric value of a single event type or a list of event types separated by non-numeric characters like spaces or commas. Whether your program receives these events depends on the value of TC_EXCLUDE_EVENT_LIST. If its value is "0", then your program receives the events listed in TC_EVENT_LIST. If TC_EXCLUDE_EVENT_LIST is set to "1", the events listed in TC_EVENT_LIST are not sent to your program.

TC_EXCLUDE_EVENT_LIST

The value of this variable determines whether the events listed in TC_AX_EVENT_LIST and TC_EVENT_LIST are sent to your program. A value of "1" means the specified events are not sent to your program. The default value is "0".

TC_INSTALLER_ARGS

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. The thin client uses the value of the TC_INSTALLER_ARGS configuration variable as the command-line options passed to the installer executable. For example, if you want "msiexec.exe" to log all of its operations to a file named "msi.log", then you could set TC_INSTALLER_ARGS to "/log msi.log". TC_INSTALLER_ARGS has no default value.

For detailed information about the thin client automatic update process, refer to section 7.4, "Thin Client Automatic Update," in the *AcuConnect User's Guide*.

TC_INSTALLER_CLIENT_FILE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. You use the TC_INSTALLER_CLIENT_FILE configuration variable to specify the path and file name of the installer file that you want to create on the client. The default value of this variable is

```
    <APPDATA>\ACUCOBOL-GT\<installer_server_filename>
```

where <APPDATA> is a special directory name for C:\Documents and Settings\<user>\Application Data and <*installer_server_filename*> is the file name specified in the TC_INSTALLER_SERVER_FILE configuration variable.

For detailed information about special directory names like <APPDATA> and about the thin client automatic update process, refer to section 7.4, "Thin Client Automatic Update," in the *AcuConnect User's Guide*.

TC_INSTALLER_RUN_ASYNC

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. You use the TC_INSTALLER_RUN_ASYNC configuration variable when you want to prevent the thin client from restarting after an automatic update or when your installer file handles the automatic update process to completion. When you set this variable to “1” (on, true, yes), the thin client starts the installer process asynchronously and then exits immediately. It does not wait for the automatic update process to complete and does not restart the application. The default value is “0” (off, false, no).

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_INSTALLER_SERVER_FILE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. You set the TC_INSTALLER_SERVER_FILE configuration variable to the path and file name of the server installer file. Its default value is

```
<runtime_path>/acuthin.msi
```

where <*runtime_path*> is the directory that contains the **runcbl** runtime executable.

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_INSTALLER_TARGET_DIR

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. You use the TC_INSTALLER_TARGET_DIR configuration variable to specify the location where you want the updated thin client to be installed. This variable has no default value.

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_INSTALLER_UI_LEVEL

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. The keywords or numeric values in the TC_INSTALLER_UI_LEVEL configuration variable control the Windows installer interface. Set TC_INSTALLER_UI_LEVEL to NONE or “0” if you do not want the Windows installer to display a user interface. Set this variable to UNATTENDED or “1” if you want the Windows installer to display informational and progress messages but to execute unattended. Set the variable to INTERACTIVE, DEFAULT, or “2” if you want the Windows installer to prompt for and accept user input for the installation process. Set the variable to REDUCED or “3” if you want to use a reduced user interface.

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide*.

TC_MAP_FILE

In thin client deployments, set this variable to point to the character map file that defines the mapping of international characters between client and server systems. A detailed description of international character handling is located in the *AcuConnect User’s Guide*, section 4.5, “International Character Handling.”

TC_NESTED_AX_EVENTS

This variable determines how thin client handles nested ActiveX events. Because thin client processes Windows messages while waiting for responses from the server, it is possible for new ActiveX events to be sent while still waiting for an earlier event procedure to return, causing event procedures to be nested within event procedures. Because nested event procedures can cause unpredictable results, including memory access violations (MAVs), this variable is set to “0” (off, false, no) by default. If you want to enable it, set it to “1” (on, true, yes).

TC_QUIT_MODE

This variable lets you control how your COBOL application shuts down when no client activity occurs during the interval defined by TC_CHECK_ALIVE_INTERVAL. Setting TC_QUIT_MODE to "-1" (the default value) shuts your program down according to the value chosen for the QUIT_MODE configuration variable. If you set this variable to "0", the runtime stops the program immediately.

When this variable is set to a value greater than "0" (up to "32767"), your application has a program-controlled exit. When the runtime determines that the thin client is no longer responding (no user interaction and no pings during **TC_CHECK_ALIVE_INTERVAL**), the MSG-MENU-INPUT event is sent to the program's main window and EVENT-DATA-2 contains the value defined by TC_QUIT_MODE. Your program can detect this in the main window's event procedure and you can perform whatever code you desire. At this point there is no connection to the thin client, so user interface operations may not be performed. You must end your shutdown code with "STOP RUN" to terminate the runtime.

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TC_REQUIRES_BUILD_NUMBER

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. When the thin client executes, it compares its build number with the value of the TC_REQUIRES_BUILD_NUMBER configuration variable. If the value of this variable does not match the client's build number, the automatic update process is initiated. Set this variable to the thin client build number required by the application. The default value of this variable is "0" (off, false, no).

For detailed information about the thin client automatic update process, refer to section 7.4, "Thin Client Automatic Update," in the *AcuConnect User's Guide*.

TC_RESTRICT_AX_EVENTS

This variable controls whether the application will ignore ActiveX events between ACCEPT statements (the termination of one ACCEPT and the beginning of the next). Setting this variable to “1” (on, true, yes) enables this behavior. The default value is “0” (off, false, no).

Ordinarily, the thin client runtime suspends all ActiveX events when the application is not processing an ACCEPT statement. However, some ActiveX controls do not support the ability to suspend and resume events when an application is not processing an ACCEPT statement. As a result, in a thin client environment, an event procedure may be run unexpectedly during a CREATE, DISPLAY, MODIFY, INQUIRE, or any other operation that waits for results from the thin client. Setting TC_RESTRICT_AX_EVENTS provides some control over these ActiveX events.

To determine if a particular ActiveX control supports suspending and resuming events, check the control’s documentation or ask the control vendor. Note that the control must implement the “IOleControl::FreezeEvents()” function.

For more information about ActiveX control handling, see Chapter 4 in *A Guide to Interoperating with ACUCOBOL-GT*, and section 6.3 of the *AcuConnect User’s Guide*.

TC_SERVER_LOG_FILE

This configuration variable applies only to the ACUCOBOL-GT Thin Client automatic update feature. If the thin client automatic update process fails for any reason, a log file may be created on the server. This file contains a log of the update operations and details about the failure. The TC_SERVER_LOG_FILE configuration variable can be used to configure the location and name of that log file. The name can optionally include the hostname of the client machine and the process ID of the server runtime that was managing the automatic update at the time of the failure.

By default, this file is named “autoupdate.%c.%p.log”, where “%c” is replaced by the client hostname and “%p” is replaced by the process ID of the server runtime. The default location is the working directory specified in the alias on the server. Note that the directory must exist at the time of the failure for the log file to be created.

For detailed information about the thin client automatic update process, refer to section 7.4, “Thin Client Automatic Update,” in the *AcuConnect User’s Guide.*

TC_SERVER_TIMEOUT

This variable lets you determine how many seconds (from “0” to “32767”) the thin client waits for a response from the server. If the thin client receives no response from the server in the specified time period, the following message box appears:

```
The remote host is not responding.
Press OK to close this program.
Press Cancel to wait another %s seconds.
```

where “%s” is the value of TC_SERVER_TIMEOUT. The default value is “20”.

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TC_TV_SELCHANGING

This variable is designed for thin client applications. It provides some control over when the runtime generates Msg-Tv-Selchanging events for tree view controls. Because most applications that use tree view controls do not process Msg-Tv-Selchanging events, the thin client suppresses its generation in some cases. This improves both performance and stability. TC_TV_SELCHANGING recognizes the following values:

0	never generate Msg-Tv-Selchanging events
1	(default) generate Msg-Tv-Selchanging events when the selection is about to change due to the user using the mouse or the keyboard to change to current selection
2	always generate Msg-Tv-Selchanging events

The default setting of “1” allows you to detect user-initiated events in your program while filtering out many other causes of the event.

If you know your program doesn’t handle any Msg-Tv-Selchanging events, you can set TC_TV_SELCHANGING to “0” to entirely inhibit generation of the event. This can slightly improve performance.

If TC_TV_SELCHANGING is set to “1” and your program experiences odd behavior with tree view controls under the thin client, you can try setting the variable to “2” to generate all Msg-Tv-Selchanging events. This setting can help you determine whether a Msg-Tv-Selchanging event is the cause of the odd behavior. If this setting eliminates the odd behavior, it indicates that your program relies on Msg-Tv-Selchanging events in cases other than the user initiating a selection change.

For more information about the thin client, refer to the *AcuConnect User's Guide*.

TEMP_DIR

This variable lets you specify where certain temporary files used by the ASSIGN clause will be created on VAX/VMS systems. These temporary files are created when you use the %TMP% option for assigning a file to a simulated pipe with “-P”. For more information, see the *ACUCOBOL-GT User’s Guide*, section 2.8, “File Name Interpretation.”

TEMPORARY_CONTROLS

By default, graphical controls are created as permanent controls. By setting this configuration variable to “1” (on, true, yes), you cause controls to be created *temporary* by default. This is useful when you are converting older programs that assume that a screen update will overwrite any existing screen data. You can make individual controls permanent or temporary explicitly by using the PERMANENT and TEMPORARY styles (see section 5.2 in Book 2, *ACUCOBOL-GT User Interface Programming*).

TEXT

This configuration variable controls the text of runtime messages. The ACUCOBOL-GT runtime system displays a number of informational and warning messages to the end user. Several of these messages can be customized via entries in the configuration file.

For each message that you want to change, place the word “TEXT” in your configuration file, followed by a message number from the list below, an “=” sign, and then the text you would like to use.

For example, the standard message #1 is “press return”. You can change that message to “push enter” by placing this line in your configuration file:

```
TEXT  1=push enter
```

Note: There is no space before or after the equals sign, and the new message is not in quotes.

These are the standard runtime messages and their numbers:

Message #	**Text**
1	"Press return"
2	"Number required"
3	"Entry required"
4	"Field must be filled with data"
5	"Too many hot keys active"
6	"Program missing or inaccessible"
7	"Not a COBOL program"
8	"Corrupted program"
9	"Inadequate memory available"
10	"Unsupported version of object code"
11	"Program already in use"
12	"Too many external segments"
13	"Large-model program not supported"
18	"Please end this application first"
19	"Japanese objects not supported" This message is displayed when a standard runtime attempts to execute an object that contains Japanese COBOL extensions.
20	"Too many lines" This message is displayed when the user exceeds the MAX-LINES setting for a multiline entry field.
21	"License manager (acushare) not running" This message is displayed when acushare is not running and the runtime is unable to start it (e.g., because it is not in the path).
22	"Data must fit this format:" This message is displayed when the user enters illegal data when using the NUMERIC_VALIDATION configuration option.
23	"&Ok"
24	"&Yes"
25	"&No"

Message #	Text
26	“&Cancel”
	Messages 23, 24, 25, and 26 are used by character-based versions for the message box facility.
28	“Unable to access the file “%s” due to heavy usage by other users. Would you like to continue waiting for it?”
	(See the configuration variable WAIT_FOR_FILE_ACCESS for more information about this message.)
30	“Connection refused - perhaps AcuConnect is not running”
31	“Please enter a value between %ld and %ld”
	This message is displayed when the user enters a value outside of the allowed range for an entry-field (see MIN-VAL/MAX-VAL in the entry-field reference). The first “%ld” is replaced by the MIN-VAL setting. The second “%ld” is replaced by the MAX-VAL setting. You may omit these if you desire. Note that the second character in the sequence is the letter “l”, and not the number one (“1”).
32	“Program contains object code for a different processor”
33	“Incorrect serial number”
34	“Connection refused - user count exceeded on remote server”
35	“License error”
36	“The remote host is not responding.\nPress OK to close this program.\nPress Cancel to wait another %s seconds.
	This message is displayed when Thin Client does not receive a response from the server in the number of seconds specified in TC_SERVER_TIMEOUT.
	Use “\n” to separate lines and “%s” to substitute the number of seconds (value of TC_SERVER_TIMEOUT). If you don’t want to display the number of seconds, omit the “%s”.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

TRACE_STYLE

This variable allows you to customize the format of error and trace messages. You can set it to the sum of one or more of the following values:

0	The default. No “ACU” prefix, process ID, time, or date is included in the trace output.
1	Adds “ACU” prefix to each line of the trace output.
2	Adds the process ID.
4	Adds the time.
8	Adds the microseconds; has an effect only if “4” is also specified.
16	Adds the date.

You can also set TRACE_STYLE to one of the following keywords, which correspond to the indicated numerical values:

NONE	0
TIMESTAMP	12 — The TIMESTAMP style is 4+8; it outputs timestamps with microseconds.
APPSERVER	23 — The APPSERVER style is 1+2+4+16; at the beginning of each line of the error file it outputs “ACU” followed by the date, the process ID, and the time without microseconds.

TRANSLATE_TO_ANSI

This variable has meaning only on graphical systems such as Windows. It is used only if:

- you are using the graphical system’s font to accept data, and
- you store your data using the OEM character set. (For example, Vision files may contain OEM characters if they were created with a DOS runtime.)

Set the variable TRANSLATE_TO_ANSI to “1” (on, true, yes) to turn on a character set translator. Then, if you use the graphical system’s font for accepting data, the runtime will translate from one character set to the other for you. Data that is accepted from the screen will be translated into the OEM character set before it is stored on disk. Data stored in the OEM character set will be translated to the ANSI character set before it is displayed on screen. This also applies to the printer, if you are using Windows spooling and the printer uses an ANSI font.

Setting TRANSLATE_TO_ANSI to the default, “0” (off, false, no), turns off the translation process.

This variable can be set from within a COBOL program with the SET verb. For example:

```
SET ENVIRONMENT “TRANSLATE_TO_ANSI” TO “YES”.
```

Note on ANSI and OEM characters:

The ANSI and OEM representations of the following standard English characters are identical:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Abcdefghijklmnopqrstuvwxyz
0123456789 <space>
! " # $ % & ' ( ) * + , - . / :
; < = > ? @ [ \ ] ^ _ ‘ { | } ~
```

Only the representations of accented vowels and other special or non-English characters are different.

TREE_ROOT_SPACE

This variable controls the number of screen columns between the left edge of the Tree-View control and the root level text. TREE_ROOT_SPACE is used only with the LINES-AT-ROOT property. If LINES-AT-ROOT is not specified, the root level item text will be displayed starting at the leftmost screen column inside the tree-view control.

For example, if TREE_ROOT_SPACE is set to 5, there will be 5 screen columns before the text of each root level item. The screen column where the root level line will be drawn is determined by this formula:

```
    root level-line = (TREE_ROOT_SPACE - 1)/2 + 1
```

Taking off from the previous example, if TREE_ROOT_SPACE=5, the root level line will be drawn in screen column 3, counting from the left edge of the Tree-View control.

This has the effect of centering the vertical root level line in the space between the left edge of the Tree-View control and the last root level text.

The “+” or “-” button is displayed in the column to the right of this vertical line if the TREE_ROOT_SPACE is set to a value greater than or equal to 2. If the TREE_ROOT_SPACE is set to 1, the “+” or “-” button appears in the first screen column of the Tree-View control. The default value of TREE_ROOT_SPACE is 2.

TREE_TAB_SIZE

This configuration variable is one of two variables that affect the appearance of character-based Tree-View controls. TREE_TAB_SIZE controls the number of screen columns between each level in the visual representation of the tree. For example, if TREE_TAB_SIZE is set to 10, the horizontal distance between the first character of text in the first level and the first character of text in the succeeding levels of the tree will be 10 screen columns each. The default value of TREE_TAB_SIZE is 3.

See **TREE_ROOT_SPACE** variable.

TRX_HOLDS_LOCKS

This configuration variable allows you to control which locks are released at the end of a transaction. If this variable is set to “1” (on, true, yes), then locks set using the READ statement that are not specifically released or replaced by extended transaction locks (for example, by a REWRITE) are held at the end of the transaction. Locks are released during a transaction by any operation that would ordinarily release them, unless those locks were replaced by extended transaction locks.

If TRX_HOLDS_LOCKS is set to the default, “0” (off, false, no), then locks are released at the end of a transaction, and the UNLOCK verb has no effect during a transaction.

UPPER_LOWER_MAP

This variable allows you to define which upper-case characters correspond to which lower-case characters, for characters outside of the standard ASCII character set (those whose underlying decimal values are 128 or larger).

You might find this useful if you are experiencing problems with the UPPER or LOWER option of the ACCEPT statement when non-standard characters are entered (such as an “e” with an accent above it). The ACUCOBOL-GT runtime system relies heavily on C library routines to handle conversions between upper-case and lower-case characters. On many machines, these routines do not handle characters outside of the standard ASCII character set correctly.

To specify corresponding characters, use UPPER_LOWER_MAP followed by pairs of characters, where the first character is the upper-case version and the second character is the lower-case version. Separate the characters by a space. Describe the characters either by typing them at the keyboard or by entering the decimal value that represents them.

For example, on a standard IBM PC, the video card represents an upper-case “U” with an umlaut (Ü) as character 154, and the lower-case “u” with an umlaut (ü) as 129. The upper-case “E” with an accent character is 144 (É) and the lower-case “e” with an accent is 130 (é). To express this in the configuration file, you would add the following line:

```
    UPPER_LOWER_MAP  154 129 144 130
```

This could be extended to include all of the character pairs available.

By default, Windows systems come with the UPPER_LOWER_MAP defined to be the character pairs available on the standard video cards produced by IBM. Note that using "code pages" can change this, so the default may not work in all cases for these machines. For other machines, the default is empty (which means that C library routines are used for conversion). If you experience difficulties, UPPER_LOWER_MAP allows you to define a mapping that reflects your hardware configuration.

Only characters whose decimal values are 128 or greater may be mapped by this technique.

Note: This variable *cannot* be read with the ACCEPT FROM ENVIRONMENT statement.

USE_CICS

Set this variable to indicate to the runtime that the program makes calls to the CICS interface. When USE_CICS is set to "1" (on, true, yes), the runtime attempts to pass calls to functions that begin with the string "CICS" to the CICS interface. If the named routine does not exist, the runtime uses the normal search sequence to find a matching function. When USE_CICS is set to the default value of "0" (off, false, no), the runtime does not perform any special handling.

USE_EXECUTABLE_MEMORY

When set to "TRUE", this variable enables a COBOL program compiled for Native Code to run on a Windows machine that has Data Execution Protection (DEP) enabled for all processes. The default value is "FALSE".

USE_EXTSM

Set this variable to indicate that the runtime should use an external sort module. When USE_EXTSM is set to “1” (on, true, yes), the runtime uses the linked-in EXTSM function to perform the SORT or MERGE operation. When USE_EXTSM is set to the default value of “0” (off, false, no), the runtime does not perform any special handling for SORT and MERGE verbs.

USE_LARGE_FILE_API

On UNIX systems, this variable allows you to turn on or off file system API support for very large files (greater than 2 gigabytes). Support for large files is enabled when USE_LARGE_FILE_API is set to “1” (on, true, yes). Some UNIX systems do not support files greater than 2 gigabytes in size. In those situations, setting this variable to the default of “0” (off, false, no) causes the runtime to use the standard 32-bit file system API. This variable has no effect on Windows platforms.

USE_LOCAL_SERVER

This variable is used by the runtime and Web Runtime to specify whether or not you want to run client applications on the same machine as an AcuServer file server. When USE_LOCAL_SERVER is set to the default of “0” (off, false, no), AcuServer is bypassed when accessing local files that have remote name notation. The remote name is stripped off and the file I/O operation is handled by the runtime or Web Runtime. Set this variable to “1” (on true, yes) to use AcuServer to access local files that have remote name notation. This variable only works with AcuServer client runtimes and AcuServer client Web Runtimes.

USE_MPE_REDIRECTION

This configuration variable applies only when running in HP COBOL compatibility mode (with the “-Cp” compiler option) on machines that support the MPE environment. With the use of the USE_MPE_REDIRECTION configuration variable, input for an ACCEPT statement is read from the file specified by STDIN=, and output from a DISPLAY statement is written to the file specified by STDLIST= on the RUN command line. To enable this behavior, set USE_MPE_REDIRECTION to “1” (on, true, yes). The default value is “0” (off, false, no). In addition, when this variable is set, no terminal manager escape sequences are written to the redirected output file.

USE_MQSERIES

Use this variable to indicate to the runtime that the program makes calls to WebSphere MQ (formerly MQSeries). When USE_MQSERIES is set to “1” (on, true, yes), the runtime attempts to pass calls to functions that begin with the string “MQ” to the WebSphere MQ interface. If the named routine does not exit, the runtime uses the normal search sequence to find a matching function. When USE_MQSERIES is set to the default value of “0” (off, false, no), the runtime does not perform any special handling.

USE_SYSTEM_QSORT

This variable instructs the runtime SORT routine to use the system qsort() function, rather than the built-in sort function. Set USE_SYSTEM_QSORT to “1” if you want to use the system qsort() function. The default value is “0” and results in the use of the built-in sort function.

Some systems have qsort() functions that perform better than the built-in function. Consider experimenting with this variable’s settings to determine if this option yields better performance on your system. Pay particular attention to the number of comparisons done during the sort, which can be seen in the runtime trace output.

USE_WINSYSFILES

This variable specifies whether the runtime should recognize calls to modules with the extensions “.drv” and “.ocx” as well as those with the extension “dll”. By default, it is set to “1” (on, true, yes).

For backwards compatibility, you can turn this feature off by setting it to “0” (off, false, no). Then, only calls to “.dll” files are supported.

V_BASENAME_TRANSLATION

This variable allows you to tell Vision whether to include full path information in the filename. By default, only the base name is included (the filename with no extension and no path information). Retaining the path information can be helpful in instances where Vision files of the same name are stored in different locations and you want to map one of the segments from one directory to a new location.

When V_BASENAME_TRANSLATION is set to “0” (off, false, no), Vision uses the entire path of the file. When it is set to “1” (on, true, yes), the default setting, Vision uses only the base name.

The setting of V_BASENAME_TRANSLATION affects the behavior of three configuration variables that handle Vision filename translation: *filename*, *filename*_DATA_FMT, and *filename*_INDEX_FMT. The following illustrates how the configuration variables interact.

For the file “/user/data/record1.vix”:

- If V_BASENAME_TRANSLATION is set to “on” (the default), *filename*, *filename*_INDEX_FMT, and *filename*_DATA_FMT use “RECORD1_VIX” as the base name.
- If V_BASENAME_TRANSLATION is set to “off”, *filename*, *filename*_INDEX_FMT, and *filename*_DATA_FMT use “_USER_DATA_RECORD1_VIX” as the base name (underscores replace instances of “/” and “.”).

For a description of *filename*, *filename*_INDEX_FMT, and *filename*_DATA_FMT, see their respective entries in this appendix.

V_BUFFERS

This variable sets the number of indexed block buffers to allocate. These buffers are used to improve the performance of indexed files. Each buffer is 512 bytes plus some overhead. Increasing the number of buffers can improve file performance. Decreasing the number conserves memory. The value of V_BUFFERS has no effect on versions of ACUCOBOL-GT that do not use Vision files. The value of V_BUFFERS can range from zero (no buffering) to 2097152. The default value is 64.

V_BUFFER_DATA

The setting of this variable determines whether or not Vision indexed file data blocks (as opposed to key blocks) will be held in the memory-resident disk buffers. When it is set to "1" (on, true, yes), both data blocks and key blocks will use the buffers. When set to "0" (off, false, no), only key blocks will use the buffers. Setting this value to "1" will usually improve performance unless very few buffers are being used.

Note: Holding data blocks in the buffers slightly increases the chances of losing data if a file opened for MASS_UPDATE is not closed properly (power failure, etc.). The default setting of this variable is "1".

V_BULK_MEMORY

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 option. The default size of this buffer is 1 MB.

Note: 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.

To change the size of the allocated memory buffer to, for example, 500 KB, you would enter:

```
V_BULK_MEMORY = 500 KB
```

V_FORCE_OPEN

This variable allows you to force the runtime to open broken files that would normally cause an error 98. This means you can write COBOL programs to recover these files in ways that are not available with **vutil**. Set V_FORCE_OPEN to "1" (on, true, yes) to open the files. The default is "0" (off, false, no).

Note: When this variable is set to "1", make sure you do not also have the **V_OPEN_STRICT** variable set to "1" because the settings conflict.

V_INDEX_BLOCK_PERCENT

This configuration variable allows you to specify index pre-allocate and extension factors as a percentage of the factors applied to the data segment. In Vision 4 and 5 files, the index data contained in the index segments is often much smaller than the record data contained in the data segments. As a result, a large pre-allocate or extension factor typically allocates many more index blocks than are needed. This can be undesirable, especially if disk space is tight.

Setting V_INDEX_BLOCK_PERCENT to a number less than 100 causes fewer index blocks than data blocks to be created. Setting the variable to a number greater than 100 causes more index blocks than data blocks to be created. The valid range for V_INDEX_BLOCK_PERCENT is one through 1000. If the value specified is less than one, it will be promoted to one. V_INDEX_BLOCK_PERCENT is set to 100 by default (the default pre-allocate and extension factors for a file).

For example, if a file has an extension factor of 10, setting V_INDEX_BLOCK_PERCENT to 50 causes 10 new data blocks and five new index blocks to be created the next time the file is extended. Setting V_INDEX_BLOCK_PERCENT to 200 causes 10 new data blocks and 20 new index blocks to be created the next time the file is extended.

Note: The number of blocks pre-allocated will never be larger than that which can fit in the initial data and index segments. If the pre-allocation value specified or calculated from V_INDEX_BLOCK_PERCENT is larger than the segment size, the pre-allocation amount is automatically reduced to the segment size.

V_INTERNAL_LOCKS

This configuration variable allows you to control whether the runtime enforces internal record or file locking. When V_INTERNAL_LOCKS is set to "0" (off, false, no), Vision tracks locks but does not enforce internal record or file locking. As a result, the runtime does not return a record or file locked condition for a record or file that was previously locked by the same run unit. When V_INTERNAL_LOCKS is set to the default of "1" (on, true, yes), internal record and file locking are enforced.

Note: The Windows operating system enforces a single lock per process on a region of a file. This means that if your program opens the same physical file as two different logical files and then tries to lock the same record in both "files", the second lock will fail (with an error "99") even if V_INTERNAL_LOCKS is set to "0". So V_INTERNAL_LOCKS 0 practically affects programs running on UNIX operating systems only.

V_LOCK_METHOD

This variable selects which locking method Vision will use to control simultaneous access to indexed files. It affects only the Vision file system, and only files directly accessed by the runtime (it does not apply to files accessed via AcuServer).

The default setting of "0" (zero) causes Vision to lock the first byte of the file for every access to the file (both reads and updates). This ensures that the process is not interfered with by another process. This locking method is always used by Vision Version 2 files.

Setting this variable to “1” causes Vision to lock the first byte of the file for all operations except random READs or READ NEXTs. These two operations proceed without the lock. Instead they perform some additional reads of the file, to ensure that they get consistent results. If they get inconsistent results, they are retried, this time locking the first byte as other operations do. This locking method is available only for Vision Version 3, 4, and 5 files.

Note: This variable must have the same setting for all the runtimes accessing a file, whether they are reading or writing to it. For example, if a runtime set with V_LOCK_METHOD=1 is reading from a file, any runtimes that are writing to that same file must also have V_LOCK_METHOD set to 1.

Lock method “1” can produce better performance on some machines. These machines fall into two categories:

- Machines that take a long time to place a lock.
- Machines that do not queue lock requests, and are very busy. In this case, some users typically get good performance, while others get poor performance.

Setting V_LOCK_METHOD to “1” might help improve performance with Vision Version 3, 4, or 5 files. For example, setting V_LOCK_METHOD to “1” can be helpful on some Windows networks. A peer-to-peer network of Windows 98 machines can exhibit problems reading Vision files when a process performs a tight read loop. The problem usually surfaces as either an error 30,33 or an unexpected error 99. This occurs because the runtime is unable to place a lock on the header of the file after 400 attempts over a 20-second period. For other networks, setting V_LOCK_METHOD to “1” can substantially reduce the number of lock requests made by the runtime and can often resolve these problems.

To get statistics about header locks, select Trace Files level “3” in the debugger (for example, “TF 3”). These statistics print on the runtime’s error output each time a Vision file is closed. They cover the operations in that file since it was last opened. You can also view these statistics (without the full trace) by adding “256” to the lock method chosen (for example, setting V_LOCK_METHOD to “257” selects method “1” and prints statistics).

Setting the V_LOCK_METHOD variable to “2” enables “asynchronous reads” of Vision files. This option is intended to further reduce the number of file locks required to perform random READs and READ NEXTs.

The advantage of the “2” setting is that it is less likely to require retrying a READ with a lock when a file is undergoing heavy modification. With V_LOCK_METHOD=1, the READ is retried with a lock whenever it detects that the file has been updated in any way; with V_LOCK_METHOD=2, the READ is retried only when Vision encounters inconsistent data while traversing the index tree or reading the record data. This leads to less locks and therefore greater performance for machines with slow locking functions.

V_LOCK_METHOD=2 works only for Vision 4 and 5 files. A fundamental requirement for the V_LOCK_METHOD=2 feature to work properly is that the operating system must provide atomic write operations. That is, if one process is writing to a file, another process will always see the contents of the file as it exists either before or after the write operation, never the intermediate contents as the write operation runs. There is evidence that Linux does not provide atomic writer operations and therefore it is not recommended to use this setting in a Linux environment.

If any process reading a particular file is using V_LOCK_METHOD=2, all other processes (runtimes) updating that file must be ACUCOBOL Version 5.0.0 or greater. This is because Version 5.0.0 contains changes that affect the way Vision updates the tree structure of its files. These changes allow for greater consistency of the tree from the viewpoint of an asynchronous reader. This requirement is not enforced by Vision, however, so it is important for the users to pay careful attention to the versions of programs accessing their files to avoid receiving erroneous data. Therefore, before enabling this option, make sure that all runtimes updating files on which asynchronous reads are to be performed (V_LOCK_METHOD=2) are Version 5.0.0 or later.

As with V_LOCK_METHOD=1, adding 256 to the value of the V_LOCK_METHOD setting causes statistics about header locks to be printed to the runtime’s error output each time a Vision file is closed. So, setting V_LOCK_METHOD=258 selects method 2 and turns on the header lock statistics.

V_MARK_READ_CORRUPT

This variable allows you to configure Vision so that it does not mark a file as broken if it encounters a corruption during a read or start operation. The effect is that the user is allowed to retry the program. This may be useful when the error is spurious (for example due to a network caching glitch). If the user retries the program and once again receives a file-corrupt message, then the file should be rebuilt or recovered normally. To enable this option, set the configuration option “V_MARK_READ_CORRUPT” to “0” (off, false, no). The default setting is “1” (on, true, yes).

V_NO_ASYNC_CACHE_DATA

This configuration variable turns on the caching of data blocks for file reads. By default, Vision 4 and 5 do not cache data blocks in its internal cache (all V_BUFFERS are allocated only to index blocks). This is required for the asynchronous reads feature (V_LOCK_METHOD=2) to work properly (each data record needs to be read/written in a single system call).

The default setting of this configuration variable is “0” (off).

If you are *not* using the asynchronous reads feature *at all*, you may turn on the caching of data blocks by setting the V_NO_ASYNC_CACHE_DATA configuration variable to “1”. This may improve READ performance.

Caution: Be certain that you do not use this configuration variable with V_LOCK_METHOD=2 in any combination, as silent data corruption may result.

V_OPEN_STRICT

By default, Vision allows OPEN INPUT on files that are marked as broken. This behavior is intended to make it easier to recover records from broken files. If you want to receive an error status when opening a file marked as broken for INPUT, set V_OPEN_STRICT to “1” (on, true, yes). The default setting of “0” (off, false, no) allows open input on broken files.

Note: When this variable is set to “1”, make sure you do not also have the **V_FORCE_OPEN** variable set to “1” because the settings conflict.

V_READ_AHEAD

Setting this configuration variable to “0” (off, false, no) turns off Vision’s read-ahead logic. This may improve performance in cases where highly random file processing is being used. The default value is “1” (on, true, yes).

V_SEG_SIZE

This configuration variable sets the maximum size of a Vision 4 or 5 file segment in bytes. The default value is 2,147,482,112 (i.e., 2GB – 1536), except on older HP/UX machines where it is 1,073,740,288 (i.e., 1GB – 1536) due to an operating system limitation. You may not use larger values, but you can set smaller ones. The default value is the maximum allowed. The value specified will automatically be rounded down to a multiple of the block size of the file being created. For example, if the default V_SEG_SIZE value is used and a file with a block size of 1024 is created, the segment size for that file will be 2,147,481,600 (i.e., 2GB – 2048).

Using a smaller value for the segment size can help if you do not have 2GB free on any disk or for testing purposes. The minimum value allowed is 81,920 bytes. To minimize the number of files created, you should set this value as high as possible.

The segment size of a file is set at file creation time and cannot be modified without recreating the file (i.e., using **vutil –rebuild** with a different V_SEG_SIZE setting). **vutil** uses this variable, but since it does not use a configuration file, this variable must be set in the environment.

V_STRIP_DOT_EXTENSION

The V_STRIP_DOT_EXTENSION variable determines whether or not Vision strips a trailing "dot extension" (".dat") from the logical name of a data file when generating file names for index and data segments (other than the first data segment). Setting this variable to "0" prevents the extension from being removed. For example, by default, the first index segment name for the logical file "file.one" is "file.vix" (which would conflict with the index segment of "file.two"). When V_STRIP_DOT_EXTENSION is set to "0" (off, false, no), the index segment name is "file.one.vix". The default value for this variable is "1" (on, true, yes).

Note: The setting of this variable affects the behavior of four configuration variables: *filename*, *filename*_DATA_FMT, *filename*_INDEX_FMT, and *filename*_VERSION. See their respective entries in this appendix for details.

V_VERSION

This variable specifies the version number of new Vision files that are created. The default value is "5", which produces Vision files in the format of the current version (Version 5). The value "4" produces Version 4 files. Version 5 and 4 files are generated in a dual file format, with data records filed in one segment and overhead key information filed in another. The value "3" produces Version 3 files, in which data and keys are stored in a single file. The value "2" produces Version 2 files. Any value other than "2", "3", or "4" produces Version 5 files.

V23_GRAPHICS_CHARACTERS

Programs written for and executed with UNIX versions of the runtime up to and including Version 2.4.0 use hex values 1-8 to display line drawing characters on the screen. Runtimes after Version 2.4.0 use hex values offset by one (1). When older programs are used with runtimes released after Version 2.4.0, line drawing characters do not display as expected. To use the old values for line drawing characters, set this variable to “1” (on, true, yes).

If the variable is set to “0” (off, false, no) or is not set at all, the runtime will use the newer offset values. This variable works only for UNIX systems.

V30_MEASUREMENTS

This configuration variable affects whether the runtime sizes certain controls according to the rules from Version 3.0 or from the current version. If the current measurement code is causing your application to display incorrectly, then setting this variable to “1” (on, true, yes) will use Version 3.0 sizing rules instead. When V30_MEASUREMENTS is set to the default “0” (off, false, no), then the current sizing rules are in effect.

The related configuration variables, V31_MEASUREMENTS and V32_MEASUREMENTS have the same effect of setting the sizing rules to that of their respective versions.

V31_FLOATING_POINT

This configuration variable allows you to disable a correction that was made to the way floating-point numbers are displayed. Because some loss of precision in the display of “USAGE DOUBLE” fields was possible in Version 3.1, an improvement was introduced. Setting this variable to “1” (on, true, yes) means that the Version 3.1 method of displaying floating-point numbers is used. When V31_FLOATING_POINT is set to the default “0” (off, false, no), then the correction is in effect.

V42_FLOATING_POINT

This variable affects how floating-point arithmetic is performed. Starting with Version 4.3, floating-point arithmetic was enhanced to more closely reflect the way that floating-point values are determined on the host system. This enhancement can affect the behavior of existing programs. To revert to the computation method used prior to Version 4.3, set the value of V42_FLOATING_POINT to “1” (on, true, yes). By default, this variable is set to “0” (off, false, no).

V43_PRINTER_CELLS

This variable affects whether the runtime sets the width of a printer cell according to the rules from Version 4.3 or from the current version. Version 4.3 (and prior versions) computed the width of a printer cell based on the average width of a selected printer font. The width of a printer cell is currently computed in the same way that cells are computed for the screen, namely by the width of the “0” character. For fixed-width fonts, such as Courier, these values are the same for all characters. For proportional fonts, such as Times New Roman, some characters might be wider than the “0” character.

If the current computation is causing your application to print incorrectly, then setting this variable to “1” (on, true, yes) will use Version 4.3 rules instead. When V43_PRINTER_CELLS is set to the default “0” (off, false, no), then the current rules are in effect.

V52_BITMAP_BUTTONS

If some event in the system forces the focus away from a bitmap-based push button after a click has been started but not finished, this variable determines whether the click is voided. If you do not want the click to be voided, set this variable to “1” (on, true, yes). The default setting is “0” (off, false, no).

V52_BITMAPS

This variable determines whether your application uses device-dependent or device-independent bitmaps for image processing. The following settings are recognized:

1	Use Version 5.2 and earlier image-processing code (device-independent) for bitmap controls.
0	Use Version 6.0 and later image-processing code (device-dependent) for bitmap controls.
-1	(default) Dynamically apply the image-processing code based on the program's object semantics. For programs compiled for pre-Version 6.0 semantics, use the older imaging code. For programs compiled for Version 6.0 or later semantics, use the newer code.

V52_GRID_GOTO

This configuration variable determines how the runtime behaves when a user clicks in a grid control cell containing the cursor. Prior to Version 5.2, the runtime would not pass a MSG-GOTO-CELL-MOUSE event to the program when the user clicked in a grid cell containing the cursor. For programs compiled with Version 5.2, or later, this event *is* passed to the program. Setting V52_GRID_GOTO to "0" (off, false, no), maintains the pre-5.2 behavior. The default of "1" (on, true, yes) enables the new behavior, even for programs compiled with Versions 5.1 or earlier and run with Versions 5.2 or later. See Appendix C: Changes Affecting Previous Versions, in the *ACUCOBOL-GT Appendices Manual* for more details.

V60_LIST_VALUE

This variable allows you to select the algorithm used by the runtime to match a list box or combo box VALUE with an item in the control's list.

Prior to Version 6.0, setting the VALUE of a combo box or list box caused the first item in the list that started with the value of VALUE to be selected, regardless of case. Beginning with Version 6.0, when a box's VALUE is set, the list is searched for an exact, case sensitive match with the specified value. If the value is found, it is selected. If an exact match is not found, the list is searched for an exact match regardless of case. If a match is still not found, the list is searched again, this time for the first string that contains the passed VALUE as a leading substring, regardless of case. V60_LIST_VALUE allows you to specify which algorithm to use. It accepts the following values:

1	directs the runtime to use the Version 6.0 search algorithm
0	directs the runtime to use the pre-6.0 search algorithm (substring search only)
-1	(default) directs the runtime to use the 6.0 search algorithm on objects compiled for Version 6.0 or later, and to otherwise use the old search algorithm. This means that objects compiled for compatibility with versions prior to 6.0 that are run with a Version 6.0 runtime will not exhibit the new behavior.

V62_MAX_WINDOW

Starting with Version 7.0, when the runtime reduces the size of a window to fit the screen, it includes any fractional lines and columns that fit, provided the COBOL program attempts to create a window with fractional lines and columns. For example, if you create a 70.0 line window, but only a 66.4 line window fits on the display, the runtime detects that no fractional lines were attempted, and truncates the number of lines to 66.0. However, if you attempt to create a 70.1 line window, the runtime recognizes the fractional measurement and displays a 66.4 line window. To preserve the pre-7.0 behavior, set the configuration variable V62_MAX_WINDOW to “1” (on, true, yes) and fractional lines and columns are always removed. The default value is “0” (off, false, no).

V71_ALIGNED_ENTRY_FIELD

Starting with Version 7.2, the wheel mouse can be used for scrolling in a center- or right-aligned entry field. To preserve the pre-7.2 behavior, set the V71_ALIGNED_ENTRY_FIELD configuration variable to “1” (on, true, yes). The default value of this variable is “0” (off, false, no).

V71_FONT_WIDTHS

Windows has a function called GetTextMetrics that returns information about a font. This data is used by the runtime to compute the “maximum character width” and “wide character width” of a font. The “maximum width” amount is used to set a lower bound for how small an entry field can be (to ensure that at least one character is always visible). The “wide width” is used to scale small entry fields and uppercase entry fields. The “wide width” is computed by averaging the maximum and average character widths. Experimentation has shown that the “maximum character width” data returned may be inaccurate, sometimes by very large margins.

With the use of the V71_FONT_WIDTHS configuration variable, the runtime validates the data returned by the Windows function and corrects it when it is too large. The change does not affect programs until they are recompiled with Version 7.2 or later, or the change is specifically enabled through the V71_FONT_WIDTHS configuration option. The variable can have the following values:

-1	(default) The change is enabled for programs using Version 7.2 or later semantics. In other words, the program has been compiled with Version 7.2 or later and the command line does not contain a compiler option for pre-7.2 semantics.
0	The change is enabled.
1	The change is disabled and the Version 7.1 and earlier font measuring code is used.

Please note the following issues regarding the use of this variable:

- The runtime's standard fonts are not affected by this configuration variable setting.

- Entry fields defined by physical units (CELLS or PIXELS) and all screens created using the AcuBench Screen Designer will not change.
- Entry fields will not grow larger due to this configuration variable setting. The majority will stay the same size, and a few might get smaller.
- Fixed width fonts are not affected by this configuration variable setting.

WAIT_FOR_ALL_PIPES

This configuration variable determines if the runtime calls the wait system call each time a "-P" file is closed. When WAIT_FOR_ALL_PIPES is set to "0" (off, false, no), the runtime does not make this call until it is ready to close the last pipe it knows about. Setting this configuration variable to the default "1" (on, true, yes) means that the runtime calls the wait system call when a "-P" file is closed.

WAIT_FOR_FILE_ACCESS

This configuration variable is designed for Windows 98 systems. It gives you some control over situations where a user must wait for access to a shared file. The runtime will try repeatedly to acquire the file lock, up to 400 times. If it has been unable to obtain a file lock after 400 tries, it will (by default) display a message box, asking if the user would like to continue waiting. If the user clicks the "Yes" button, then the runtime will try again another 400 times (or the value of LOCKING_RETRIES). If the user clicks the "No" button, then the runtime will return an error to the COBOL program (such as file error 30,33 (system error) or file error 99 (record locked).

The WAIT_FOR_FILE_ACCESS variable lets you choose one of three behaviors: either the user will always see the message box and make a choice, or the program will always return an error code if it cannot acquire the lock, or the runtime will always behave as if the user answered "yes" to the message box.

You can modify the text shown to the user in the message box via the TEXT configuration variable. The message is number 28. To include the filename in your message, insert “%s” at the place where you want the name of the file to appear. You can introduce line breaks by including “\n” in the message.

Possible values for the WAIT_FOR_FILE_ACCESS variable are:

0	“No”	Do not display message box if lock is not acquired. Send error to COBOL program.
1	“Ask”	Show the message box and ask the user. (Default)
2	“Yes”	Do not show the message box. Assume that the user wants to wait for the file. This ensures that the user eventually can access the file, but introduces a small risk of an infinite loop if the system’s lock table becomes corrupt.

For programs running in background (“-b” runtime option), or programs with redirected input or output, the “Ask” option is treated the same as the “Yes” option.

WAIT_FOR_LOCKS

This determines how the runtime handles file status error 99 conditions on record reads. This variable is not checked on record write operations. It can have one of the following values:

0	Do not wait for locked records, return error 99.
1	Wait for the locked record if no Declarative is available for the file, otherwise return error 99.
2	Always wait for the locked record, never return error 99.

Any other value (including the default value of “-1”) causes the runtime to wait for locked records only if you have compiled for RM/COBOL compatibility and the file does not have a Declarative.

WARNINGS

This configuration variable controls whether a warning message is printed and an error raised for the following conditions:

1. when non-numeric data is used in a context where numeric data is required
2. when there is a reference modification range error

By default, the runtime silently corrects reference modification range errors as follows:

- A start reference less than 1 is treated as 1. For example, var(0:3) is treated as var(1:3).
- A length reference less than 0 is treated as 0. Moving a zero-byte item is equivalent to moving spaces to the destination item. A zero-byte destination is not affected by the move. In a STRING statement, a length of zero for a string source is treated as 1, not 0.
- A start plus length reference that is past the end of the item is treated as meaning to the end of the item. For example, if the var is a PIC X(5) item, var(4:23) is treated as var(4:2).

WARNINGS can take the following values:

0	(off, false, no) No warning is printed.
1	(on, true, yes) A warning is printed. This is the default.
2	A warning is printed or sent to the error file. If you are in the debugger, an automatic breakpoint occurs.
3	For a non-numeric error, a warning is printed, an *intermediate* error is generated that calls the installed error procedures, if any, and the runtime is halted. For more information on error procedures, see CBL_ERROR_PROC in Appendix I of the *ACUCOBOL-GT Appendices Manual*.

Note: The setting you select for WARNINGS applies to reference modifier range errors when the start plus length reference is past the end of the item. Reference modifiers that are equal to or less than zero are always silently corrected as described above.

WARNING_ON_RECURSIVE_ACCEPTS

An event procedure may CALL another procedure which may contain ACCEPT statements, which, in turn, may contain embedded procedures. Although this is handled in the same fashion as nested PERFORMs and is perfectly legal, doing this poses the danger of going from one ACCEPT to another uncontrollably. When the limit of 10 nested accepts is reached, the program starts overwriting memory. It is possible to warn the user when the limit is reached by giving this configuration variable a zero (“0”) value. This gives users the opportunity to continue at their own risk. Giving WARNING_ON_RECURSIVE_ACCEPTS a non-zero value suppresses the warning.

To avoid overwriting memory, you may choose to re-code affected programs to terminate the ACCEPT and perform the CALL after you exit from the ACCEPT. You may also use CHAIN or CALL PROGRAM instead of the regular CALL, if applicable.

WHITE_FILL

This variable has meaning only on graphical systems such as Windows. Some graphical systems (such as Windows) use a “background brush” when they resize a window. By default, the background brush color for ACUCOBOL-GT is black (“0”, off, false, no). If you have arranged your default background to be white, you will see a black flash when you resize the window. This does not affect the final appearance of the window, but is briefly noticeable while the window is being redrawn.

Set WHITE_FILL to "1" (on, true, yes) to cause ACUCOBOL-GT's background brush to be set to white instead of black. Doing this will also cause the initial screen that ACUCOBOL-GT paints to be white instead of black.

Note: This variable *must* be set in the configuration file to be effective. Modifying this variable with the SET ENVIRONMENT verb has no effect.

WIN_ERROR_HANDLING

This variable has meaning only on graphical systems such as Windows. Use WIN_ERROR_HANDLING to control how hardware errors are handled.

When this variable is set to the default of "1" (on, true, yes), certain errors are handled directly by the host environment, and do not automatically return a file error code. For these errors, a dialog box is displayed that describes the error and offers "Cancel" and "Retry" buttons. The user may correct the error and press "Retry". If the user presses "Cancel", then your program receives the file error that it would have normally received.

If you set WIN_ERROR_HANDLING to "0" (off, false, no), then the dialog box is not shown, and your program receives the error directly.

WIN_F4_DROPS_COMBOBOX

This configuration variable applies only to programs running under Windows.

If WIN_F4_DROPS_COMBOBOX is set to its default value of "1" (on, true, yes), then combo boxes use the standard Windows handling for the <F4> key. Pressing <F4> while a combo box is active causes it to drop its drop-down list, and the COBOL program is not notified of an exception.

When this variable is set to "0" (off, false, no), pressing <F4> with a combo box active causes the COBOL program to get the exception, but the combo box does not drop its drop-down list.

It is not possible to get both behaviors at the same time.

WIN_SPOOLER_PORT

This variable allows you to divert printer output to a file or port through the Windows print spooler. Files created in this way are stored in binary encoding. You may set the Windows print spooler with "-P SPOOLER" or "-Q <*printername*>" with or without the DIRECT option. However, if you omit the DIRECT option, the resulting file will include all the embedded control codes formatting the print job for the original target printer.

By default, the value of WIN_SPOOLER_PORT is undefined. Set WIN_SPOOLER_PORT to a valid filename or port. This can be done in a configuration file, in the environment, or in the program. For example:

```
WIN_SPOOLER_PORT  c:\mydir\myprint.prn
```

or

```
SET ENVIRONMENT "WIN_SPOOLER_PORT" TO "c:\mydir\myprint.prn".
```

This will affect all print jobs performed in the current instance of the runtime. Any graphics operations performed in the COBOL application, such as WINPRINT-BITMAP or WINPRINT-GRAPH-DRAW, are preserved in the file, and will print. However, these options may result in a very large binary file.

The resulting file can be copied directly to any printer that is compatible with the original target printer. For example, the following command:

```
COPY /B c:\mydir\myprint.prn LPT1
```

will send the file to LPT1, while the "/B" option tells the COPY command that the file contains binary encoding.

WIN3_CLIP_CONTROLS

This option is specific to the Windows versions of ACUCOBOL-GT. It affects the way in which updates to a window interact with graphical controls in a window. Normally, Windows allows updates to a parent window to show through any controls in that window. The controls are then updated to create the proper final appearance. This is very fast, but it can cause controls to flash when the background is being updated. When this option is set to "1" (on, true, yes), the controls are clipped from the update region in the parent window before the parent is repainted. This causes the controls to remain relatively stable; however, screen repaints can be significantly slower, particularly when the runtime is creating and destroying controls. The default setting for this option is “0” (off, false, no). We recommend that you experiment with both settings to see which you prefer. Note that this option is examined when a floating window is created. Once a window is created, changes to this option have no effect on that window.

Note: When turned on, this option causes the Windows WS_CLIPCHILDREN style to be used whenever floating windows are created.

WIN3_EF_PADDED

This configuration variable has meaning only on Windows systems. Under Windows, unboxed entry fields include a small amount of extra space so the cursor can be seen when it is placed after the last character position. This space can be a problem if you want to convert a program and align screen items. When WIN3_EF_PADDED is set to “0” (off, false, no), this extra space does not appear in unboxed entry fields, and the entry field has only enough space for its character positions. When this variable is set to the default “1” (on, true, yes), the extra space appears in unboxed entry fields.

WIN3_GRID

This option is specific to Windows. When set to a non-zero value, it causes a fine grid to be drawn in each floating window. The grid outlines the character cells in the windows. This is intended as a debugging tool, to help you see how various controls line up against the window’s character cells. It can also help you adjust the layout of a screen.

The grid is drawn using the color number that WIN3_GRID is set to (see the COLOR phrase for the exact values). For example, setting WIN3_GRID to “4” will draw a cyan grid. The grid is drawn with dashed lines. Every fifth horizontal line and every tenth vertical line is drawn with a solid line.

WIN32_3D

This configuration variable causes the runtime to use the native 3-D features of Windows when drawing controls with the 3-D style. This has an effect only with the 32-bit Windows runtime. Turn this feature on by setting WIN32_3D to "1" (on, true, yes). When set to the default of "0" (off, false, no), the runtime supplies its own 3-D effects. The advantage of using the native Windows 3-D is that you get a slightly more modern appearance and a closer match to the appearance of other Windows programs. The disadvantages are:

1. Windows always draws the border using the colors selected in the system's control panel. As a result, the effect looks right only when placed on a window whose background is the USER-GRAY color. You can accomplish this easily by creating STANDARD windows that specify BACKGROUND-LOW.
2. The Windows 3-D effect is slightly larger than the runtime's 3-D effect. Windows draws a 1-pixel wide border around the control that is the same color as the USER-GRAY color. This border is essentially invisible against a window with the USER-GRAY background. However, this border can overwrite anything else that may be positioned there. The net effect is that you can't place controls as close together as you can with the runtime's 3-D.
3. This 3-D style can be used only with the 32-bit runtime.

The runtime adjusts for the physical differences between the two styles. Under either style, the position and usable size of the control's interior should be same.

Note: This configuration setting can effect the behavior of an application if it is using the latest Windows control styling, that is the **WIN32_NATIVECTLS** configuration variable set to 1, true, or on. If WIN32_3D has not been set by the user then the default value will be overridden and set to false (0 or off). If the user has set WIN32_3D then their settings will not be overridden and if it is set to true (1 or on) then 3D drawing will occur over the top.

WIN32_CTL_INPUT_STATUS

Setting this variable to the default of "1" (on, true, yes) causes ACCEPT…FROM INPUT STATUS to return a non-zero status if data is available in a control. If set to "0" (off, false, no), then the data in the control does not affect the status returned by ACCEPT…FROM INPUT STATUS.

This variable is only available in the Windows runtime and is not available to the thin client.

WIN32_NATIVECTLS

This variable enables your application to use the Windows control style that is in use on the workstation, (the workstation's theme is set to Windows XP or Vista). To enable these visual styles, your application must be running on an operating system that contains ComCtl32.dll version 6, which is included with Windows XP and Vista.

When set to "1" (on, true, yes), the application will display the current control styling available on that operating system, the XP look and feel on the Windows XP OS or the Vista look and feel on the Windows Vista OS.

Note: In addition to visual differences, some XP and Vista controls have different behaviors than their Windows classic counterpart (by Microsoft design). The behavior differences if any, that our internal testing has identified are documented in Book 2, Chapter 5 under the applicable control. Alternatively, you can find a consolidated list in the 8.1 ECN List (ECN 3734) located at the support section of the Micro Focus website.

The default setting is "0" (off, false, no) which prevents the runtime from using the Windows control styling, and forces the "gray chiseled" or classic Windows look.

Note: The configuration variable **WIN32_3D** can also change the look of controls in an ACUCOBOL-GT application. It is generally recommended that you leave WIN32_3D set to its default behavior of off or false when setting WIN32_NATIVECTLS to on or true.

Note: The Windows OS allows users to configure (accessibility options) whether or not keyboard shortcut names appear with underlines. For example "ctrl+c" vs. "<u>ctrl+c</u>". The WIN32_NATIVECTLS respects this setting and will display shortcut names accordingly.

WINDOW_INTENSITY

This configuration variable controls whether the color settings specified in the COLOR phrase of the DISPLAY WINDOW statement are used or ignored by the runtime. When the value of this variable is set to "0" (off, false, no), the COLOR intensity settings in all DISPLAY WINDOW statements are ignored. When the value of this variable is set to "1" (on, true, yes), which is the default, the runtime sets the windows intensity as specified.

WINDOW_TITLE

This variable has meaning only on graphical systems such as Windows. The ACUCOBOL-GT runtime system automatically sets the title of the application window to the base name of the initial object file. For example, if you run a program called "notepad.cbx", then the title on the main window will be set to "Notepad". The title is shown in lower-case except for the first letter, which is made upper case.

You may provide an alternate title by setting WINDOW_TITLE to the desired text. No translation of the text is done, so you should enter it using the desired case.

Note: Setting WINDOW_TITLE from within a program has no effect, because the WINDOW_TITLE setting determines only the window's initial title.

To change the title from within your program, use a DISPLAY statement. The syntax is:

```
DISPLAY text UPON GLOBAL WINDOW TITLE
```

where *text* is an alphanumeric literal or variable. Enter the title with the desired case. The title is always shown in the ANSI font, so if you are using a different font, your text will be translated to ANSI.

To ensure that the WINDOW_TITLE variable operates as expected, make sure that the first screen operation in your program is not DISPLAY WINDOW with a title (the DISPLAY WINDOW title is stored in the same place as the WINDOW_TITLE). Instead, do some other screen operation first, such as “DISPLAY WINDOW, ERASE”.

WINPRINT_NAMES_ONLY

This variable allows you to generate a list of the names of printers installed on a Windows PC. It does this by altering the behavior of some of the operations of the WIN$PRINTER library routine. When WINPRINT_NAMES_ONLY is set to a value of “1” (on, true, yes), the WIN$PRINTER operations that retrieve printer information return only the names of installed printers, rather than the real-time status of all available printer capabilities.

This variable can be set in the configuration file or directly in your program with the following code:

```
SET ENVIRONMENT "WINPRINT-NAMES-ONLY" TO "1".
```

Note on WIN$PRINTER library routine:

When this variable is turned on, the following operations of the WIN$PRINTER library routine are affected:

```
WINPRINT-GET-PRINTER-INFO
WINPRINT-GET-PRINTER-INFO-EX
WINPRINT-GET-CURRENT-INFO
WINPRINT-GET-CURRENT-INFO-EX
```

Instead of returning detailed information about the capabilities of each printer (duplex, copying, etc.), the routine returns only the name of the printer. This can provide a significant performance improvement, particularly with networked printers.

If you are using the default printer settings, set the WINPRINT_NAMES_ONLY variable to "1", generate a list of printer names using WINPRINTER-GET-PRINTER-INFO-EX (see the WIN$PRINTER documentation in the Appendices of the ACUCOBOL-GT manual set, or refer to the sample program "prndemox.cbl"), and select the desired printer.

If you want to modify the printer settings, such as the number of copies or the paper orientation, you should perform the steps described above, and then set WINPRINT_NAMES_ONLY back to the default of "0" (off, false, no). You may then use WINPRINT-GET-PRINTER-INFO-EX to obtain detailed information about the capabilities of the selected printer.

For more information about Windows printing, refer to WIN$PRINTER in Appendix I.

WRAP

The setting of this variable determines whether a DISPLAY statement will wrap around or be truncated when it extends past one line. When it is set to "0" (off, false, no), DISPLAY statements will be truncated. Also, any DISPLAY statement that references a column past the right edge of the current window will be ignored. An ACCEPT statement that references a column past the right edge will be placed in the home position of the window. The default value for this setting is "1" (on, true, yes).

XFD_DIRECTORY

This variable tells the runtime system the name of the directory that contains the data dictionaries built by the ACUCOBOL-GT compiler. The default value is the current directory.

For example, to tell the runtime that the dictionaries are stored in the directory "/usr/inventory/dictionaries" you would enter:

```
xfd_directory   /usr/inventory/dictionaries
```

See also the “-Fo” compile-time option, which tells the compiler where to put the dictionaries. Unless you have moved the dictionaries, you should use the same value for XFD_DIRECTORY that you used with the “-Fo” option.

If you have embedded an XFD file in an object library, the runtime will read that file instead of an XFD file that has the same name but is stored in the directory specified by XFD_DIRECTORY. The exception to this is when the XFD_DIRECTORY configuration variable uses remote name notation.

Remote name notation is allowed for the XFD_DIRECTORY variable if your runtime is client-enabled. See *ACUCOBOL-GT User’s Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

XFD_PREFIX

This variable defines a series of directories to search for XFD files, rather than indicating only one (as in XFD_DIRECTORY). Each directory is searched in order until an XFD matching the name of the file is found. Once a file with the same name is found, the runtime stops searching, even if other files of the same name are located in a subsequent directory in the search parameter. Only named directories are searched, not subdirectories.

Note: If the XFD you are searching for does not match the file specifications (max-keys, max-rec-size, min-rec-size, and key parameters, for example) of the file you are trying to open, the runtime will not continue searching the directories listed in XFD_PREFIX until a correct XFD file is found.

The default for XFD_PREFIX is empty. If this variable is set to any other value, the configuration variable XFD_DIRECTORY (in which you specify only one directory) is ignored. You can specify a directory path that contains embedded spaces if you surround the path with quotation marks. Separate entries using a semi-colon (;). For example:

```
XFD_PREFIX C:\ “Sales Data”;C:\Customers
```

You may specify up to 4096 characters for this variable. Remote name notation is allowed for the XFD_PREFIX variable if your runtime is client-enabled. See *ACUCOBOL-GT User's Guide* sections 5.2.1 and 5.2.2 for more information about client-enabled runtimes and remote name notation.

XTERM_PROGRAM

Some users may want to debug with an xterm, but don't actually want to debug with the xterm executable because it doesn't have some of the abilities they need (such as displaying non-ASCII characters). You can specify the executable used to show the debugger on UNIX by setting the XTERM_PROGRAM runtime configuration variable.

Its default value is "xterm", but it can be set to any compatible program such as dtterm or kterm. The runtime executes this program when it tries to create the program for background debugging. Note that the runtime passes some arguments to this program, so this program must be able to execute with those arguments. These arguments are:

-title "title of the window"

-Sccn

-display Xserver-name

The "-Sccn" option allows the program to be used as the input and output channel for the runtime, and is absolutely required. Without this option, the program won't know to display data from the runtime.

4 Runtime Options

Key Topics

4.1 Using the Runtime

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 for 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 the *ACUCOBOL-GT User's Guide* 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.

4.2 List of 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”.

-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. Remote name notation is allowed for this option if your runtime is client-enabled.
--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 **Chapter 5, "Chapter 5: 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.

--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$KEYBUF using opcode "9" and this file name prior to executing the first COBOL program. The effect is that the keystrokes recorded in the file are treated as the runtime's first user input. For more information on W$KEYBUF, see 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. 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
--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 4.4.3 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". For more information, see the *ACUCOBOL-GT User's Guid*e section 3.7, "The Profiler."
-p0	Tells the profiler to include zero execution count paragraphs in the "acumon.xml" file.

-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 of the *ACUCOBOL-GT User's Guide*. Windows DLLs and UNIX shared libraries are described in Chapters 3 and 4 of *A Guide to Interoperating with ACUCOBOL-GT.* Remote name notation is allowed. See section 5.2.2 of the *ACUCOBOL-GT User's Guide* **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.

5 Runtime Debugger

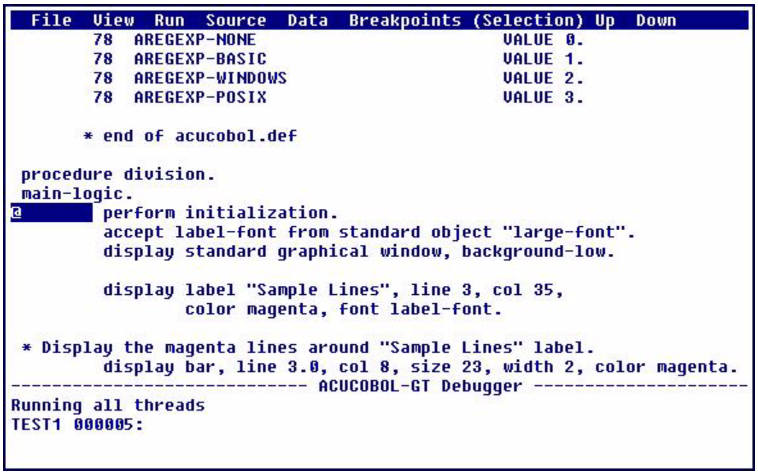
Key Topics

5.1 About the 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.

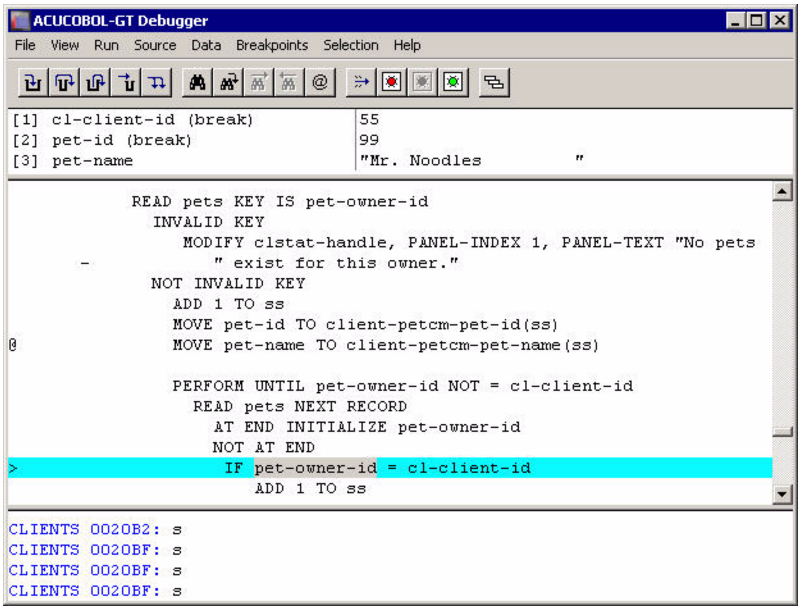
In all environments, the runtime debugger interface contains a menu bar and command window. To navigate through source code in character environments, use the “Up” and “Down” menu items. You can also use the arrow keys and Page Up and Page Down keys to move through the code.



```
        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.



```
[1] cl-client-id (break)          55
[2] pet-id (break)                99
[3] pet-name                      "Mr. Noodles          "
```

```
           READ pets KEY IS pet-owner-id
             INVALID KEY
                MODIFY clstat-handle, PANEL-INDEX 1, PANEL-TEXT "No pets
      -            " exist for this owner."
             NOT INVALID KEY
               ADD 1 TO ss
               MOVE pet-id TO client-petcm-pet-id(ss)
@              MOVE pet-name TO client-petcm-pet-name(ss)

               PERFORM UNTIL pet-owner-id NOT = cl-client-id
                 READ pets NEXT RECORD
                   AT END INITIALIZE pet-owner-id
                   NOT AT END
>                   IF pet-owner-id = cl-client-id
                        ADD 1 TO ss
```

```
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 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 7 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.

5.2 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.

5.3 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 -- 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.

Using variables, you may specify data names that require arguments, such as tables that require indexes. You cannot specify literals.

You can use the mouse, the 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.

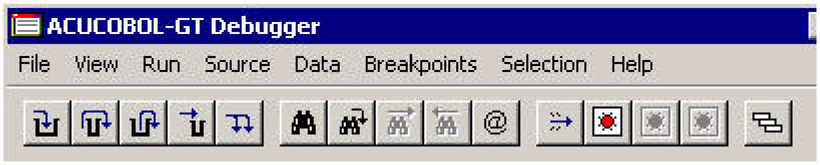
Display a variable -- to view the value of a variable, double-click on that variable.

View procedure -- to scroll quickly to a paragraph or section, double-click on its name.

Run to desired line -- 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.

5.4 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.



The Debugger Menu Bar and Toolbar (Windows).

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.

5.4.1 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/ Toggle at Cursor Line	Sets or removes a breakpoint at the source line containing the cursor.

Command	**Menu Option**	**Description**
<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 *text*	Source/Find Backwards	Locates *text* in the program's source code. The debugger searches backwards from the current cursor line.
FF *text*	Source/Find Forward	Locates *text* in the program's source code. The debugger searches forward from the current cursor line.
FT *text*	Source/Find from Top	Locates *text* in the program's source code. The debugger starts at the top of the current program source.
VP	View/Perform Stack	Lists all of the nested paragraphs leading up to the current statement, starting from the beginning of the program.
W *procedure*	Source/Paragraph	Positions the cursor at the procedure you name. The procedure must be located in the current program.
W@	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.

Command	**Menu Option**	**Description**
WT	Source/Line 1	Positions the cursor at the first line (top) of your program.

5.4.2 Other Commands

Command	**Menu Option**	**Description**
!	File/Shell	Invokes the operating system's command processor, allowing you to enter commands.
<*script-file*	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.
>*script-file*	File/Record Script	Turns on a recorder that saves all of your keyboard input and menu selections to a file of your choice.
A *variable*	Data/Accept	Lets you modify the contents of a variable.
B	View/Breakpoints Breakpoints/View	Displays a dialog box with all existing breakpoints. You can add/modify breakpoints from this dialog box.
B *address, [skip #]*	Breakpoints/Set	Sets a breakpoint with a skip count. The breakpoint will not be activated until it has been hit *skip#* times.
B *address, [skip #], [WHEN cond]*	Breakpoints/Set	Sets a breakpoint with a skip count and/or condition. The breakpoint will not be activated unless *cond* is true # times.
C *address*	Breakpoints/Clear	Removes a breakpoint. You can enter either the breakpoint's paragraph name or hexadecimal address.

Command	**Menu Option**	**Description**
CA	Breakpoints/Clear All	Removes all breakpoints.
CM *number*	Data/Monitor/ Clear	Clears variable monitor *number.*
CMA	Data/Monitor/ Clear All	Clears all variable monitors.
CWA		Clears all variable watches.
D *variable* *[, X]*	Data/Display	Shows the contents of a variable. The value is shown in the debugger command window. If *X* is appended to the display command, the variable is displayed in hexadecimal. If the variable is specified by its *absolute address* from a program listing, it must be preceded by "." (a period)
D *variable*(*x*:*y*)		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.
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.
G *address*		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.

Command	**Menu Option**	**Description**
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.
M *variable*	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.
R *script*		Run a debugging script. The debugger reads commands from a script (but user-input is gathered normally).
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.

Command	**Menu Option**	**Description**
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.
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 *not* cause program execution to halt when they change.
WS *number*	Source/Window Size	Specifies the number of lines to show in the command window.

Command	**Menu Option**	**Description**
WW *number*	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.

5.4.3 Multi-threading 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.

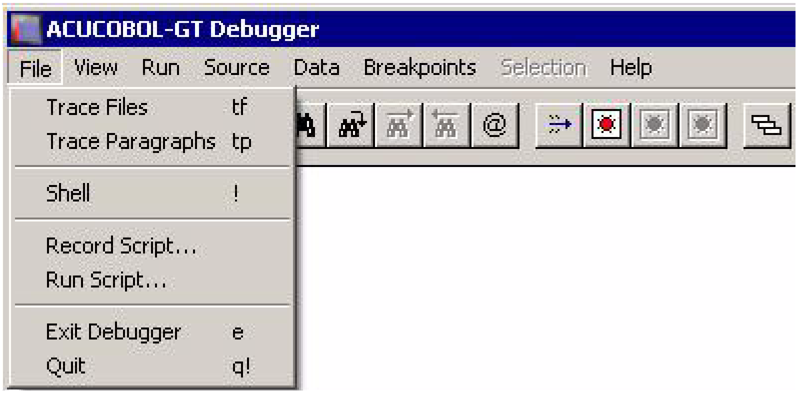
5.4.4 Getting Help

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.

5.4.5 File Menu

The *File* menu contains commands relating to the overall operation of the debugger.



The File Menu (Windows).

Trace Files toggles file tracing on or off.

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.

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.

Some file systems can print extra information if a higher level of tracing is enabled. This extra information is mostly useful to our Technical Support department, and they may ask you to execute a “tf n” for some integer “n”.

File trace example:

```
runcbl  -de  trace.fil  program1
```

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.

The keyboard form of this command is “TF [#]”.

****Trace Paragraphs** toggles paragraph tracing on or off.

Paragraph tracing is a listing of all paragraphs and sections entered at runtime.

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.

The keyboard form of this command is “TP”.

Shell 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”.

Note: Under the default Windows setup, the command processor will run as a full screen application.

The keyboard form of this command is “!”.

Record Script 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.

Play back the recording with the Run Script command.

See also the description of the W$KEYBUF routine in Appendix I.

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.

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.

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.

The keyboard form of this command is "> *script-file*". The runtime does not process the filename. To turn off the recorder, use ">" by itself.

Run Script runs a debugger script file. Control returns to the keyboard when the script is finished.

The keyboard form of this command is "< *script-file*".

Exit Debugger turns off the debugger but continues execution of your program.

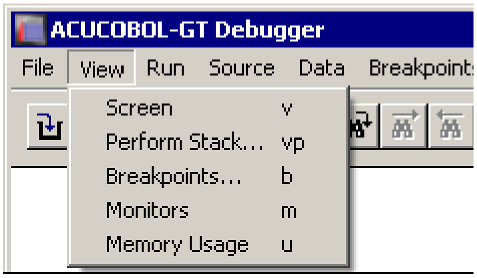
The keyboard form of this command is "E".

Quit halts your application and exits the debugger.

The keyboard form of this command is "Q!".

5.4.6 View Menu

The *View* menu contains commands related to viewing and monitoring your program.



The View Menu (Windows).

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".

View Breakpoints 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.

The keyboard form of this command is “B”.

View Monitors 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*, below.

The keyboard form of this command is “M”.

Memory Usage displays the amount of dynamically allocated memory currently used by the runtime system. There are five types:

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.

File memory is memory used by your open files, including the indexed file cache.

Window memory is memory used by your pop-up windows. This includes the debugger’s own pop-up window.

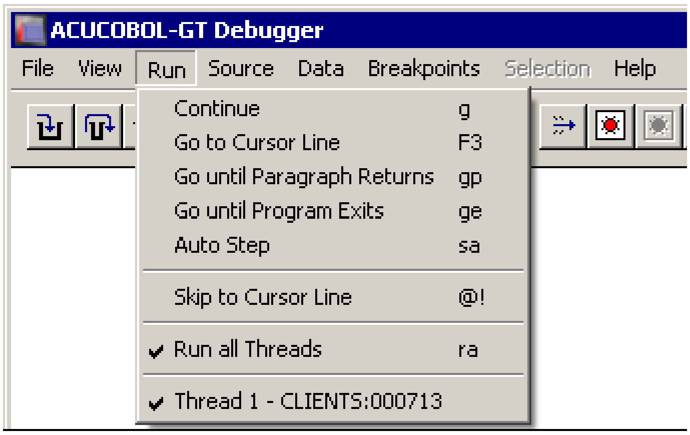
Overhead memory is memory used directly by the runtime system that is not controlled by your program.

Dynamic memory is memory allocated by the program via the M$ALLOC library routine.

The keyboard form of this command is “U”.

5.4.7 Run Menu

The *Run* menu contains commands related to executing your program.


ACUCOBOL-GT Debugger
File View Run Source Data Breakpoints Selection Help
Continue g
Go to Cursor Line F3
Go until Paragraph Returns gp
Go until Program Exits ge
Auto Step sa
Skip to Cursor Line @!
✓ Run all Threads ra
✓ Thread 1 - CLIENTS:000713

The Run Menu (Windows).

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”.

Auto Step 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.

The keyboard form of this command is “SA”.

Step and P-Step 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.

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)

The keyboard command is “S”. You may follow the keyboard command with a number of steps to take.

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.

The keyboard command is “P.” You may follow the keyboard command with the number of “perform steps” to take.

***Skip to Cursor Line** 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.

Use this command with care, because the skipped lines are not executed. You may skip important sections of code and experience unexpected results.

The keyboard form of this command is “@!”.

Note: The “@!” command is not available for debugging a native-code module.

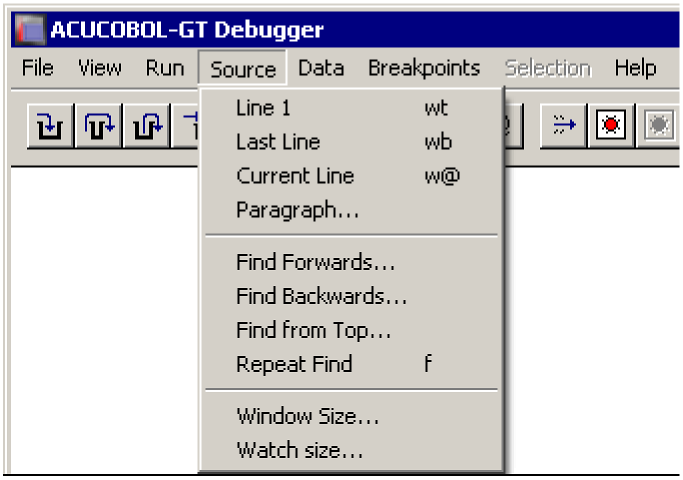
Run all Threads toggles or sets the “Run All Threads” setting. Once it is set, all threads run simultaneously under the debugger. Though all threads run simultaneously, only the debugger’s current thread is traced when you are stepping through a program. However, breakpoints in other threads are active 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.

The keyboard form of this command is “RA [#]”.

Thread shows the threads contained in the program, and places a check mark next to the current thread.

5.4.8 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).

***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 *procedure*”.

Find Forwards 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.

The default text for the search is shown in a dialog box. This is the current *selection* (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.

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.

The keyboard form of this command is “FF *text*”.

***Find Backwards** 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.

The default text for the search is shown in a dialog box. This is the current *selection* (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.

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.

The keyboard form of this command is “FB *text*”.

***Find from Top** 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.

The keyboard form of this command is “FT *text*”.

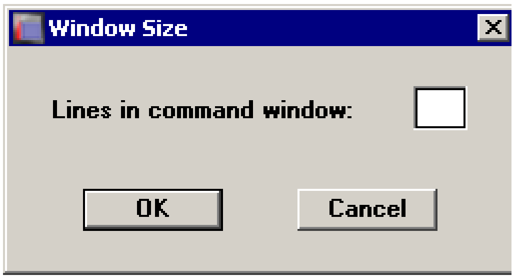
***Repeat Find** repeats the last Find command, starting at the cursor line.

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 *number*”.

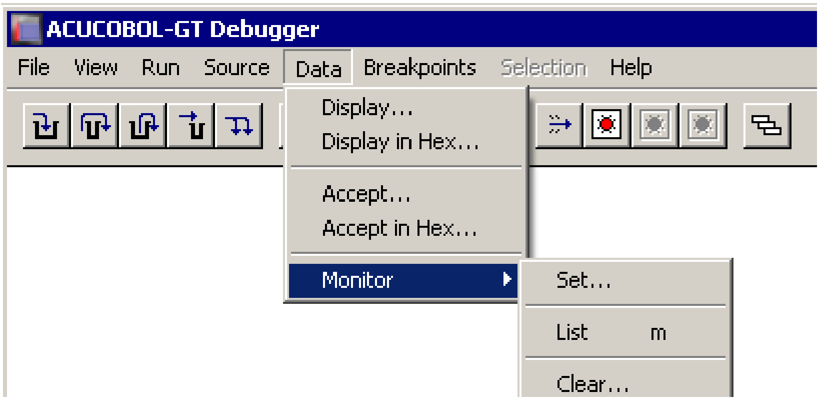
***Watch Size** displays a dialog box that allows you to specify the number of lines to display in the Watch Window. The number cannot exceed the total number of items being monitored and watched. Specifying a larger number results in no change. The Watch Window size dialog box (actually titled “Window Size”) looks like this:



The keyboard form of this command is “WW *number*”.

5.4.9 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.

Display 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 *variable*".

Table elements cannot be highlighted with a mouse click. Instead, use this keyboard command:

```
d  variable (index)
```

The variable's name is followed by the desired index in parentheses. The index must be a numeric literal.

To display a reference modified variable (that is, to view some portion or substring of the data item), use the syntax:

```
d  variable(x:y)
```

This shows *y* characters of *variable*, starting from character *x*.

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:

```
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).
```

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.

Reference modification and indexing are valid with duplicate names. For example, all of the following are valid:

```
d field-1(1:1)
d field-1(1)
d field-1(1)(1:1)
```

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 command "d ws-day(1:ws-month)" would fail, because there is more than one "ws-month" defined.

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, preceded by "." (a period). For example:

```
d  .213.5
```

Display in Hex 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.)

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.

The keyboard form of this command is "D *variable*, X".

Accept 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.

The keyboard form of this command is "A *variable*".

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.

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:

```
a  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.

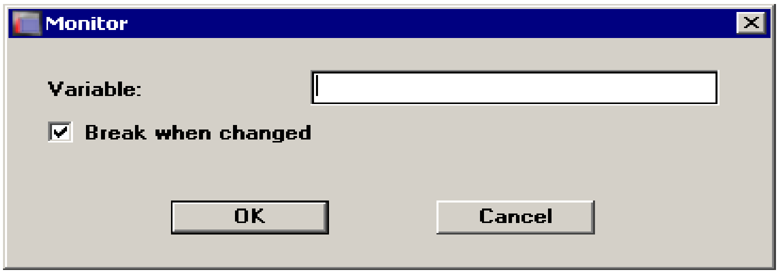
Accept in Hex allows you to modify the contents of a variable in hexadecimal format. You can enter or display up to 2048 hex characters (1024 bytes of data).

To accept a variable in hexadecimal format from the command line, use the command:

```
a  variable  x
```

The **Monitor** submenu contains commands that relate to monitored variables.

Set displays a dialog box which 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.) The Monitor dialog box looks like this:



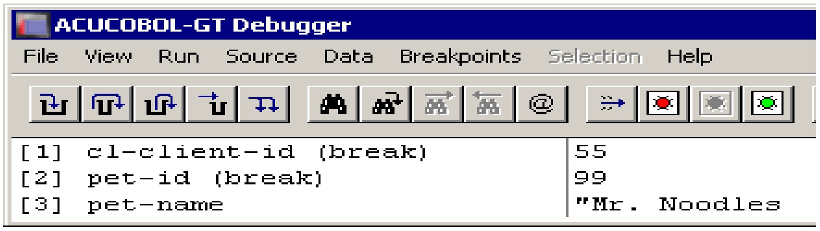
The Monitor dialog box includes a check box labeled "Break when changed". When this box is checked, the selected variable becomes monitored, and if it is unchecked, the variable is only *watched*. The default value of this check box is On (checked).

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:



```
[1] cl-client-id (break)     55
[2] pet-id (break)           99
[3] pet-name                 "Mr. Noodles
```

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.

The keyboard form of this command is "M *variable*".

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 *Clear*, 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 *List* option to display all monitors and their numbers.

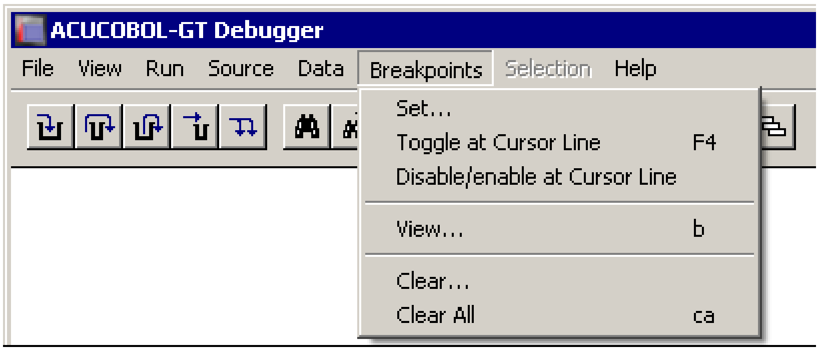
The keyboard form of this command is "CM *number*".

Clear All clears all monitors.

The keyboard form of this command is “CMA”.

5.4.10 Breakpoints Menu

The Breakpoints menu contains commands for managing a program’s breakpoints.



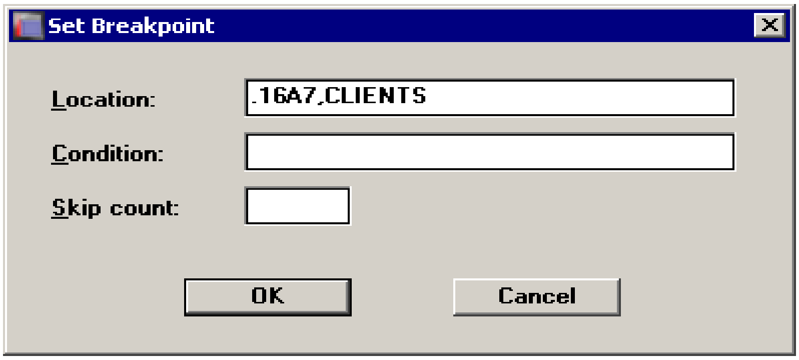
The Breakpoints Menu.

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.

Set allows you to set a breakpoint at a paragraph. Selecting Set displays the Set Breakpoint dialog box.



Location: .16A7,CLIENTS

Condition:

Skip count:

Set Breakpoint Dialog Box.

The Set Breakpoint dialog box prompts you for a breakpoint *Location*, *Condition*, and *Skip count*.

The ***Location*** 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. 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.

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 *Set* or *Toggle at Cursor* to set the desired breakpoint.

If you have compiled for full source debugging, you can use an alternate notation to set a breakpoint within a called program. Instead of specifying an exact address, provide the full path and file name for the ACUCOBOL-GT source file, followed by a colon (":") and the line number.

The syntax is:

```
    /fullpath/filename.cbl:line
```

For example, in the command window, you could type:

```
   b /usr/source/invoice.cbl:559
```

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.

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

```
   b address, counter
```

This command can also be set from the command line with:

```
   b address [,program] [,SKIP count] [,WHEN condition]
```

A second command that is also supported but does not allow conditions to be set is:

```
   b address [,program] [,count]
```

***Toggle at Cursor Line** sets or removes a breakpoint at the source line containing the cursor.

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.

***Disable/enable at Cursor Line** 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.

View 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.

Clear 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. Exact addresses are given in the View command described above.

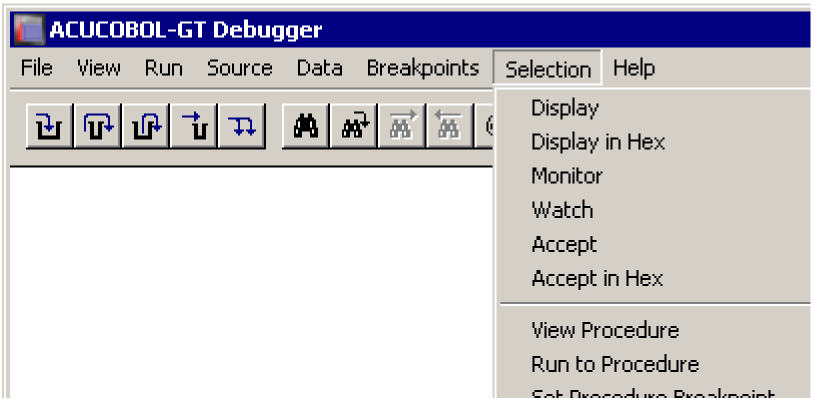
To use this command from the keyboard, type “C address”.

Clear All removes all breakpoints.

To use this command from the keyboard, type “CA”.

5.4.11 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.

***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.

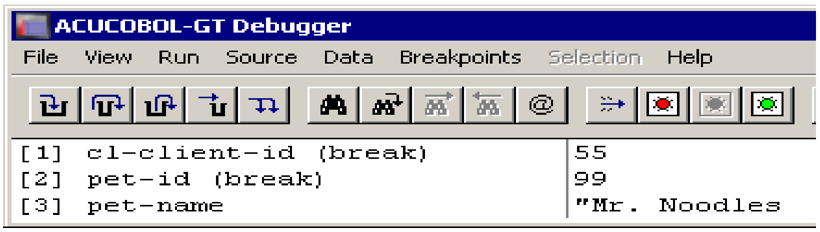
You can also perform this by double-clicking the left mouse button on the desired variable.

***Display in Hex** shows the contents of the selected variable in hexadecimal notation. Allows you to view the internal storage of every byte in the variable.

***Monitor** 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.

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.

This is what a Watch Window looks like:



```
[1] cl-client-id (break)      55
[2] pet-id (break)            99
[3] pet-name                  "Mr. Noodles
```

***Watch** sets a watch on the selected variable. Changes to a watched variable do *not* 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:

****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.

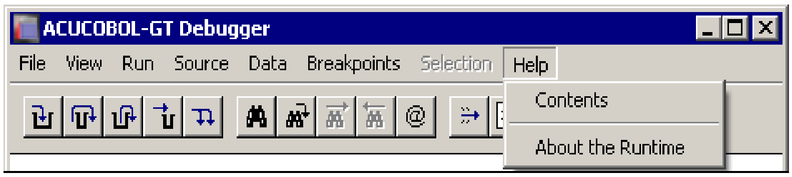
F7 (display variable on current line) and the Tab key (highlight variable on current line), as well as the mouse, also pay attention to qualified and indexed data items in the source. As long as a variable and all of its qualifiers and indexes are on one line, the entire expression is evaluated by these 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.

5.4.12 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).

Contents shows you the Debugger Help table of contents.

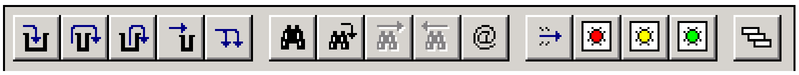
Search allows you to search for specific words, as you would in a book index.

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.

5.4.13 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 dim (unavailable).



The Debugger Toolbar (Windows).

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 dialog box 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.

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.

5.5 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 our 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.

5.6 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 our 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.

5.7 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.

5.8 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.

5.8.1 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.

5.8.2 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

5.9 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.

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.

5.10 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.

```
*** 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 ***
```

5.10.1 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 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 (“-Gl”) 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
```

5.10.2 ADR Restrictions

1. Tables are only expanded up to 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 (“-Gl”) in order to show line numbers, and symbol information (“-Gs”) to see data items in detail.

6 File Status Codes

Key Topics

6.1 Standards for File Status Codes

ACUCOBOL-GT conforms to five different standards regarding the values of file status codes. These codes are those used by RM/COBOL-85 (ANSI 85), RM/COBOL version 2 (ANSI 74), Data General ICOBOL, VAX COBOL, and IBM DOS/VS COBOL. By default, ACUCOBOL-GT uses the RM/COBOL-85 set. You can change the current set by changing the configuration variable **FILE_STATUS_CODES** (see also the *User's Guide*, section 2.7.3, "File Status Codes") .

The table in the next section describes the various file status codes returned by each condition. Some of the status values in the table have a second two-character code listed. This code distinguishes between different causes for the same FILE STATUS code. You can obtain this second code value by calling the ACUCOBOL-GT library routine C$RERR, which is described in Appendix I. Where a second code is not listed, its value is "00".

For file systems that support READ PREVIOUS, wherever READ NEXT is mentioned, you may assume that READ PREVIOUS is also implied. An *end of file* for READ NEXT is analogous to a *beginning of file* for READ PREVIOUS.

6.2 Table of Codes

Regardless of which set of status codes is being used:

- Any code that starts with a "0" is considered successful.
- Any code that starts with a "1" is considered to be an "at end" condition.
- Any code that starts with a "2" is considered to be an "invalid key" condition.

Refer to the following table for a description of each status code:

85	**74**	**Vax**	**DG**	**IBM**	**Condition**
00	00	00	00	00	Operation successful.
02	02	00	00	00	The current key of reference in the record just read is duplicated in the next record. (read next)
02	02	02	00	00	The operation added a duplicate key to the file where duplicates were allowed. (write, rewrite)
05	00	05	00	10	Optional file missing. If the open mode is I-O or EXTEND, then the file has been created. This is also returned by DELETE FILE if the file is not found. (open, delete file)
07	00	07	00	00	A CLOSE UNIT/REEL statement was executed for a file on a non-reel medium. The operation was successful.
0M	0M	0M	0M	00	The operation was successful, but some optional feature was not used. For example, if you opened a file that specified an alternate collating sequence, but the host file system did not support that feature, then the open would succeed, but it would return this status.
10	10	10	10	10	End of file. (read next)
14	00	14	00	00	A sequential READ statement was attempted for a relative file, and the number of digits in the relative record number is larger than the size of the relative key data item. (read next)
21	21	21	21	21	Primary key was written out of sequence, or the primary key on a rewrite does not match the last record read. This error occurs only for an indexed file open with the sequential access mode. (write, rewrite)
22	22	22	22	22	Duplicate key found but not allowed. (write, rewrite)
23	23	23	23	23	Record not found.
24	24	24	24	24	Disk full for relative or indexed file. (write)

85	**74**	**Vax**	**DG**	**IBM**	**Condition**
24, 01	00	24, 01	00	24	A sequential WRITE statement was executed for a relative file, and the number of digits in the relative record number was larger than the size of the relative key data item. (write)
30, xx	30, xx	30, xx	30, xx	30	Permanent error. This is any error not otherwise described. The secondary code value is set to the host system's status value that caused the error. See your operating system user manual for an explanation, and C$RERR in Appendix I.
34	34	34	34	34	Disk full for sequential file or sort file. (write, sort)
35	94, 20	35	91	93	File not found. (open, sort)
37, 01	95, 01	37, 01	91, 01	93	The file being opened is not on a mass-storage device which is required for the file type or the requested open mode. (open)
37, 02	95, 02	37, 02	91, 02	93, 02	Attempt to open a sequential file with fixed-length records as a Windows spool file.
37, 07	90, 07	39, 07	91, 07	93	User does not have appropriate access permissions to the file. (open)
37, 08	95, 08	37, 08	91, 08	93	Attempt to open a print file for INPUT. (open)
37, 09	95, 09	37, 09	91, 09	93	Attempt to open a sequential file for I/O and that file has automatic trailing space removal specified. (open)
37, 99	95, 99	37, 99	91, 99	93, 99	A Windows or Windows NT runtime that is not network-enabled tried to access a file on a remote machine.
38	93, 03	38	92	93	File previously closed with LOCK by this run unit. (open)

85	**74**	**Vax**	**DG**	**IBM**	**Condition**
39, xx	94, xx	39, xx	9A, xx	95	Existing file conflicts with the COBOL description of the file. (open) The secondary error code may have any of these values: **01** - mismatch found but exact cause unknown (this status is returned by the host file system) **02** - mismatch found in file's maximum record size **03** - mismatch found in file's minimum record size **04** - mismatch found in the number of keys in the file **05** - mismatch found in primary key description **06** - mismatch found in first alternate key description **07** - mismatch found in second alternate key description The list continues in this manner for each alternate key.
41	92	41	91	93	File is already open. (open)
42	91	42	92	92	File not open. (close)
42	91	94	91	92	File not open. (unlock)
43	90, 02	43	92	23	No current record defined for a sequential access mode file. (rewrite, delete)
44	97	44	92	21	Record size changed. The record being rewritten is a different size from the one existing in the file, and the file's organization does not allow this. (rewrite) This status code can also occur if the record is too large or too small according to the RECORD CONTAINS clause for the file. (write, rewrite)
46	96	46	92	21	No current record. This usually occurs when the previous operation on the file was a START that failed, leaving the record pointer undefined. (read next)

85	**74**	**Vax**	**DG**	**IBM**	**Condition**
47, 01	90, 01	47, 01	92, 01	13	File not open for input or I-O. (read, start)
47, 02	91, 02	47, 02	92, 02	13	File not open. (read, start)
48, 01	90, 01	48, 01	92, 01	13	A file that is defined to be access mode sequential is open for I-O, or the file is open for INPUT only. (write)
48, 02	91, 02	48, 02	92, 02	13	File not open. (write)
49, 01	90, 01	49, 01	92, 01	13	File not open for I-O. (rewrite, delete)
49, 02	91, 02	49, 02	92, 02	13	File not open. (rewrite, delete)
93	93	91	94	93	File locked by another user. (open)
94, 10	94, 10	97	97, 10	93	Too many files open by the current process. (open)
94, 62	94, 62	39, 62	92, 62	93	One of the LINAGE values for this file is illegal or out of range. (open, write)
94, 63	94, 62	39, 62	92, 62	93	Key not specified (specifying a table whose size is zero) in a SORT or MERGE statement
98, xx	98, xx	30, xx	9B, xx	93	Indexed file corrupt. An internal error has been detected in the indexed file. The secondary status code contains the internal error number. The file should be reconstructed with the appropriate utility.
99	99	92	94	23	Record locked by another user.
9A	9A	9A	9A	23	Inadequate memory for operation. This most commonly occurs for the SORT verb, which requires at least 64K bytes of free space. (any)
9B	9B	9B	9B	23	The requested operation is not supported by the host operating system. For example, a deferred file system initialization failed, or a READ PREVIOUS verb was executed and the host file system does not have the ability to process files in reverse order. (any) If you are using AcuXML, this error results when the program tries to open a file EXTEND or I-O. With AcuXML, programs are able to open files INPUT or OUTPUT only.

85	**74**	**Vax**	**DG**	**IBM**	**Condition**
9C	9C	9C	9C	23	There are no entries left in one of the lock tables. The secondary error code indicates which table is full: **01** - operating system lock table **02** - internal global lock table (see the **MAX_LOCKS** configuration variable) **03** - internal per-file lock table (see the **LOCKS_PER_FILE** configuration variable)
9D, xx	9D, xx	9D, xx	9D, xx	92	This indicates an internal error defined by the host file system. The "xx" is the host system's error value. This is similar to error "30", except that "xx" is specific to the host file system instead of the host operating system. For example: **02** - In Acu4GL or AcuXML, 9D,02 indicates that an XFD file is corrupt. This could be the result of a parsing error. **03** - In Acu4GL or AcuXML, 9D,03 indicates that an XFD file is missing. This could be the result of a parsing error. **05** - In AcuXML, 9D,05 indicates that there was an XFD parsing error, so AcuXML was unable to read a record. Refer to the specific product documentation for more details on the host file system's error codes.
9E, xx	9E, xx	9E, xx	9E, xx	92	This indicates an error occurred in the transaction system. The exact nature of the error is shown by the contents of TRANSACTION-STATUS. For more information, see **section 6.5, "Transaction Error Codes."**
9Z	9Z	9Z	9Z	92	This indicates that you are executing the program with a runtime that has a restriction on the number of records it can process. You have exceeded the record limit.

6.3 Input/Output Error Codes for Error 23s

These codes are specific to RM/COBOL START statement when the WHILE KEY LIKE phrase is specifed.

23	An attempt was made to randomly access a record that does not exist in the file, or a START or random READ statement was attempted on an optional input file that is not present. For a relative file, this means the relative key data item contains a value that is less than one, refers to a deleted record, or is greater than the highest relative record number existing in the file. For an indexed file, this means the specified value of the record or alternate record key does not refer to a record existing in the file.
23,01	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern class character range cannot include a multi-character escape. This value corresponds to compiler message 682, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,02	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern class character range cannot be a hyphen '-' except at the beginning or end of a positive character group. This value corresponds to compiler message 683, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,03	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern class character range cannot be an opening bracket '['. This value corresponds to compiler message 684, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,04	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern class character range cannot specify a decreasing range. This value corresponds to compiler message 685, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.

23,05	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern character class subtraction cannot be followed by an additional class specification. This value corresponds to compiler message 686, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,06	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern escape sequence (initiated by '\') is not valid. This value corresponds to compiler message 687, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,07	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value that requires more memory than is available for pattern compilation. This value corresponds to compiler message 688, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,08	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern quantifier opened with an opening brace '{' is missing the closing brace '}'. This value corresponds to compiler message 689, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,09	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern character class expression is missing the closing bracket ']'. This value corresponds to compiler message 690, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,10	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern parenthesized subexpression is missing the closing parenthesis ')'. This value corresponds to compiler message 691, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.

23,11	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern category escape '\p{' or '\P{' is missing the closing brace '}'. This value corresponds to compiler message 692, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,12	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern category escape '\p{' or '\P{' is missing the opening brace '{'. This value corresponds to compiler message 693, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,13	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern category escape '\p{' or '\P{' contains an unknown category specification. This value corresponds to compiler message 694, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,14	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern quantifier maximum count is less than the minimum count. This value corresponds to compiler message 695, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,15	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern quantifier maximum count is missing; at least one decimal digit was expected. This value corresponds to compiler message 696, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,16	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern quantifier maximum count is too large (> 65535). This value corresponds to compiler message 697, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.

23,17	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern quantifier minimum count is missing; at least one decimal digit was expected. This value corresponds to compiler message 698, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,18	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: A pattern quantifier minimum count is too large (> 65535). This value corresponds to compiler message 699, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,19	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected closing brace '}'. This value corresponds to compiler message 700, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,20	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected closing bracket ']'. This value corresponds to compiler message 701, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,21	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected closing parenthesis ')'. This value corresponds to compiler message 702, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,22	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected quantifier '*' that is not preceded by a valid atom. This value corresponds to compiler message 703, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.

23,23	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected quantifier '+' that is not preceded by a valid atom. This value corresponds to compiler message 704, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,24	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected quantifier '?' that is not preceded by a valid atom. This value corresponds to compiler message 705, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,25	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern contains an unexpected quantifier '{' that is not preceded by a valid atom. This value corresponds to compiler message 706, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,26	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern data item that has a value with a syntax error: The pattern is too large or complex to compile. This value corresponds to compiler message 707, which is described in Appendix B: Compiler Messages of the RM/COBOL Language Reference Manual.
23,27	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a pattern pointer data item that does not point to a valid compiled pattern.
23,28	An attempt was made to execute a START statement with a WHILE phrase and the LIKE condition specifies a compiled pattern that contains an unrecognized pattern matching instruction code; the compiled pattern is not valid . This can mean that the compiled pattern data item was not properly initialized or was corrupted by an unintended store into the pattern data item after initialization.

6.4 Vision Secondary Error Codes for Error 98s

Following is a brief description of the secondary error codes for error 98s for the Vision file system.

01	The file size listed in the file's header does not match the actual file size.
02	The header's next record pointer points to an area that is invalid.
03	Unique ID used to distinguish duplicate keys has already been used and cannot be used with a new key.
04	Missing tree terminator key.
05	An error was detected while performing a bulk read of a record.
06	The key being deleted from the tree was not found in the tree.
07	A child node was not found in its parent.
08	An I/O error occurred when the runtime was trying to read key information out of the file's header.
09	A pointer in a node points past the end of the file.
12	A node in the free node list was not marked as a free node.
13	A record in the deleted record list was not marked as a deleted record.
20	Non-zero key prefix on first key in node.
21	Key prefix larger than key size.
22	Key prefix or key size larger than maximum key size.
31	A record pointer in a Vision Version 3 file points to a record-chain value. In a Version 3 file, record pointers should always point to the start of a record, never to a record-chain value.
42	The unique record counter has been exhausted. Rebuild the file to correct the error.
68	A Vision 4 or 5 data segment is not found during an open.
69	A Vision 4 or 5 index segment is not found during an open.
81	Invalid data found in record header when a compressed record was read.
82	Invalid data found in record header when a non-compressed record was read.

83	When a record was read, an I/O error occurred or the record was too short.
84	When a record link was read, an I/O error occurred or the link was too small.
85	Record contains invalid record compression codes--the record would uncompress into a record that was larger than the maximum record size.
86	During a record write, a read of a record-chain value failed, probably due to an end-of-file condition.
87	Vision Version 4 or 5 detects that it is about to write a record to an area of a file that does not contain an appropriate record header. An appropriate record header indicates that a record currently does not exist at this address.
89	In Vision Version 4 or 5, on open, a data segment's internal revision number does not match the internal revision number stored in the header of the first data segment.
90	In Vision Version 4 or 5, on open, an index segment's internal revision number does not match the internal revision number stored in the header of the first data segment.
99	Vision Version 4 or 5 has tried to open the 65,537th data or index segment for this file. Vision can only support 65,536 data segments and 65,536 index segments per logical file.

6.5 Transaction Error Codes

A transaction management error is one that follows a START TRANSACTION, COMMIT, ROLLBACK or call to C$RECOVER, or one that occurs during some other file operation within a transaction (resulting in an error 9E). Error codes associated with these are stored in the TRANSACTION-STATUS register. This section lists and describes the primary and secondary transaction error codes.

6.5.1 Primary Error Codes

Following is a list of the primary error codes for the transaction management system.

01	This is returned from a ROLLBACK statement or call to C$RECOVER when an error occurs in an external routine. For more information, see **section 6.5.2, "Secondary Error Codes for Error 01."**
02	An attempt to open the log file failed because the maximum number of files per process would be exceeded. This is returned from a START TRANSACTION or call to C$RECOVER.
03	An attempt to open the log file failed because some element of the specified directory path is non-existent. This is returned from a START TRANSACTION statement or call to C$RECOVER.
04	An attempt to open the log file failed because the user has insufficient access privileges for the file. This is returned from a START TRANSACTION statement or call to C$RECOVER.
05	This indicates an operating system error that is not otherwise covered by one of the standard error conditions. You can determine the exact nature of this error by examining the value of the secondary error code.
06	This indicates that the log file is corrupted. The error is returned when the program encounters an unexpected end of file, or when an invalid transaction type code is found during recovery.
07	An attempt to open the log file failed because the file is locked (MS-DOS only). This is returned from a START TRANSACTION statement or a call to C$RECOVER.
08	This indicates that the system ran out of dynamic memory.
09	This indicates that a write failed because the disk is full.
10	This is returned from a START TRANSACTION statement or call to C$RECOVER when no log file was specified in the LOG-DIR configuration variable.
11	This is returned from a ROLLBACK or COMMIT statement when an unexpected end of file is reached while the rollback log file is being read.
12	A START TRANSACTION, ROLLBACK or COMMIT failed because the last transaction in the log file is incomplete.

13 This error is returned in the TRANSACTION-STATUS register from a WRITE, REWRITE, CLOSE, or DELETE if the file was not opened *within a transaction*. Note that, if the FILE-CONTROL paragraph for the file contains the WITH ROLLBACK phrase, all OPENs are automatically performed within a transaction.

14 This is a file-system specific error that is not one of the standard errors, and not an error returned by the operating system. The secondary and tertiary error codes indicate the exact meaning, which is file-system dependent.

16 This error is returned when the runtime is executing a START TRANSACTION while another transaction is already active.

99 This warning indicates that the requested transaction operation is not supported by a host file system. The transaction operation is still attempted for other file systems.

6.5.2 Secondary Error Codes for Error 01

The following is a list of the secondary error codes for transaction error 01.

	Secondary Error	**Corresponding file-status error**
01	operating system error (see tertiary code for system-specific error code)	30
02	illegal parameter	39,01
03	attempt to open more files than system allows	94,10
04	open mode does not allow operation	48,01 or 49,01
05	requested record is locked	99
06	index file is corrupt	98,xx
07	duplicate key where duplicates not allowed	22
08	requested record not found	23
10	disk became full while adding a new record	24
11	file locked against requested open mode	93
12	record size mismatch during rewrite	44

	Secondary Error	**Corresponding file-status error**
14	out of dynamic memory	9A
15	requested file does not exist	35
16	inadequate access permissions to file	37,07
17	requested operation not supported	9B
18	out of lock-table entries	9C
19	file-system specific error	9D

6.6 IBM DOS/VS Error Codes

IBM DOS/VS COBOL has a form of the USE statement in the DECLARATIVES section that is not normally recognized by ACUCOBOL-GT:

```
USE AFTER STANDARD ERROR PROCEDURE ON file-name GIVING
       data-name-1 [data-name-2]
```

This form is accepted by ACUCOBOL-GT when the "-Cv" option is in effect.

When an error handler introduced by this statement is invoked, the runtime puts special error codes into the eight-byte data item *data-name-1*. For more information and the list of codes, see Chapter 5, "IBM DOS/VS COBOL Conversions," in the *Transitioning Your COBOL Applications to ACUCOBOL-GT* book.

7 Utilities

Key Topics

7.1 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

7.1.1 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,"** of the *ACUCOBOL-GT User's Guide* 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 2.9, "Calling Subprograms,"** of the *ACUCOBOL-GT User's Guide* 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.

7.1.2 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.

7.1.2.1 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 2.1.17.1, "Remote file name handling,"** of the *ACUCOBOL-GT User's Guide* 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.

7.1.3 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.

7.1.4 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.

7.2 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.

7.2.1 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 7.2.3, “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 7.2.2, “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.

7.2.2 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.

7.2.3 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 of the** *ACUCOBOL-GT User's Guide.* 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

7.2.4 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.

7.2.5 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 7.2.2, "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.

7.2.6 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.

7.2.7 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.
source	is the name of an existing Vision 5 file.
destination	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.

7.2.8 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

7.2.8.1 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 7.2.8.2, "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.

7.2.8.2 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** of the *ACUCOBOL-GT User's Guide* 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.

7.2.9 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.

7.2.10 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.

7.2.11 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.

7.2.12 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.*

7.2.13 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 Version5. 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.

7.2.14 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.

7.2.15 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.

7.2.16 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.

7.2.17 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 our 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.

7.2.18 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**.

7.3 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".

7.3.1 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.

7.3.2 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.

7.3.3 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
```

7.3.4 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.

7.4 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).

7.5 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:

7.5.1 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:

```
TY  PID     Term  Client    User   Date/Time       File ID   Filename
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
```

7.5.2 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

7.6 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.

7.6.1 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 profile 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 **runcbl 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.

7.6.2 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 *or* --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 *or* --output	To give the report file a name other than the default, “acumon.rpt,” use this flag.
-c *or* --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 *or* --sort-name	Sort data in the report file alphabetically by program and paragraph name. (By default, the report is sorted by time.)
-q *or* --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.

7.6.3 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 7.3.0 (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.

7.6.4 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 <Runtime-Version> HTML tag 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).

The <Run-Date> tag marks the date and time of the run that produced the profile data.

The <Has-Timer> tag 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).

When available, <User-Time-Msecs> shows the number of milliseconds of CPU time spent processing user code for this run.

When available, <System-Time-Msecs> 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:

- The <Program-Name> tag 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 I$IO. Note that this counts opens, not files (so if you open the same file ten times in the course of a run, that counts as “10” and not “1”). Some machines open files much more slowly than others, so a large number here usually suggests a potential performance issue.
- <File-Reads> lists the number of record read attempts. READ, READ NEXT, and READ PREVIOUS all count here and are not distinguished.
- <File-Writes> indicates the number of records written.
- <File-Rewrites> gives the number of records rewritten by the program.
- <File-Deletes> shows the number of records deleted by the program.
- <File-Starts> lists the number of file positions made using the START statement or the I$IO 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.

7.7 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*.

7.7.1 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 7.7.3, "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
```

7.7.2 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. The following sections provide details on these functions. Refer to **section 7.7.3, "Code Sample,"** for **AcuSort** sample code.

7.7.2.1 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.

7.7.2.2 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 **AcuSor**t 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.

7.7.2.3 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 7.7.3, "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 2.1.9 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.

7.7.2.4 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\ordrdetl org ix
   record f 143
   key (1, 36, p)
   give c:\acuprod\data\ordrdet_sorted
```

the input file is "ordrdetl" 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 7.7.3, "Code Sample."**

7.7.2.5 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 7.7.2.3, "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.

7.7.3 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\ordrdetl 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
```

7.7.4 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.

7.8 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.

7.8.1 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.

7.8.2 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.

7.8.2.1 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	Instructions you wish to give to the preprocessor. **Boomerang** includes several keywords you can use to specify several basic files: • 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	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.

Field	**Description**
Required precompiler extension	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.

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 “-Sf” 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
```

7.8.2.2 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
```

7.8.2.3 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.

7.8.2.4 Step 4: Starting the Server

To start the server, issue the following command:

```
boomerang -start -c boomerang.cfg -e boomerang.err
```

7.8.3 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 #	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. “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. “2” provides information about client requests. The buffer is flushed to the error file when the buffer is filled or **Boomerang** terminates. “3” provides the information described for “1” and “2”. “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.) “6” is equivalent to “2”, but the tracing buffer is flushed to the error file each time a client connection is requested. “7” provides the information described for “5” and “6”.

7.8.4 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 (our IDE product). 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.

7.8.5 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 7.8.6** 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>	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.
-COPY	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.

7.8.6 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
```

8 Shared Memory

Key Topics

8.1 Shared Memory Management with acushare

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. For information on **acushare**'s role in shared memory management in UNIX/Linux environments, see **section 8.2, "Using Shared Memory."** Also see **Section 8.3, "Using acushare,"** for comprehensive instructions on the use of **acushare**.

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

- Versions of **acushare** shipped with ***extend*7** products are compatible with all Version 7.*x* products but cannot be used with previous versions.
- Versions of **acushare** shipped with pre-***extend*7** products can be used with all pre-***extend*7** products but not with any ***extend*7** products.
- On the same system, you can run both ***extend*7** and pre-***extend*7** versions of **acushare** concurrently.

8.2 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. For more information, see **section 8.2.1, "Indicating Programs to Share."**
3. Start the **acushare** program. After it's started, you can use the program to perform other tasks.

8.2.1 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. Further code segments are then placed in conventional memory. 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.

8.3 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 8.2, "Using Shared Memory,"** for a description of **acushare**'s role shared memory management.

Acushare has several command line formats. They include:

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

These formats are described in the following sections.

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.

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.

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. For more information, see **section , “acushare -clean.”**

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.

acushare -version

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

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 8.1.0
(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
```

8.4 Runtime Error Handling

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.

If runtime warning messages are enabled (see the WARNINGS configuration variable and the "-w" command line option), certain errors cause a warning message to be printed. 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.

Index