A Guide to Interoperating with ACUCOBOL-GT®

Version 8.1.3

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

E-01-UG-100501-Interop-8.1.3

Contents

Chapter 1: Introduction

Chapter 2: Working with Java Technology

Chapter 3: Working with Windows Technologies

Chapter 4: Using ActiveX Controls and COM Objects

Chapter 5: Working With .NET Assemblies

Chapter 6: Working with C and C++ Programs

Chapter 7: Deploying ACUCOBOL-GT Applications on the Web

Chapter 8: Accessing ACUCOBOL-GT Applications from Mobile Devices

Chapter 9: Working with Transaction Processing Systems

Chapter 10: Working with Messaging Middleware

Chapter 11: Working with Non-Vision Data

Index

1 Introduction

Key Topics

1.1 Introduction

The ***extend***® family of technologies includes many opportunities for extending and enhancing your legacy applications, allowing you to integrate that code with other enterprise information technology components regardless of their platform, language, database, or network infrastructure. You can combine our technologies in a number of ways to solve your business issues, while protecting your valuable investment in legacy applications. *A Guide to Interoperating with ACUCOBOL-GT* provides information to help you facilitate this integration as the need arises.

1.2 Documentation Overview

This manual describes various methods that allow your ACUCOBOL-GT applications to interoperate with technologies that provide enhanced capabilities and functionality. Topics include

- "Chapter 2: Working with Java Technology" provides information that can help your ACUCOBOL-GT applications interoperate with Java. The chapter includes details about methods for calling COBOL from Java and calling Java from COBOL. You can also learn about mapping Java data types, J2EE application server technology, and Web services.

- In "Chapter 3: Working with Windows Technologies," you learn how to leverage Microsoft Windows technologies in your ACUCOBOL-GT programs. The chapter includes information about calling dynamic link libraries (DLLs), accessing the Windows Application Programming Interface (API), and using some Windows-specific ACUCOBOL-GT features.

- "Chapter 4: Using ActiveX Controls and COM Objects" describes how to include Microsoft ActiveX controls and COM objects in your ACUCOBOL-GT program.

- "Chapter 5: Working With .NET Assemblies" describes how to call .NET assemblies from your ACUCOBOL-GT program and how to invoke COBOL from a .NET assembly. It also discusses interacting with .NET Web services from COBOL.

- "Chapter 6: Working with C and C++ Programs" provides information about how your ACUCOBOL-GT applications can interoperate with C and C++ programs. Learn about direct calls from C to COBOL and from COBOL to C, interfacing to COBOL from C via the ACUCOBOL-GT C API, and matching C data items.

- In "Chapter 7: Deploying ACUCOBOL-GT Applications on the Web," you can learn about our various technologies that help you deploy your ACUCOBOL-GT applications on the Internet. The chapter includes descriptions of the ACUCOBOL-GT Web Thin Client and Web Runtime, our Common Gateway Interface (CGI) extensions, and more.

- "Chapter 8: Accessing ACUCOBOL-GT Applications from Mobile Devices" explores the basic concepts of accessing COBOL programs from mobile devices running non-COBOL applications. You receive background information on mobile terminology, infrastructure, and platform trends. Some mobile system design issues are covered, and a sample mobile system with a COBOL back end is described.

- "Chapter 9: Working with Transaction Processing Systems" discusses how ACUCOBOL-GT can interoperate with online transaction processing (OLTP) systems. You learn about transaction processing in general, and then find out how ACUCOBOL-GT can work with specific transaction processing technologies.

- In "Chapter 10: Working with Messaging Middleware," you learn how to integrate ACUCOBOL-GT applications with message passing middleware, specifically IBM WebSphere MQ (formerly MQ Series).

- In "Chapter 11: Working with Non-Vision Data," you learn how ACUCOBOL-GT applications can interoperate with external data sources, including XML documents, SQL databases, ODBC-compliant data sources, C-ISAM and KSAM files, and file systems that use an EXTFH interface to access files.

Other manuals in the ***extend*** documentation set are referenced in this book as well. These manuals may be accessed from support section of the Micro Focus website or installed from your product media.

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.).

1.3 The *extend* Family of Products

Your strategy for interoperability may include one or more members of the ***extend*** family of products. Brief descriptions of these technologies appear in the following sections.

Acu4GL®

ACUCOBOL-GT uses Acu4GL libraries to access information stored in relational database management systems (RDBMSs). Data dictionaries generated by the compiler guide the libraries in mapping the field names and data types that are passed between COBOL and the database engine. The essence of Acu4GL libraries is that standard COBOL I/O statements are used to access databases.

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

ACUCOBOL-GT

ACUCOBOL-GT is an ANSI 1985 COBOL compiler designed to provide a powerful development environment for a wide range of computers. Fast compile speed, clear error messages, and a multi-window source level debugger work together to provide a high performance, easy to use COBOL development platform. Portable object code, a generic interface to a variety of file systems, and a device-independent terminal interface help to simplify the distribution of applications developed with ACUCOBOL-GT.

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

AcuXDBC™

AcuXDBC is a data management system, designed to integrate ACUCOBOL-GT data files into a relational database-like environment. AcuXDBC enables you to apply SQL and relational database concepts to your COBOL data sources resulting in data that is accessed and managed in much the same way as many of today's popular relational database management systems.

AcuXDBC lets you retrieve and update ACUCOBOL-GT's Vision indexed files, relative files, and sequential files from Windows-based applications including Microsoft Word, Excel, and Access. Business Intelligence tools such as Crystal Reports® Professional, and custom applications developed in ODBC-supported environments such as Visual Basic® are supported as well. With the enterprise edition, you can also retrieve data through Java applications that utilize JDBC standards. Direct SQL access to your ACUCOBOL-GT data is available in both the Windows and UNIX environments.

AcuXDBC Server is an add-on to AcuXDBC that supports remote processing on a UNIX/Linux or Windows server.

AcuConnect®

AcuConnect is a client/server technology that is an integral part of our distributed computing solution. AcuConnect lets you implement a client/server system in which the client piece can be as "thin" or as "thick" as you need.

AcuConnect has two deployment environments. With AcuConnect's distributed processing deployment, users can distribute application logic between client and server machines in a way that best suits their needs. AcuConnect users can also take advantage of our Thin Client technology,

which lets you run the user interface (UI) portion of your application on a graphical display host while the rest of the application and data reside on the server.

AcuServer®

AcuServer is an add-on module that provides remote file access services to ACUCOBOL-GT applications running on most UNIX, Linux, and Windows TCP/IP based networks. AcuServer provides the ability to create and store indexed, relative, and sequential data files on any UNIX, Linux, or Windows NT/2000/2003/2008 server equipped with AcuServer. It also provides full function remote access from supported clients to indexed, relative, sequential, and object files stored on an AcuServer server.

AcuSQL®

AcuSQL is an add-on tool that supports the inclusion of embedded SQL (ESQL) statements in ACUCOBOL-GT program source code. The AcuSQL pre-compiler, in combination with the AcuSQL runtime library, allows your ESQL COBOL programs to access IBM® DB2®, Microsoft® SQL Server, and ISO/ANSI SQL92 compliant data sources.

1.4 Technical Services

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

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

For worldwide technical support information, please visit:

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

2 Working with Java Technology

Key Topics

2.1 COBOL/Java Interoperability

Businesses want to deploy Java technology for a variety of reasons. They include:

- The flexibility of JavaServer Pages™ for graphical front ends, Internet portals, mobile devices, etc.
- The availability of an enterprise standard, Java2 Enterprise Edition (J2EE™)
- The promise of application server technology from vendors such as BEA, IBM, Sun, and Oracle

Despite the opportunities that Java affords, many businesses recognize that their legacy applications have high value to them. They know that COBOL runs their business. Their COBOL programs have been time-tested, fine-tuned, and proven reliable and scalable. They have been custom-fitted to their business processes.

Rather than replacing COBOL with Java, many organizations integrate their legacy assets with the newer Java components.

Java scenario

A bank has a mission-critical loan processing application written in COBOL. The bank wants to make the application accessible on a Web site as part of a customer loan portal. The portal will be supported by an application server running J2EE applications. The bank wants to take the existing COBOL application and integrate it with the J2EE applications so that requests coming in through the application server will be routed to and processed by the COBOL application.

2.2 Calling COBOL from Java

With ACUCOBOL-GT®, there are many ways to achieve interoperability with Java. You can call COBOL directly from a Java command line or from a Java application. If calling from a Java application, there are several ways to do this:

- Use ACUCOBOL-GT's Java compiler options to generate Java classes that call your ACUCOBOL-GT program. Java programmers can then invoke these classes as they would any native Java code.

- Use the Java native interface, "CVM.jar", to interact with the COBOL program at the API level. "CVM.jar" contains a singleton class, CVM, that encapsulates the ACUCOBOL-GT runtime. With the CVM, the Java programmer can programmatically instantiate an instance of the ACUCOBOL-GT runtime and invoke a COBOL program. The programmer can use other classes or methods of CVM to specify runtime options and program options.

- Use the C$SOCKET library routine to facilitate interprocess communication via sockets. C$SOCKET is a low-level option, but it is very flexible.

- Use our CGI extensions. The Java programmer can use CGI to call a remote COBOL procedure through a Web server.

- Use the Java Native Interface (JNI) to call the ACUCOBOL-GT runtime dynamic link library (DLL) in Windows or shared library in UNIX.

- Use named pipes to pass data between your COBOL and Java applications if they reside on the same host machine. Passing data through named pipes is a low-level solution requiring the development of C code. Named pipes are a good option for legacy applications that perform strictly file I/O.

- Use AcuXDBC™ to access COBOL Vision data from a Java Database Connectivity (JDBC)-enabled application.

2.2.1 Calling COBOL from a Java Command Line

You can call COBOL from a Java command line and without having to write a Java program. This is done by specifying a Java command line that calls the ACUCOBOL-GT Java Native Interface "CVM.jar", specifies the path to the ACUCOBOL-GT runtime, and passes an ACUCOBOL-GT runtime command.

Syntax

```
java -cp path-to-CVM.jar com.acucorp.acucobolgt.runcbl
--acugt path-to-AcuGT-runtime [java-options]
runtime-command-line path-to-cobol-program
```

Syntax Definitions and Parameters

java -cp *path-to-CVM.jar* com.acucorp.acucobolgt.runcbl	Required syntax for calling the AcuGT CVM.
--acugt	Required parameter that brings in the AcuGT runtime.
path-to-AcuGT-runtime	Required syntax that specifies the location of the AcuGT runtime.
java-options	Optional parameters described in the next table.
runtime-command-line	Any valid runtime command line options. See the *ACUCOBOL-GT User's Guide*, Section 2.3 for details on runtime command lines.
cobol-program	The name of the COBOL program to run.

java-options	**Description**
--log	Creates a log
--logfile	Creates a log file
--verbose	Creates detailed log

java-options	**Description**
--lib	There is a default list of libraries that automatically get loaded for both UNIX and Windows. On most systems the Java runtime will determine the correct list to load. The default library lists on UNIX is: libacme,libacuterm,libvision,libclnt,libaxml,libsrvmgmt,libruncbl The default list on Windows is: acme,atermmgr,avision5,libexpat,axml32,wrun32 On some systems such as HP-UX it may be required to specify these files manully by using the --lib option. The library names must be separated by either commas or semi-colons.The library names should also be specified in the order given above. In most UNIX and Windows cases, it should not be necessary to use this option.
--libext	Used to specify the library name's file extension. This option may be needed on certain systems such as HP-UX. In most UNIX and Windows cases, it should not be necessary to use this option.

Syntax Example

The following is a sample Java command line for calling a COBOL program named "tour".

```
Java -cp "C:\Program
Files\Acucorp\Acucbl811\AcuGT\bin\CVM.jar"
com.acucorp.acucobolgt.runcbl --acugt "C:\Program
Files\Acucorp\Acucbl811\AcuGT\bin" -dle xxx "C:\Program
Files\Acucorp\Acucbl811\AcuGT\bin\tour"
```

2.2.1.1 Static Method RunCbl

It also possible to call the ACUBOBOL-GT Java command line functionality from another Java program by calling the static method RunCbl in runcbl class, and passing a Java String array that contains a valid runtime command line and logging switches.

There are two versions of the static method RunCbl.This version takes command line arguements only:

RunCbl(String[])

This version takes command line arguemnts and linkage section parameters as object array:

RunCbl(String[], Object[])

2.2.2 Using the Java Compiler Options

There are two compiler options that make it easy for you to provide COBOL services to a Java program:

Compiler Option	Description
--javaclass	Generates a Java class that calls your COBOL program
--javamain	Generates a Java class with a main method

–javaclass

When you specify the "--javaclass" option at compile time, the compiler generates a ".java" file in addition to a ".acu" file. The ".java" file has the same prefix as the ".acu" file and is placed in the same directory. This ".java" file is a Java class that calls the COBOL program being compiled. Java programmers can then invoke this class as they would any native Java code.

–javamain

Same as “--javaclass” except “--javamain” generates a class with a main method added.

2.2.3 Using the Java API, “CVM.jar”

Another way to call COBOL from Java is to use the application programming interface (API) contained in the Java archive, “CVM.jar”. This interface can be used by Java developers to call COBOL functionality (programs, entry points, etc.) from their Java class.

Note: This feature is available only to shared library or DLL versions of the ACUCOBOL-GT runtime. On Windows, the DLL version is automatically available. To see if this feature is available to you on UNIX, type “ls lib” from the ACUCOBOL-GT installation directory. If you see the filename “libruncbl.so” or “libruncbl.sl”, then the feature is available. For instructions on creating a shared library for HPUX 11.0, see section 2.2.3.8.

“CVM.jar” consists of two main classes:

Class	Description
CVM	CVM is a singleton class representing the ACUCOBOL-GT runtime. This class allows Java developers to programmatically manage the ACUCOBOL-GT runtime, giving them low-level control of COBOL objects from Java.
CALL_OPTIONS	This options class is used for setting options for each called COBOL program.

Note that you can call COBOL from Java locally or remotely. You can even have the runtime execute remotely without a COBOL object executing on the client. All you need on the client is a Java program and a runtime. For this to work, you set CODE_PREFIX in the configuration file that you provide with the runtime initialization to point to a remote server hosting your COBOL application. The remote server must also be running AcuConnect.

AcuConnect is able to execute a COBOL object remotely and share data with the local runtime. For more information on executing remote COBOL programs with AcuConnect, please refer to the *AcuConnect User's Guide.*

2.2.3.1 CVM class

CVM is a Java class representing the ACUCOBOL-GT runtime. The CVM class exposes public methods for setting runtime options, calling and cancelling programs, getting object libraries, and much more.

The following table contains a description of each method. Please note that the get method returns the current value of a particular property or string. The set method sets the string or property value. For example, "setErrorsOut" sets the name of the file to which to send error messages, and "getErrorsOut" returns the filename that is currently set for the error log.

Boolean properties like TerminalInit are set to false by default. If you want to set a boolean property to true, then you call the set method for that property. For example, TerminalInit is set to false by default, meaning that terminal initialization is not inhibited. If you want to inhibit terminal initialization, set TerminalInit to true by calling "setTerminalInit" passing in true. Call "getTerminalInit" to see what boolean value is currently set for this property.

Public Method	**Description**
initialize(RT_OPTS options)	Initializes the ACUCOBOL-GT CVM
initialize(String cmdLine)	Initializes the CVM with command-line options
callProgram(String name, Object params[], CALL_OPTIONS options)	Calls the named COBOL program using specified parameters and program options
cancelAllPrograms()	Cancels all programs
cancelProgram(String name)	Cancels the named program and holds it in memory
unloadAllPrograms()	Empties memory of all programs

Public Method	**Description**
unloadProgram(String name)	Empties memory of the named program
shutdown()	Shuts down the CVM
CVM GET_INSTANCE()	Returns the instance of the CVM in this process
CVM GET_INSTANCE(String logPropertiesFile)	Specify a Java String that is the name of the logging properties file. This enables you to use a different file which is not the default Java logging properties file.
CVM GET_INSTANCE(String logPropertiesFile, String libLoc, String ext)	The first String parameter has the same meaning as the previous GET_INSTANCE. The second String parameter is the location of the Acu shared libraries. On windows, this is where the acu dlls are installed ("c:\Program Files\Acucorp\Acucbl810\AcuGT\bin. The third parameter is the extension of the shared libraries – on windows the extension is ".dll", on linux it is ".so", and on some versions of HP-UX it is ".sl".
setLog(Logger log)	Overrides the default log with a user-specified log

StatusInfo GetStatusInfo()	Checks the status of a called COBOL program that has finished running. Use StatusInfo as follows:
	`class StatusInfo {`
	`  public long cobol_return_code;`
	`  public int exit_code;`
	`  public int signal_number;`
	`  public int call_error;`
	`  public String exit_msg;};`

getCVMError()	Gets the last error message of the CVM object class

Public Method	Description
getLastErrorMsg()	Returns the last error message string from the runtime
get/setSwitches()	Gets or sets the list of Special Names switches to turn on
get/setConfigFile()	Gets or sets an alternate configuration file
get/setErrorsOut()	Gets or sets an error messages file
get/setErrorsAppend()	Gets or sets a file to append error messages to
get/setKeyFile()	Gets or sets a keyboard input file
get/setImport()	Gets or sets a variable for importing graphical screens
get/setPlays()	Gets or sets a file of input keystroke scripts
get/setDisplayOut()	Gets or sets a file for display output
get/setDisplayAppend()	Gets or sets a file to append display output
get/setDebugCmds()	Gets or sets a file containing debugger commands
get/setTerminalOut()	Gets or sets a file to capture terminal output
get/setObjLib()	Gets or sets an object file library
get/setEmbeddedLib()	Gets or sets a configuration file from a COBOL object library
get/setTerminalInit()	Inhibits terminal initialization
get/setCGIWarnings()	Suppresses warning messages in CGI programs
get/setIgnoreSignals()	Ignores terminal hang-up signals
get/setListConfig()	Lists contents of the configuration file
get/setNoSaveDebug()	Prevents the debugger from reading and writing adb
get/setSafeMode()	Runs in safe mode

Public Method	**Description**
get/setNonNumeric()	Suppresses warnings when non-numeric data is used as numeric data
get/setExtendedError()	Displays extended error codes for file error “30”
get/setDumpMem()	Dumps memory for memory access violations
get/setThrowErrors()	Displays error message text in a MessageBox
get/setCharToGui()	Converts character screens to GUI equivalent
get/setZipErrorFile()	Compresses the error file
get/setLinkageLength()	Disables Linkage item length test

To set options in the CVM class (i.e., to set runtime options), use the specific “setOption” method such as “setConfigFile” and the value to set. You can also call “setOption” with the option name passed as a string.

Call “cvm.initialize” after setting options. After “initialize” is called, setting options has no effect until you call “initialize” again.

2.2.3.2 CALL_OPTIONS class

The CALL_OPTIONS class represents the options for each called COBOL program. If you want to pass program options to the “cvm.callProgram” method that runs the COBOL program, create a CALL_OPTIONS object, then add options to it. The CALL_OPTIONS class is structured as follows:

```
class CALL_OPTIONS {
   public String GetOption( String key );
   public void SetOption( String key, String value );
};
```

Valid call options include:

- cache – an unsigned value that determines whether the runtime should maintain the program in a memory cache after it has been canceled. This parameter is useful for application servers like CICS that allow each program to be configured as resident or nonresident.

  If "cache" is FALSE ("0"), the runtime removes the program from memory and sets the Working-Storage to its initial state on subsequent calls. If "cache" is TRUE ("1"), it marks the program as "cached" and resets Working-Storage for the next call; the program remains in memory according to the caching rules. For information on managing logical and physical cancels that may affect the behavior of "cache", refer also to the LOGICAL_CANCELS configuration variable in Appendix H of the ACUCOBOL-GT documentation set.

- debug_method (-dn) where n is 0-3:

  0 = ADM_NONE for no debugging
  1 = ADM_XTERM to debug using a new xterm
  2 = ADM_TERMINAL to debug using an existing terminal through runcbl
  3 = ADM_THINCLIENT to debug using a waiting thin client

  Based on the debug_method selected, you may need to also specify debug_method_string.

- debug_method_string – a char* that sets the display setting for the debug_method

  For ADM_XTERM, set to the Xservername:displaynumber of the xterm or set to NULL to allow the xterm to use the default display given by the DISPLAY environment variable. For ADM_TERMINAL, set the string to the tty device on which you will execute runcbl. For ADM_THINCLIENT, set to client:port where the client is the host on which acuthin is executing and port is the port on which it is listening.

Note: The value of debug_method_string overrides the value, if any, in the DISPLAY configuration variable for the xterm.

See section 9.8, "Background Debugging Options," for more information on background debugging.

- no.stop – an unsigned input value that, when set to “1”, causes STOP RUN to be ignored.

- program.name – the name of the COBOL program being called

The CALL_OPTIONS class contains a linkage_signature property that describes the data in the linkage section. For example, a linkage_signature of “X45X20SSIJ” describes two PIC X items of 45 and 20 bytes respectively, two shorts, an integer, and a long.

The linkage_signature ensures that there is enough memory for Java strings to get passed in, even when the Java string has a shorter length than the PIC X data item in the linkage section. For example, a Java string of length 10 can be passed into a PIX X(45) data item. In this case, 45 bytes are allocated to memory, not 10 bytes.

2.2.3.3 Sample use case

```
CVM cvm = CVM.GET_INSTANCE();
cvm.setErrorsOut("/tmp/errfile");
cvm.setConfigFile("c:/myproject/config");
cvm.initialize();

CALL_OPTIONS co = new CALL_OPTIONS();
co.setOption("debug_method", "1");

Object objInt = new Integer(1);
Object objString = new String("Test String Parameter");
Object params[] = {
objInt,
objString
};
cvm.callProgram("TestJavaToCobol", params, co);
cvm.cancelProgram("TestJavaToCobol");
cvm.shutdown();
```

2.2.3.4 Configuration and deployment

To call COBOL from Java using the CVM, perform the following steps:

1. Install and correctly configure the ACUCOBOL-GT runtime. Optionally, install AcuBench® if the system will also be used for COBOL development.

2. Install and correctly configure a Java Runtime Environment (JRE) Version 1.4.2 or later. Optionally, install a J2SE Software Developer's Kit (SDK) if the same system will also be used for Java development.

3. Place the path to the JRE /bin directory in the PATH environment variable. Here is an example:

   ```
   PATH =D:/j2sdk1.4.2_04/bin.
   ```

   If the ACUCOBOL-GT installation directory is not the current directory, then that directory should also be placed on the path. The runtime path must be correctly configured so that a call to LoadLibrary("wrun32.dll") or loading the shared library "libruncbl.so" succeeds.

4. Place the path to "CVM.jar" in the CLASSPATH environment variable. Also ensure that the class or JAR (Java Archive) file that contains the declaration of main is included in the classpath. For JAR files, include the filenames in the classpath. For class files, include the directory where the class files reside. Here is a Windows example:

   ```
   CLASSPATH=d:\cobol7\bin\acuUtilities.jar;d:\cobol7\bin
   \CVM.jar;c:\cobol7\JavaProject
   ```

   For UNIX platforms, use a colon as a delimiter instead of a semicolon.

5. Add the location of the runtime DLLs and shared libraries to the variable LD_LIBRARY_PATH. For example, on Windows:

   ```
   LD_LIBRARY_PATH=C:\Program
   Files\Acucorp\Acucbl800\AcuGT\bin
   ```

   On UNIX or Linux, the shared libraries are located in /AcuGT/lib.

6. Do one of the following:

   a. Place a copy of the COBOL program to be called in the same directory as the ACUCOBOL-GT runtime. This would be the compiled ".acu" file that contains the COBOL program.

   b. Pass the fully qualified filename of the COBOL program to be called to the runtime.

   c. Use a configuration variable to identify the location of the COBOL program.

7. Ensure that all configuration options located in the configuration file are set up correctly. This includes the location of the JRE, preloading the JVM, and the command line that will be passed to the JVM.

8. The Java class being used to call COBOL must provide a main function such as this:

```
public static void main(String[] args)
```

9. The Java program calling COBOL must include an import statement that imports the ACUCOBOL-GT Java class that is used. Here is an example:

```
import com.acucorp.acucobolgt.*;
```

10. The Java program calling COBOL must declare two objects: one of type CVM and one of type CALL_OPTIONS. The following is sample Java code that shows how to call a COBOL program, passing two parameters, using the “CVM.class”:

```
CVM cvm = CVM.GET_INSTANCE();
cvm.setErrorsOut("/tmp/errfile");
cvm.setConfigFile("c:/myproject/config");
cvm.initialize();

CALL_OPTIONS co = new CALL_OPTIONS();
co.setOption("debug_method", "1");

Object objInt = new Integer(1);
Object objString = new String("Test String Parameter");
Object params[] = {
objInt,
objString
};
cvm.callProgram("TestJavaToCobol", params, co);
cvm.cancelProgram("TestJavaToCobol");
cvm.shutdown();
```

11. The COBOL program being called must provide a Linkage Section that matches the order and type of the Java parameters passed in the Java Object array. This is done by the COBOL programmer. Here is an example of a Linkage Section that does this for the above program:

```
linkage section.
77 test-integer-parameter usage is signed-int.
77 test-string-parameter pic x(21) value spaces.
```

2.2.3.5 Security

The CVM class in "CVM.jar" supports the default security manager class in Java, known as java.lang.SecurityManager. For information on this class or overriding the default security manager, please refer to the Java API documentation provided by Sun Microsystems.

2.2.3.6 Example of Java calling COBOL

Java program

```
import com.acucorp.acucobolgt.*;

public static void main(String[] args) throws IOException {
  try{
     CVM cvm = CVM.GET_INSTANCE();
     cvm.setErrorsOut("/tmp/errfile");
     cvm.setConfigFile("c:/myproject/config");
     cvm.initialize();

     CALL_OPTIONS co = new CALL_OPTIONS();
     co.setOption("debug_method", "1");

     int intParam = 1;
     Integer objInt = new Integer(intParam);

     byte byteParam = 'a';
     Byte objByte = new Byte(byteParam);

     char charParam = 'b';
       Character objChar = new Character(charParam);

     Object params[] = {
       objInt,
       objByte,
       objChar
     };
     cvm.callProgram("TestJavaToCobol", params, co);
     cvm.cancelProgram("TestJavaToCobol");

     objInt = (Integer)params[0];
     objByte = (Byte)params[1];
     objChar = (Character)params[2];
```

```
      System.out.println("COBOL changed value to " +
         objInt.intValue());
      System.out.println("COBOL changed value to " +
         objByte.byteValue();
      System.out.println("COBOL changed value to " +
         objChar.charValue());

      cvm.shutdown();
    } catch (Exception e){
     e.printStackTrace();
    }
}
```

COBOL program

```
identification division.
program-id. TestJavaToCobol.

data division.
working-storage section.
COPY "java.def".
01 status-val pic 9(02) value zero.

linkage section.
01 integer-parameter usage is signed-int.
01 byte-parameter pic x.
01 char-parameter pic x.

procedure division
   using integer-parameter, byte-parameter,
   char-parameter.

main-logic.
   move 3 to integer-parameter.
   move "d" to byte-parameter.
   move "e" to char-parameter.
   exit program.
```

2.2.3.7 Sample programs for Java interoperability

12. Your ACUCOBOL-GT distribution includes Java interoperability sample programs. You will find them in the \acugt\sample\java\ directory where ACUCOBOL-GT is installed. You will also find a text file containing detailed instructions on runnning these sample programs.

2.2.3.8 Building a Shared Library for HP-UX 11.0

Because we do not offer a shared library distribution of ACUCOBOL-GT on HP-UX 11.0 or before, customers who want to use the Java API feature need to create the shared library manually. To do so, follow these instructions:

1. Add the following five lines to the end of $ACUCOBOL/lib/Makefile. Note that the whitespace before the fourth and fifth lines must be tabs, not spaces.

```
SHAREDLIB_LDFLAGS = -s +b $(ACUVERSPATH)/lib:$(ACUPATH)/lib:.:../
stdlib:/usr/lib:/lib
libruncbl.sl: amain.o $(SUBS)
      ld -b $(SHAREDLIB_LDFLAGS) -o libruncbl.sl amain.o $(SUBS) \
             $(RUNTIME_LIB) $(LIBS) $(SYS_LIBS)
```

2. Run “make libruncbl.sl” from the $ACUCOBOL/lib directory.

   This creates a file named “libruncbl.sl” that can be loaded by the CVM class when calling COBOL from Java. You can also add “libruncbl.sl” to the “clean” target so that “make clean” will remove “libruncbl.sl”.

2.2.4 Using C$SOCKET

If desired, you can facilitate communication between Java and COBOL programs on a socket level. ACUCOBOL-GT includes a library routine, known as C$SOCKET, to perform interprocess communication.

When calling COBOL from Java:

1. The COBOL programmer uses the C$SOCKET routine to create a server socket (op-code 1) and wait for and accept a connection from the Java client (op-code 2).

2. The Java programmer creates a socket, connects via TCP/IP to the port of the COBOL program, and writes data to it.

3. Via C$SOCKET, the COBOL program reads the data (op-code 6), processes it, and returns data to the socket (op-code 5).

Of course, because the data format is totally open and undefined, the COBOL and Java programmers must agree on a common format.

Following is sample code to demonstrate this capability:

```
*The following code creates a server socket.
CALL "C$SOCKET" USING AGS-CREATE-SERVER, 8765
GIVING SOCKET-HANDLE-1.

*The following code waits for a connection.
CALL "C$SOCKET" USING AGS-NEXT-READ, SOCKET-HANDLE-1,
TIMEOUT.

*If have a connection request. Accept the connection.
CALL "C$SOCKET" USING AGS-ACCEPT, SOCKET-HANDLE-1.

*Read data from the connecting socket.
CALL "C$SOCKET" USING AGS_READ, SOCKET-HANDLE-2,
SOCKET-IN, IN-DATA-LENGTH
GIVING READ-AMOUNT.

*Write outgoing data back to the client socket:
CALL "C$SOCKET" USING AGS-WRITE, SOCKET-HANDLE-2,
SOCKET-OUT, OUT-DATA-LENGTH.
```

Refer to Appendix I in *ACUCOBOL-GT Appendices* for information on the C$SOCKET library routine.

2.2.5 Using ACUCOBOL-GT's CGI Extensions

ACUCOBOL-GT offers extensions designed to simplify communication with Web servers using the Common Gateway Interface (CGI) standard. These CGI extensions can be used to connect a Java program to an ACUCOBOL-GT program.

You develop a CGI program to act as an interface between the Web server and the ACUCOBOL-GT program. The CGI program can be written in ACUCOBOL-GT. (Section 4.5 of *A Programmer's Guide to the Internet* details how you accomplish this.)

From Java, you then open an HTTP connection to the Web server with a URL. The URL must have a pointer to the CGI program in it, encoded using CGI encoding.

Through CGI extensions to ACCEPT and DISPLAY syntax, your CGI program accepts CGI input data from the Java program; launches or subsumes your ACUCOBOL-GT application; and generates HTML, WML, or XML output forms from the results—whatever the Java program requires. The output could be considered a service, it could use SOAP, or it could be simple markup language output.

When you place your CGI program and ACUCOBOL-GT application on the Web server, along with the necessary configuration, license, and data files, your ACUCOBOL-GT application becomes immediately available to end users of the Java application. Conceptually, you're using CGI to do a remote procedure call.

Refer to Chapter 4 of *A Programmer's Guide to the Internet* for full details on using ACUCOBOL-GT's CGI syntax.

2.2.6 Using the Java Native Interface (JNI)

Java programs can also call COBOL programs through a C calling interface known as the JNI. You can use JNI to call the ACUCOBOL-GT runtime DLL in Windows or a shared library that contains COBOL code routines in UNIX.

Windows

To simplify the process of calling an ACUCOBOL-GT program from other programming languages in a Windows environment, the ACUCOBOL-GT runtime is encapsulated in a DLL file, "wrun32.dll".

To call the ACUCOBOL-GT runtime DLL from JNI, you add declarations to the source program for the DLL's initialization, shutdown, and call libraries. Then you call those libraries to initialize the runtime, call the COBOL program, and shut down when you are finished.

For more information on calling the ACUCOBOL-GT runtime DLL, refer to Chapter 3 of this guide.

UNIX

To access COBOL from Java in UNIX environments, you can place native COBOL code routines in shared libraries and call them from your Java programs via JNI.

2.2.7 Using Named Pipes

Another way to pass data between COBOL and Java programs is through named pipes. Named pipes are a method for exchanging information between two unrelated processes.

Note: To communicate via named pipes, the COBOL and Java programs must be on the same host machine.

Technically, named pipes are files with known pathnames. Because a named pipe is associated with a pathname, unrelated processes can open the file to begin communications with one another. Because a Java program can open a named pipe just as it would a normal file, no special Java or JNI code is required. By opening the file for reading, a process has access to the reading end of the pipe, and by opening the file for writing, a process has access to the writing end of the pipe. In effect, named pipes allow independent processes to "rendezvous" their I/O streams.

Named pipes can be created in two ways—via the command line or from within a program.

In UNIX, to create a named pipe with the file named "npipe" you can use the following command on the command line:

```
% mkfifo npipe
```

Alternatively, you could create the named pipe from within your program using:

```
int mkfifo(const char *path, mode_t mode)
```

where “path” is the path of the file and “mode_t” is the mode (permissions) with which the file should be created.

A named pipe can be opened using the open( ) system call or the fopen( ) standard C library function. (Refer to Chapter 6 of this guide for information on interfacing ACUCOBOL-GT programs to C routines.)

As with normal files, if the call succeeds, you get either a file descriptor or a “FILE” structure pointer, depending on how you opened the file. You can then use this information for reading or writing, depending on the parameters you passed to open( ) or fopen( ).

Reading from and writing to a named pipe are very similar to reading from and writing to a normal file. You can use the standard C library function calls read( ) and write( ).

Named pipes can also be used on Windows systems. You create Windows pipes with the CreateNamedPipe( ) API. You can then use the OpenFile( ) API to access the other end of the newly created named pipe.

Although named pipes can be very effective for communicating between COBOL and Java applications, bear in mind the following issues:

- Named pipes work only for processes on the same host machine.
- Named pipes can be created only in the local file system of the host.
- Named pipe data is a byte stream, and no record identification exists.
- Named pipes provide only a half-duplex flow of data. They are also known as “fifos” for their method of “first in, first out” communication. To establish full-duplex communication, you must create and manage two pipes, which can be complicated and result in file deadlocks if you are not careful.

2.2.8 Using AcuXDBC

If you want to access COBOL Vision data from a Java Database Connectivity (JDBC)-enabled application, you can use AcuXDBC Enterprise Edition. AcuXDBC is a data management system, designed to integrate ACUCOBOL-GT data files into a relational database-like environment. In addition to database-like features, the enterprise editions of AcuXDBC can give users of JDBC-enabled Java applications seamless access to ACUCOBOL-GT Vision files.

Refer to the *AcuXDBC User's Guide* for more information.

2.3 Calling Java from COBOL

To call Java from your COBOL application, you can:

- Call the C$JAVA library routine. You can use configuration variables to preload the JVM and pass command-line options to it.
- Use the C$SOCKET library routine to facilitate interprocess communication via sockets
- Call the Java Virtual Machine (JVM) DLL or shared library
- Use the C$SYSTEM library routine to send a Java command line to the host machine
- Use named pipes to pass data between your COBOL and Java applications if they reside on the same host machine

2.3.1 Calling the C$JAVA Routine

An easy, effective way to call Java from COBOL is via the C$JAVA library routine. A call to C$JAVA causes the JVM to be loaded (if it is not already) and the specified Java class to be loaded.

The COBOL statement used to make a call to Java from COBOL has the following syntax:

```
CALL "C$JAVA"
     USING OP-CODE, CLASS-NAME, METHOD-NAME, METHOD-SIGNATURE,
          FIELD-INT, FIELD-RETURN
     GIVING STATUS-VAL.
```

For example:

```
CALL "C$JAVA"
     USING CJAVA-NEW, "acuCobolGT/CAcuCobol", "()V"
     GIVING OBJECT-HANDLE.
```

The default CALL “C$JAVA” statement is designed to call a Java method. It requires a class name fully qualified with the package name if necessary. It also requires a method name and a method signature describing the parameter types and return types. (See section 2.3.1.1 for more information on the method signature.) After the method signature, pass the parameters that the method requires, and finally pass a parameter to hold the Java return value from the method. If the method is void, no return parameter is required. A giving value is returned to pass any other error code that may have occurred.

Refer to Appendix I in ACUCOBOL-GT Appendices for complete information on the C$JAVA library routine and its op-codes. Section 2.3.1.8 contains information about configuration variables related to the use of the C$JAVA routine.

Note: To call Java from COBOL, HP-UX users must relink the runtime so that it is statically linked to “libjvm.sl”. For instructions, refer to section section 2.3.1.12.

2.3.1.1 Method signatures

Parameter signatures are used by the JNI functions to get the method ID of a method in a Java class so that it can be called by non-Java programs. Two examples are “(I)I” and “(Z)Z”. The first one describes a Java method taking an int parameter and returning an int value. The second is a Java method taking a boolean parameter and returning a boolean value. For a Java method like this:

```
int MyJavaMethod( boolean param1, int param2, long
param3, double param4)
```

The signature would look like this:

```
(ZIJD)I.
```

The return value comes last after the close parenthesis. “Z” was chosen to represent boolean because “B” is used to describe a byte data value.

Section 2.3.1.2 shows a list of the parameter types supported by ACUCOBOL-GT. An “L” represents some object type. “J” is used for longs. Everything else in Java is an object (strings, arrays, etc.), and the signatures look like this:

Object Types	**Signature**
String	Ljava/lang/String
Object	Ljava/lang/Object
Array of strings	[Ljava/lang/String

Example syntax is shown in the following table:

Signature	**Description**
()V	Java-defined void method taking no parameters
(Z)I	Takes boolean, returns int
(ZISDJ)Z	Takes boolean, int, short, double, long, returns boolean
(B[J[I)X	Takes byte, long array, int array, returns string
(XLjava/lang/Object;)Ljava/lang/String;	Takes string, object, returns string
(C)C	Takes char, returns char

Comments on syntax

The type for Java Strings in a method signature can be either “Ljava/lang/String;” or “X”, but there is no need to specify length since the convert data routine will determine length from the COBOL declaration of the particular variable. So an appropriate method_signature could be “Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;” or “XXX” (both mean the same thing), but “X3X4X5” will be treated as three Strings and the digits will be ignored.

The method_signature for a data item that will be converted to a java char type is “C” not “X” even though the COBOL variable is declared PIC X – the important consideration is how it is declared in Java. If it is declared as a String, use “X”, if it is declared as char, use “C”.

The use of JNI functions for String conversion require the use of C or null terminated strings. If you need 10 characters for your string, then declare the PIC X item with a length of at least 11 and ensure the value for the last position is a low value. If you declare the string as 10 and use all ten positions for character data, the 10th item will be overwritten during conversion.

Finding a method signature

A Java utility that comes with the Java JRE produces all the parameter signatures of a given JAR file or class automatically so that it is not necessary to determine the signature manually. Use this utility, called “javap.exe”, to get the exact signature to use with your CALL “C$JAVA” statement.

Here is the output from running “javap” on “acuUtilities/AcuJavaTest”. The part following the word “Signature” could be cut and pasted into a CALL “C$JAVA” for a given method.

```
D:\cobol7\bin>javap -s acuUtilities/AcuJavaTest
Compiled from "AcuJavaTest.java"
public class acuUtilities.AcuJavaTest extends
java.lang.Object{
public acuUtilities.AcuJavaTest();
  Signature: ()V
public static void main(java.lang.String[])   throws
java.io.IOException;
  Signature: ([Ljava/lang/String;)V
public static int executeCommand(java.lang.String);
```

```
  Signature: (Ljava/lang/String;)I
}
```

2.3.1.2 Supported parameter types

Following is a list of Java parameter types that are supported by ACUCOBOL-GT:

```
V – void
Z - boolean
B - byte
C - char
S - short
I - int
J - long
F - float
D - double
X - string
LString;
Ljava/lang/String;
Ljava/lang/Object;

[Z - boolean array
[B - byte array
[C - char array
[S - short array
[I - int array
[J - long array
[F - float array
[D - double array
[X - string array
[LString; - string array
[Ljava/lang/String; - string array
[Ljava/lang/Object; - object array
```

2.3.1.3 Creating and using Java objects in COBOL

Using the C$JAVA routine, you can create new Java objects in COBOL, call methods on Java objects, and destroy Java objects. The following sections describe how.

Creating a new Java object

Create a new Java object using the CJAVA-NEW op-code to the C$JAVA routine. Be sure to pass a fully qualified package/class name and a constructor signature. Use the GIVING statement to return the object handle. Here is an example of how to create a new Java object:

```
CALL "C$JAVA" USING CJAVA-NEW, "acuCobolGT/CAcuCobol", "()V"
GIVING OBJECT-HANDLE.
```

Calling methods on Java objects

You can call Java methods as static methods, virtual methods, or non-virtual methods by using op-codes 8-10 of the C$JAVA routine, or you can call a Java main method using op-code 29. If you do not use an op-code when you call C$JAVA, the default runtime behavior is to try to call the method statically, and then virtually by trying to create an object using a default constructor. A non-virtual method is called on the specific object that is being used. A virtual method can be called on a method that is inherited from one of the object's superclasses. Here are examples of each of the types of calls:

Default:

```
CALL "C$JAVA" USING "acuCobolGT/CAcuCobol",
"CobolCallingJavaChar", "(C)C", FIELD-CHAR, FIELD-CHARRET
GIVING STATUS-VAL.
```

Virtual:

```
CALL "C$JAVA" USING CJAVA-CALL, OBJECT-HANDLE, "acuCobolGT/
CAcuCobol", "CobolCallingJavaLong", "(J)J", FIELD-LONG,
FIELD-LONGRET GIVING STATUS-VAL.
```

Non-virtual:

```
CALL "C$JAVA" USING CJAVA-CALLNONVIRTUAL, OBJECT-HANDLE,
"acuCobolGT/CAcuCobol", "CobolCallingJavaBoolean", "(Z)Z",
FIELD-BOOL, FIELD-BOOLRET GIVING STATUS-VAL.
```

Static:

```
CALL "C$JAVA" USING CJAVA-CALLSTATIC, "acuCobolGT/CAcuCobol",
"CobolCallingJavaDouble", "(D)D", FIELD-DOUBLE,
FIELD-DOUBLERET GIVING STATUS-VAL.
```

Main:

```
CALL "C$JAVA" USING CJAVA-CALLJAVAMAIN, "CobolCallingJava",
"StrParam1",
    "StrParam2", "StrParam3", "StrParam4" GIVING STATUS-VAL.
```

This example calls a Java main method with the following signature:

```
public static void main( String[] args );
```

Additional examples:

```
CALL "C$JAVA" USING CJAVA-CALLNONVIRTUAL, OBJECT-HANDLE,
"acuCobolGT/CAcuCobol", "CobolCallingJavaVoid", "()V" GIVING
STATUS-VAL.

CALL "C$JAVA" USING CJAVA-CALL, OBJECT-HANDLE, "acuCobolGT/
CAcuCobol", "CobolCallingJavaStringV", "(X)X", FIELD-STRING,
FIELD-STRINGRET GIVING STATUS-VAL.
```

Destroying Java objects

To destroy a Java object, use C$JAVA's CJAVA-DESTROY op-code, and pass a valid object handle:

```
CALL "C$JAVA" USING CJAVA-DESTROY, OBJECT-HANDLE GIVING
STATUS-VAL.
```

2.3.1.4 Creating and using Java arrays in COBOL

You can use the C$JAVA routine to create and pass Java arrays of primitive types, objects, and strings; to get and set array elements; to clear arrays; and to convert COBOL tables to Java arrays and vice versa.

Creating and passing arrays of primitive types

To create Java arrays, use the op-code CJAVA-CREATEARRAY and pass in the type of the array and the size of the array. Return the array handle through the GIVING statement.

In the example below, an array of ints is created, and ARRAY-SIZE is declared USAGE IS SIGNED-INT VALUE 10. An object method that would take this array would have a parameter in its signature of type [I such as "([I)I". The primitives array types are documented in section 2.3.1.1.

```
CALL "C$JAVA" USING CJAVA-CREATEARRAY, CJAVA-INTARRAY,
ARRAY-SIZE GIVING ARRAY-HANDLE.
```

Creating and passing arrays of objects

You can create an object array as shown here:

```
CALL "C$JAVA" USING CJAVA-CREATEARRAY, CJAVA-OBJECTARRAY, 10
GIVING ARRAY-HANDLE.
```

In this case, the array consists of an array of object handles. Here is an example of calling a Java method that takes an array of objects:

```
CALL "C$JAVA" USING CJAVA-CALL, OBJECT-HANDLE, "acuCobolGT/
CAcuCobol", "CobolCallingJavaObjectArray", "([Ljava/lang/
Object;)X", ARRAY-HANDLE, FIELD-STRINGRET GIVING STATUS-VAL.
```

Creating and passing arrays of strings

Even though strings in Java are objects, they are treated separately for the convenience of using them with PIC X tables. Here is an example of creating a string array:

```
CALL "C$JAVA" USING CJAVA-CREATEARRAY, CJAVA-STRINGARRAY, 10
GIVING ARRAY-HANDLE.
```

Here are examples of setting a string array element. In this example, STRING-TABLE is declared PIC X(20) OCCURS 10.

```
MOVE "99999999999999999999" TO STRING-TABLE(10)

CALL "C$JAVA" USING CJAVA-SETARRAYELEMENT, ARRAY-HANDLE, 1,
STRING-TABLE(10), GIVING STATUS-VAL.
```

This example demonstrates how to call a Java method that takes an array of strings as a parameter:

```
CALL "C$JAVA" USING CJAVA-CALL, OBJECT-HANDLE, "acuCobolGT/
CAcuCobol", "CobolCallingJavaStringArray", "([Ljava/lang/
String;)X", ARRAY-HANDLE, FIELD-STRINGRET GIVING STATUS-VAL.
```

Getting and setting array elements

You set array elements using the CJAVA-SETARRAYELEMENT op-code and passing in an array handle, the position in the array to set, and the value to set. In the following example, the first element of an array is set with the first value from an integer table that is USAGE IS SIGNED-INT OCCURS 10.

```
CALL "C$JAVA" USING CJAVA-SETARRAYELEMENT, ARRAY-HANDLE, 1,
INT-TABLE(1), GIVING STATUS-VAL.
```

Getting array elements is done using a similar syntax with the op-code CJAVA-GETARRAYELEMENT. This call requires an array handle, the position in the array to get, and the variable into which the array value will be placed. Here is an example:

```
CALL "C$JAVA" USING CJAVA-GETARRAYELEMENT, ARRAY-HANDLE, 5,
INT-TABLE(1), GIVING STATUS-VAL.
```

In this case, we are getting element 5 from the array and placing it in the first element of an integer table.

Getting and setting array regions

You set array regions using the CJAVA-SETARRAYREGION op-code. This op-code takes a Java array object copies the elements from a COBOL table data item into a specified range. Getting array regions is done using a similar op-code, CJAVA-GETARRAYREGION. This op-code takes a Java array object, gets the specified range of elements, and copies them into a COBOL table data item.

Clearing arrays

Clearing arrays is straightforward. Use the op-code CJAVA-CLEARARRAY and pass in the array handle of the array to be cleared, as shown:

```
CALL "C$JAVA" USING CJAVA-CLEARARRAY, ARRAY-HANDLE GIVING
STATUS-VAL.
```

Implicit COBOL table/Java array conversion

With ACUCOBOL-GT, it is possible to pass a COBOL table directly to a method that requires a Java array. The contents of the table are automatically converted to an array of the type the Java method expects. When the method completes, the contents of the table are updated with what is in the array. You do not have to explicitly convert the COBOL table to a Java array and convert it back again. No special op-code is required to do the conversion. When the runtime sees the array type in the signature, it tries to convert that table parameter to an array. Here is an example of a table being passed to a Java method that takes an array parameter:

```
CALL "C$JAVA" USING CJAVA-CALL, OBJECT-HANDLE, "acuCobolGT/
CAcuCobol", "CobolCallingJavaIntArray", "([I)I", INT-GROUP,
FIELD-RET GIVING STATUS-VAL.
```

In the above example, INT-GROUP is declared:

```
01 INT-GROUP.
    03  INT-DATA occurs 10 times.
        05  INT-ELEMENT signed-int.
```

The values for INT-GROUP are set as follows:

```
   MOVE 1111 to INT-ELEMENT(1)
   MOVE 2222 to INT-ELEMENT(2)
   MOVE 3333 to INT-ELEMENT(3)
   MOVE 4444 to INT-ELEMENT(4)
   MOVE 5555 to INT-ELEMENT(5)
```

It should be noted that the type of the table passed into the Java method should be the appropriate type, that is, data of the same element size (in bits). The size of the Java array will be the number of elements in the table.

Explicit COBOL table/Java array conversion

With ACUCOBOL-GT, you can also use C$JAVA op-codes to explicitly convert Java arrays to COBOL and COBOL tables to Java. This functionality gives you more precise control over the conversion process.

The op-code to convert a Java array to a table is JAVA-CONVERTARRAYTOTABLE. Here is an example of an array of Java ints being converted to a USAGE SIGNED-INT OCCURS 10 COBOL table:

```
CALL "C$JAVA" USING CJAVA-CONVERTARRAYTOTABLE, ARRAY_HANDLE,
10, 0, INT-TABLE(1) GIVING STATUS-VAL.
```

The call takes the array handle, the number of elements to convert, the starting element position in the array, and the COBOL table variable in which to place the converted array.

To explicitly convert a COBOL table to Java, you can use the C$JAVA op-code CJAVA-CONVERTTABLETOARRAY. Here is an example of a call that converts a table to an array:

```
CALL "C$JAVA" USING CJAVA-CONVERTTABLETOARRAY, INT-TABLE(1),
10, 0, ARRAY-HANDLE, GIVING STATUS-VAL.
```

In this case, the call requires the COBOL table from which the values are taken, the number of elements, the position of the first element, and the handle of the destination array.

2.3.1.5 Using Java logging from COBOL

With the C$JAVA routine, you can also log Java messages and configure the Java log.

Logging messages

If you want to log Java messages from a COBOL program, use the CJAVA-LOGMESSAGE op-code as follows:

```
CALL "C$JAVA" USING CJAVA-LOGMESSAGE, "Message to log".
```

The advantage of using the Java log is that it is thread-safe, and all of the messages from a given thread of execution are written to the same log whether that thread is executing COBOL or Java. Also, logs in Java are highly configurable. Note that the sample log output shown below is formatted to report date, time, class, method, and log level before the message.

```
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO --> COBOL LOG --> Entered
TestJavaToCobol
```

```
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO --> COBOL LOG --> Exiting
TestJavaToCobol
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO --> call.error = 0,
exit.code = 0, signal.number = 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO --> exit message =
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Call error: 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Exit code: 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Signal number: 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Exit message:
11/30/04 2:13:57 PM acuCobolGT.CAcuCobol cblLog INFO --> CobolCallingJavaTest: Complete
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO -->
CobolCallingJavaReentrantTest ()V
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO --> call.error = 0,
exit.code = 0, signal.number = 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_cobol INFO --> exit message =
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Call error: 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Exit code: 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Signal number: 0
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM cblLog INFO --> Exit message:
11/30/04 2:13:57 PM com.acucorp.acucobolgt.CVM acu_shutdown INFO --> shutdown called -
shutdown param: 0
11/30/04 2:13:57 PM acuUtilities.AcuJavaTest main INFO --> shutdown complete
11/30/04 2:13:57 PM acuUtilities.AcuJavaTest main INFO --> calling cobol end
```

Configuring the Java log

To configure the Java log created with the CJAVA-LOGMESSAGE op-code, modify the “logging.properties” file that is located in the runtime directory. The output location of the log (console or file) can be specified, as well as the log level, for example, INFO or SEVERE. Below is a sample of the “logging.properties” file:

```
# setting to limit messages printed to the console.
.level= INFO
#.level= FINEST

############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################

# default file output is in user's home directory.
#java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.pattern = CVM.log
java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 1
```

```
java.util.logging.FileHandler.append = false
java.util.logging.FileHandler.level = INFO
#java.util.logging.FileHandler.formatter =
java.util.logging.XMLFormatter
#java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter
java.util.logging.FileHandler.formatter =
com.acucorp.acucobolgt.logFormat

# Limit the message that are printed on the console to INFO and
above.
#java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.level = INFO
#java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter
java.util.logging.ConsoleHandler.formatter =
com.acucorp.acucobolgt.logFormat

############################################################
# Facility specific properties.
# Provides extra control for each logger.
############################################################

# For example, set the com.acucorp.acucobolgt.CVM logger to
only log SEVERE
# messages:
#com.acucorp.acucobolgt.CVM.level = SEVERE
com.acucorp.acucobolgt.CVM.level = INFO
#acuCobolGT.CAcuCobol.level = SEVERE
acuCobolGT.CAcuCobol.level = INFO
#acuUtilities.AcuJavaTest.level = SEVERE
acuUtilities.AcuJavaTest.level = INFO
```

Alternate logging

Rather than using the CJAVA-LOGMESSAGE op-code, COBOL developers could use the C$JAVA routine to call **log4j** from COBOL. **log4j** is an open source tool developed for putting log statements into a Java application. It provides a robust, reliable and easy to implement framework for logging Java applications for debugging and monitoring purposes.

This form of logging should only be considered if you are running a Java application server and AcuConnect on the same machine and if the COBOL program already uses C$JAVA routine.

2.3.1.6 Creating and using a JDBC ResultSet

With ACUCOBOL-GT, there are two ways to create and use a Java Database Connectivity (JDBC) ResultSet in COBOL:

- Using a class called DBConnect that is included in ACUCOBOL-GT's COBOL Virtual Machine Java archive file, "CVM.jar". (See section 2.2.3.1 for more information on ACUCOBOL-GT's CVM class.)
- Using the CJAVA-DBCONNECT and CJAVA-DBQUERY op-codes to the C$JAVA routine

Using DBConnect class to get a ResultSet object

ACUCOBOL-GT's "CVM.jar" package contains a class for connecting to JDBC data sources, and querying those data sources for ResultSet objects. The class is called DBConnect.

The DBConnect class has two public static methods called "connect" and "query". The "connect" method takes two string parameters: a JDBC driver string, and a JDBC connection string, and returns a java.sql.Connection object. The "query" method takes two parameters: a query string, and a java.sql.Connection object, and returns a ResultSet object. The ResultSet object can then be used to access and update the data.

Here is an example of using the DBConnect class in COBOL:

```
MOVE "sun.jdbc.odbc.JdbcOdbcDriver" to DB-DRIVERSTR.

MOVE "jdbc:odbc:DefaultDir=D:\cobol7\bin;Driver={Microsoft
Text Driver (*.txt; *.csv)};DriverId=27;UID=admin;Initial
Catalog=D:\cobol7\bin" to DB-CONNECTSTR.

MOVE "SELECT * FROM dataFile.csv" to DB-QUERY.

CALL "C$JAVA" USING CJAVA-NEW, "java/lang/Object", "()V"
GIVING DB-CONNECT.
```

```
CALL "C$JAVA" USING CJAVA-NEW, "java/lang/Object", "()V"
GIVING DB-RESULTSET.

CALL "C$JAVA" USING "com/acucorp/acucobolgt/DBConnect",
"connect", "(XX)Ljava/sql/Connection;", DB-DRIVERSTR,
DB-CONNECTSTR, DB-CONNECT GIVING STATUS-VAL.

CALL "C$JAVA" USING "com/acucorp/acucobolgt/DBConnect",
"query", "(XLjava/sql/Connection;)Ljava/sql/ResultSet;",
DB-QUERY, DB-CONNECT, DB-RESULTSET GIVING STATUS-VAL

CALL "C$JAVA" USING CJAVA-CALL, DB-RESULTSET, "java/sql/
ResultSet", "next", "()Z", FIELD-BOOLRET GIVING STATUS-VAL.

CALL "C$JAVA" USING CJAVA-CALL, DB-RESULTSET, "java/sql/
ResultSet", "getRow", "()I", FIELD-RET GIVING STATUS-VAL.

MOVE 1 to FIELD-INT.

CALL "C$JAVA" USING CJAVA-CALL, DB-RESULTSET, "java/sql/
ResultSet", "getString", "(I)X", FIELD-INT, FIELD-STRINGRET
GIVING STATUS-VAL.

CALL "C$JAVA" USING CJAVA-DELETE, DB-CONNECT GIVING
STATUS-VAL.

CALL "C$JAVA" USING CJAVA-DELETE, DB-RESULTSET GIVING
STATUS-VAL.
```

In this example, the JDBC driver used was the jdbc:odbc bridge that ships with the Java SDK. The ODBC driver is the Microsoft Text driver, and a text CSV file was used as the data source. Two new object handles are created to contain the Connection and ResultSet handles. The ResultSet method “next” is called to move to the first row, the “getRow” method is called to find the current row number, and the “getString” method is called to get the value in column one which happens to be of type string. Finally, the two objects are deleted.

Using op-codes to get a ResultSet object

Another way to access JDBC ResultSet objects is to issue a call to the C$JAVA routine using the op-codes CJAVA-DBCONNECT and CJAVA-DBQUERY. To do this, you must include the "java.def" file that comes with ACUCOBOL-GT in the COBOL program's working storage section. "java.def" is located in the sample/def directory where you installed ACUCOBOL-GT.

This method is somewhat more efficient than using the CVM's DBConnect class, because the Connection and ResultSet handles do not have to be created prior to being used. Here is an example of using the op-codes:

```
MOVE "sun.jdbc.odbc.JdbcOdbcDriver" to DB-DRIVERSTR

MOVE "jdbc:odbc:DefaultDir=D:\\cobol7\\bin;Driver={Microsoft
Text Driver (*.txt; *.csv)};DriverId=27;UID=admin;Initial
Catalog=D:\\cobol7\\bin" to DB-CONNECTSTR

MOVE "SELECT * FROM dataFile.csv" to DB-QUERY.

CALL "C$JAVA" USING CJAVA-DBCONNECT, DB-DRIVERSTR,
DB-CONNECTSTR GIVING DB-CONNECT.

CALL "C$JAVA" USING CJAVA-DBQUERY DB-QUERY, DB-CONNECT GIVING
DB-RESULTSET.

CALL "C$JAVA" USING CJAVA-CALL, DB-RESULTSET, "java/sql/
ResultSet", "next", "()Z", FIELD-BOOLRET GIVING STATUS-VAL.

CALL "C$JAVA" USING CJAVA-CALL, DB-RESULTSET, "java/sql/
ResultSet", "getRow", "()I", FIELD-RET GIVING STATUS-VAL.

MOVE 1 to FIELD-INT.

CALL "C$JAVA" USING CJAVA-CALL, DB-RESULTSET, "java/sql/
ResultSet", "getString", "(I)X", FIELD-INT, FIELD-STRINGRET
GIVING STATUS-VAL.

CALL "C$JAVA" USING CJAVA-DELETE, DB-CONNECT GIVING
STATUS-VAL.

CALL "C$JAVA" USING CJAVA-DELETE, DB-RESULTSET GIVING
STATUS-VAL.
```

2.3.1.7 Java Remote Method Invocation (RMI) interoperability

The following sections describe how to use the ACUCOBOL-GT runtime class, CVM, as a RMI client and RMI server. They also describe how to use the C$JAVA routine to connect to an RMI server.

Using the runtime as a Java RMI Client

ACUCOBOL-GT's "CVM.jar" package contains a class for connecting to an RMI server and returning an object through which remote methods can then be called on the RMI server. The class is called RemoteConnect. (See section 2.2.3.1 for more information on the CVM class.)

The RemoteConnect class has a public static method called "CreateRemoteObject" that takes two strings and an int as parameters and returns a java.rmi.Remote object handle. The strings it requires are the host name of the server, the name of the RMI server object, and the int is the port number on which the server is listening. Once the remote object handle has been returned, remote methods on that object can be called in the same way as methods as any other Java object handle. Here is an example of using the RemoteConnect class in COBOL:

```
CALL "C$JAVA" USING CJAVA-NEW, "java/lang/Object", "()V"
GIVING REMOTE-OBJ.

CALL "C$JAVA" USING "com/acucorp/acucobolgt/RemoteConnect",
"CreateRemoteObject", "(XXI)Ljava/rmi/Remote;", "localhost",
"TestRemoteInterface", PORT-NUMBER, REMOTE-OBJ GIVING
STATUS-VAL.

CALL "C$JAVA" USING CJAVA-CALL, REMOTE-OBJ, "acuUtilities/
TestRemoteInterface", "TestRemoteMethod", "()X",
FIELD-STRINGRET GIVING STATUS-VAL.

CALL "C$JAVA" USING CJAVA-DELETE, REMOTE-OBJ GIVING
STATUS-VAL.
```

In this example, an instance of the object is created, and then the method "CreateRemoteObject" in RemoteConnect is called. In this case, the host is localhost, and the name of the remote interface is "TestRemoteInterface". "TestRemoteInterface" extends Remote and has one remote method called "TestRemoteMethod". The port number is "0" here. Passing in "0" causes

the method to look for the object on the default RMI port, 1099. Note that for this to work, the RMI registry needs to have been previously started using the command “start rmiregistry” from the command line, and then the server object must be registered with the RMI registry.

Using an op-code to connect to an RMI Server

It is also possible to connect to an RMI server by calling the C$JAVA routine with op-code CJAVA-NEWREMOTEOBJECT. It is very much like the CJAVA-NEW op-code, but instead of creating an object in the local JVM, it creates an instance of a remote object. This op-code takes three parameters: the host name, the server name, and the port number. Here is an example of the call:

```
CALL "C$JAVA" USING CJAVA-NEWREMOTEOBJECT, "localhost",
"TestRemoteInterface", PORT-NUMBER GIVING REMOTE-OBJ.

CALL "C$JAVA" USING CJAVA-CALL, REMOTE-OBJ, "acuUtilities/
TestRemoteInterface", "TestRemoteMethod", "()X",
FIELD-STRINGRET GIVING STATUS-VAL.

CALL "C$JAVA" USING CJAVA-DELETE, REMOTE-OBJ GIVING
STATUS-VAL.
```

Using the runtime as a Java RMI Server

To create and use a Java RMI server object, you must first create a class for the object. This takes two steps. First, you must create an interface that extends the Java interface “java.rmi.Remote”. Not only does the interface need to extend Remote, all the methods that can be called remotely must throw the RemoteException. Second, you must write a class that implements the interface. Once this is done, you can create an RMI server object and register it for use with the Java RMI Registry.

Here is a very simple illustration of this concept. First, an interface must be created in Java:

```
public interface TestRemoteInterface extends Remote {
    String TestRemoteMethod() throws RemoteException;
}
```

Next, a class that implements the TestRemoteInterface must be written. In the case of the methods of this class, you do not need to throw RemoteException, because the interface method declarations have that. Here is an example of such a class in Java:

```
public class TestRMIServer implements TestRemoteInterface {

    public TestRMIServer() {}

    public String TestRemoteMethod() {
return "TestRemoteMethod successfully called.";
    }
}
```

Once this has been written, compiled, and packaged, you can register the server with the RMI registry. First, you must start the RMI registry on the host that will be the RMI server. You can do this using the command “start rmiregistry”. Next, you can use a COBOL program to start the RMI server. In this COBOL program, make a call to the C$JAVA routine with the op-code CJAVA-STARTREMOTESERVER.

Here is an example of the code required to start an RMI server which uses the interface and server classes shown above:

```
CALL "C$JAVA" USING CJAVA-NEW, "acuUtilities/TestRMIServer",
"()V" GIVING REMOTE-SERVER.
CALL "C$JAVA" USING CJAVA-STARTREMOTESERVER, REMOTE-SERVER,
"TestRemoteInterface", PORT-NUMBER GIVING STATUS-VAL.
```

The first step is to create the server object. The second step is to start the server. CJAVA-STARTREMOTESERVER takes the remote server object handle just created, the name of the remote server to register, and the port number on which the server will listen. Once the server has started, that instance of the runtime that started it will block and remain running, listening for requests for the server from RMI clients.

2.3.1.8 Handling Java exceptions

The C$JAVA library routine includes two op-codes for handling exceptions that are thrown by Java: CJAVA-EXCEPTIONOCCURRED and CJAVA-GETEXCEPTIONOBJECT.

CJAVA-EXCEPTIONOCCURRED is returned by any call to C$JAVA that returns a status value, but during which an exception was thrown.

CJAVA-GETEXCEPTIONOBJECT returns the exception object of the last exception thrown. Once the exception object is returned, you can call any of the methods on the exception object that are documented in the Java documentation.

In addition, exception information is now written to stderr or the error file specified by the "-le" command line option when a Java exception occurs. This information is formatted like a normal Java stack trace.

2.3.1.9 Releasing memory

The Java Virtual Machine (JVM) doesn't have an explicit method to allocate and free memory. For this reason, if the COBOL program gets a reference to an object from the JVM, it is the COBOL program's responsibility to release the reference, otherwise the JVM might exhaust machine resources.

To aid in this effort, we suggest that you establish a JVM trace file using the following configuration variables:

- A_JAVA_TRACE_FILENAME - Names the file where you want to store trace information
- A_JAVA_TRACE_VALUE - Specifies the types of calls to trace

Using the JVM trace file, you can determine when a call to the JVM returns an object reference that must be released, then you can call C$JAVA with CJAVA_DESTROY or CJAVA_DELETE to remove the reference.

For there to be no memory leaks, any call that returns a reference to a Java object needs to be paired with a call to release that reference. When the runtime gets an object reference from the JVM, it is the runtime's responsibility to release the reference. When the runtime calls the JVM, it deletes the local reference to any memory the JVM allocated on behalf of the runtime.

In the JVM, an entity known as the JVM garbage collector also de-allocates memory that is no longer being used. To specify how often the runtime should call the JVM garbage collector, use the A_JAVA_GC_COUNT configuration variable.

Refer to Appendix H of the *ACUCOBOL-GT Appendices* for more information on these and other configuration variables.

2.3.1.10 C$JAVA configuration variables

ACUCOBOL-GT includes several configuration variables for calling Java via the C$JAVA routine.

Configuration Variable	**Purpose**
PRELOAD_JAVA_LIBRARY	Tells the runtime to preload the JVM on startup
JAVA_LIBRARY_NAME	Specifies the name and path of the JVM library to load
JAVA_OPTIONS	Specifies Java command-line options
A_JAVA_TRACE_FILENAME	Names the file where you want to store JVM call trace information
A_JAVA_TRACE_VALUE	Specifies the types of calls to trace
A_JAVA_GC_COUNT	Specifies how often the runtime should call the JVM garbage collector
A_JAVA_CHARSET	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”.

All of these variables are optional. If desired, you include them in your runtime configuration file, just as you would any ACUCOBOL-GT configuration variable. See Appendix H in the *ACUCOBOL-GT Appendices* for details on using these variables.

For Java interoperability, your configuration file may look like this:

```
PRELOAD_JAVA_LIBRARY=1
JAVA_LIBRARY_NAME=jvm.dll (libjvm.so unix)
JAVA_OPTIONS="-Djava.library.path="c:\usr\lib" -Xms128m
-Xmx128m -classpath /java/MyClasses/myclasses.jar"
```

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).

You can set these properties using the JAVA_OPTIONS variable. This is the first place that the runtime looks when trying to call Java from COBOL. If you prefer, you can set them in the environment:

- To set the Java system property, java.library.path, set LD_LIBRARY_PATH in the environment.
- To set the java.class.path property, set a CLASSPATH configuration or environment variable.

2.3.1.11 Configuration and deployment

To call Java from COBOL using the C$JAVA routine, perform the following steps:

1. Install and correctly configure an ACUCOBOL-GT runtime. Optionally, install AcuBench if the system will also be used for COBOL development.
2. Install and correctly configure a JRE version 1.4.2 or later. Optionally, install a J2SE SDK if the same system will also be used for Java development.
3. Configure the ACUCOBOL-GT runtime as appropriate by creating a runtime configuration file.

   By default, the JVM is loaded by the runtime the first time it executes a CALL "C$JAVA" statement. If you want to preload the JVM, use the PRELOAD_JAVA_LIBRARY variable. Use the JAVA_LIBRARY_NAME variable if you want to specify the DLL that

exports the JNI API for loading the JVM. If desired, specify Java command-line options using the JAVA_OPTIONS variable. You can specify additional libraries and classpaths here as well.

4. Issue a CALL "C$JAVA" statement to call the C$JAVA library routine.

2.3.1.12 Linking the runtime to "libjvm.sl" on HP-UX

To call C$JAVA on HP-UX platforms, you must first relink the runtime so that it is statically linked to the "libjvm.sl" shared library. Here are instructions:

1. Edit "Makefile", located in the ACUCOBOL-GT /lib directory. Modify the CC and EXTRA_LDFLAG entries as shown for your port below:

   32-bit Static:

   ```
   Original: CC = cc -Ae +DAportable -Wl,+s
   -D_LARGEFILE64_SOURCE
   Modified: CC = cc -mt -Ae +DAportable -Wl,+s
   -D_LARGEFILE64_SOURCE
   Original: EXTRA_LDFLAG =
   Modified: EXTRA_LDFLAG = -L /opt/java1.5/jre/lib/
   PA_RISC2.0/server -ljvm
   ```

   32-bit Shared:

   ```
   Original: CC = cc -Ae +DAportable -Wl,+s
   -D_LARGEFILE64_SOURCE +z
   Modified: CC = cc -mt -Ae +DAportable -Wl,+s
   -D_LARGEFILE64_SOURCE +z
   Original: EXTRA_LDFLAG = -Wl,+b -Wl,$(ACUVERSPATH)/
   lib:$(ACUPATH)/lib:.:/usr/lib:/lib
   Modified: EXTRA_LDFLAG = -Wl,+b -Wl,$(ACUVERSPATH)/
   lib:$(ACUPATH)/lib:.:/usr/lib:/lib \   -L /opt/java1.5/
   jre/lib/PA_RISC2.0/server -ljvm
   ```

   64-bit Static:

   ```
   Original: CC = cc -Ae +DS2.0 +DA2.0W +DD64 -Wl,+s
   Modified: CC = cc -mt -Ae +DS2.0 +DA2.0W +DD64 -Wl,+s
   Original: EXTRA_LDFLAG =
   ```

```
Modified: EXTRA_LDFLAG = -L /opt/java1.5/jre/lib/
PA_RISC2.0W/server -ljvm
```

64-bit Shared:

```
Original: CC = cc -Ae +DS2.0 +DA2.0W +DD64 -Wl,+s +z
Modified: CC = cc -mt -Ae +DS2.0 +DA2.0W +DD64 -Wl,+s +z
Original: EXTRA_LDFLAG = -Wl,+b -Wl,$(ACUVERSPATH)/
lib:$(ACUPATH)/lib:.:/usr/lib:/lib
Modified: EXTRA_LDFLAG = -Wl,+b -Wl,$(ACUVERSPATH)/
lib:$(ACUPATH)/lib:.:/usr/lib:/lib \  -L /opt/java1.5/
jre/lib/PA_RISC2.0W/server -ljvm
```

2. Rebuild the runtime. In the ACUCOBOL-GT /lib directory execute "make runcbl".
3. Copy the new "runcbl" to the ACUCOBOL-GT /bin directory.
4. Set the following in the runtime configuration file:

   32-bit:

   ```
   JAVA_LIBRARY_NAME /opt/java1.5/jre/lib/PA_RISC2.0/
   server/libjvm.sl
   ```

   64-bit:

   ```
   JAVA_LIBRARY_NAME /opt/java1.5/jre/lib/PA_RISC2.0W/
   server/libjvm.sl
   ```

   All:

   ```
   JAVA_OPTIONS -XX:+UseAltSigs
   ```

5. Set the following in the environment:

   32-bit:

   ```
   SHLIB_PATH=/opt/java1.5/jre/lib/PA_RISC2.0/server:/
   opt/java1.5/jre/lib/PA_RISC2.0;$SHLIB_PATH
   ```

   64-bit:

   ```
   LD_LIBRARY_PATH=/opt/java1.5/jre/lib/PA_RISC2.0W/
   server:/opt/java1.5/jre/lib/
   PA_RISC2.0W;$LD_LIBRARY_PATH
   ```

2.3.1.13 Example

The following sample code is a simple COBOL program calling a Java program. It shows the minimum necessary pieces to call a Java method from a COBOL program.

```
identification division.
program-id.  CobolToJava.
data division.
working-storage section.

01 CLASS-NAME PIC X(80).
01 METHOD-NAME PIC X(80).
01 METHOD-SIGNATURE PIC X(80).
01 STATUS-VAL PIC S9(02) VALUE ZERO.
01 FIELD-INTUSAGE IS SIGNED-INT.
01 FIELD-RETURNUSAGE IS SIGNED-INT.

procedure division.
main-logic.

move "com.acucobolgt.CVM" TO CLASS-NAME
move "CobolCallingJavaInt" TO METHOD-NAME
move "(I)I" TO METHOD-SIGNATURE
move 0 to FIELD-INT
move -1 to FIELD-RETURN

CALL "C$JAVA" USING CLASS-NAME, METHOD-NAME,
METHOD-SIGNATURE, FIELD-INT, FIELD-RETURN GIVING
STATUS-VAL.
```

2.3.1.14 Running the Java interoperability sample programs

Your ACUCOBOL-GT distribution includes Java interoperability sample programs. You will find them in the \acugt\sample\java\ directory where ACUCOBOL-GT is installed. For instructions on setting up and running the samples, refer to section 2.2.3.7.

2.3.2 Using C$SOCKET

You can also use the C$SOCKET library routine to facilitate interprocess communication between Java and COBOL programs via sockets. C$SOCKET is a low-level option, but it is very flexible.

When calling Java from COBOL:

1. The Java programmer creates a server socket and waits for and accepts a connection to the COBOL client.
2. The COBOL programmer uses the C$SOCKET routine to create a client socket (op-code 1), connect via TCP/IP to the port of the Java program, and write data to the socket (op-code 5).
3. The Java program reads the data, processes it, returns data to the socket.
4. The COBOL program uses C$SOCKET to read the data (op-code 6).

Of course, because the data format is totally open and undefined, the COBOL and Java programmers must agree on a common format.

The following sample code demonstrates this capability:

```
*Create a Client Socket.
CALL "C$SOCKET" USING AGS-CREATE-CLIENT, 8765, SERVER-NAME
GIVING SOCKET-HANDLE.

*Write data to socket.
CALL "C$SOCKET" USING AGS-WRITE, SOCKET-HANDLE,
DATA-FROM-CLIENT, DATA-LENGTH.

*Read the return data from the socket.
CALL "C$SOCKET" USING AGS-READ, SOCKET-HANDLE,
DATA-FROM-CLIENT, DATA-LENGTH.

*Close the socket.
CALL "C$SOCKET" USING AGS-CLOSE, SOCKET-HANDLE.
```

Refer to Appendix I in *ACUCOBOL-GT Appendices* for complete information on the C$SOCKET library routine.

2.3.3 Calling the Java Virtual Machine (JVM) DLL or Shared Library

If desired, you can invoke a Java program from COBOL by calling the JVM shared library or DLL and then invoking the routines in that library. In the case of JRE 1.4.2_04, these files are called “jvm.dll” and “libjvm.so”. You can call these files in one of two ways:

1. Load the JVM shared library or DLL into your COBOL program using the CALL statement, then make calls to the JVM functions. For example, in UNIX:

   ```
   CALL "libjvm.so"
   CALL "JVMfunction"
   ```

   (Refer to Chapters 3 and 4 for more information on calling DLLs and shared libraries and their functions.)

   In Windows, you can CALL the DLL to load it, as shown here:

   ```
   CALL "JVM.dll"
   ```

   Then you can call any of the routines contained in the DLL using the direct C interface. This is described in Chapter 6 of this book.

2. Call the JVM through C. ACUCOBOL-GT can make calls to C, and C can make calls to the JVM. Refer to Chapter 6 of this book for information on how to call C subroutines from ACUCOBOL-GT.

2.3.4 Using C$SYSTEM

Another way to invoke a Java program from COBOL is to send a Java command line to the host operating system. You can do this using ACUCOBOL-GT’s C$SYSTEM library routine. This routine combines the functionality of the “SYSTEM” and “C$RUN” routines.

To call a Java program from COBOL via C$SYSTEM, you:

1. Call the C$SYSTEM library routine as described in Appendix I in *ACUCOBOL-GT Appendices*.

2. Send a Java command line to C$SYSTEM to invoke the Java program. For example, to call the Java program AcuOrders, you might use this code:

```
move "javaw -classpath cvm.jar;./ acuprod.AcuOrder"
to the-run-command.
call "C$SYSTEM" using the-run-command.
```

   The C$SYSTEM routine submits the command line to the host operating system as if it were a command keyed in from the terminal.

   Note that you can call C$RUN instead of C$SYSTEM to run the Java program asynchronously, as in:

```
call "C$RUN" using the-run-command.
```

3. Pass data from the Java program to the COBOL program through a disk file or communicate through a database.

Refer to Appendix I in ACUCOBOL-GT Appendices for complete information on the C$SYSTEM and C$RUN library routines.

2.3.5 Using Named Pipes

If your COBOL and Java applications reside on the same host machine, you can call Java from a COBOL program through a named pipe. Named pipes are a method for exchanging information between two unrelated processes. Refer to section 2.2.7 for details.

2.4 Mapping Java Data Types

When interfacing with external Java systems, it is often necessary to map COBOL data types to Java data types. Following are the Java primitive types:

```
boolean - 8 bits, unsigned
byte - 8 bits, signed
char - 16 bits, unsigned
short - 16 bits, signed
int - 32 bits, signed
```

```
float - 32 bits, signed, floating point
long - 64 bits, signed
double - 64 bits, signed, floating point
```

All other types are objects that are composed of other objects or primitive types. For example, the String type is an object. Note that encoding of native Java strings and numbers in memory may be different from way you represent data in COBOL.

Even on 32-bit platforms, Java represents longs as 64 bits. On 32-bit platforms, the ACUCOBOL-GT runtime truncates longs to 32 bits regardless of whether the "-Dw32" or "-Dw64" flags are used for compilation. In order to effectively interoperate using Java longs and use all 64 bits on a 32-bit platform, you must use the PIC S9(18) COMP-5 declaration shown below. Also, in order to use the entire range available to Java shorts and ints (short 32767 to -32768 int 2147483647 to -2147483648) with USAGE IS SIGNED-INT and USAGE IS SIGNED-SHORT declarations, the "-Dw32" flag must be specified at compile time.

The following sample declarations have been used to test COBOL/Java interoperability.

```
01 FIELD-INT USAGE IS SIGNED-INT.
01 FIELD-BOOL pic 9.
01 FIELD-BYTE pic x.
01 FIELD-CHAR pic x.
01 FIELD-SHORT USAGE IS SIGNED-SHORT.
01 FIELD-LONG PIC S9(18) COMP-5.
01 FIELD-FLOAT USAGE IS FLOAT.
01 FIELD-DOUBLE USAGE IS DOUBLE.
01 FIELD-STRING PIC X(80).
```

Another method of declaring ints and shorts is shown below. With these two declarations, the use of the "--TruncANSI" compiler switch is required so that the range checking is correct for the range the native platform allows. (Refer to Chapter 2 in *ACUCOBOL-GT User's Guide* for information about the "--TruncANSI" option.)

```
01 FIELD-INT PIC S9(9) COMP-5.
01 FIELD-SHORT PIC S9(5) COMP-5.
```

Currently, ACUCOBOL-GT converts Unicode UTF-16 Java strings to UTF-8 for representation in PIC X variables. If your program uses code points that require more than 16 bits to represent supplementary characters or if it uses UTF-32, then you should use arrays of Java ints to represent the data.

If your Java strings or PIC X data items contain characters outside of the ISO-8859-1 range, you need to instruct the runtime which character set to use by specifying it in the A_JAVA_CHARSET runtime configuration variable. The default setting is "IS0-8859-1". Be aware of a common misconception that ISO-8859-1 is equivalent to Windows-1252. This is for the most 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.

Note that Java implementations represent data in big endian format regardless of platform. For considerations on moving data between big endian and little endian hosts, refer to the documentation for C$SOCKET in Appendix I in *ACUCOBOL-GT Appendices*, and for the Usage Clause in section 5.7.1.8 of *ACUCOBOL-GT Reference Manual*.

Note: Be careful when sending numeric data across the network via sockets, because some machines use different byte ordering than others, and native numeric data can appear swapped on different machines. COMP-4 data is in the order that most network servers expect binary data to be in, so if you are communicating with a non-COBOL client or server, you should use COMP-4 data of the correct size for the machine in question. If your client and server are both COBOL, you can use standard COBOL types.

2.5 J2EE Application Servers

In recent years, growing demand for infrastructure that can manage hundreds or thousands of applications, integrate them, perform load balancing, provide messaging services, and so on, has led to a fiercely competitive application server market.

One type of application server software revolves around Sun Microsystems' Java 2 Platform, Enterprise Edition (J2EE). BEA WebLogic Server and IBM WebSphere are the two leading application server products for the development of enterprise Java applications. Other vendors such as Sun, Oracle, SAP, and Sybase also compete in this area.

The second category is the Microsoft Platform, which relies on Windows, .NET, COM+, and other technologies. While Microsoft does not have an application server per se, it provides features in its Windows Server editions that compete with the J2EE application servers.

We provide interoperability with applications written for both the J2EE and Microsoft application server platforms. This section describes integration with J2EE application server platforms. Chapter 5 describes how to integrate COBOL with .NET.

2.5.1 Working with J2EE Application Server Products

ACUCOBOL-GT applications can be integrated with application server platforms using any of the methods described in this chapter. Your Java components are managed by the application server, but they can still gain access to COBOL using our CVM class, the C$SOCKET routine, CGI programs, the runtime DLL, or named pipes.

Similarly, your COBOL applications can access your Java components, managed by the application server, via our C$JAVA routine, the C$SOCKET routine, the CALL statement, the C$SYSTEM routine, or named pipes.

2.6 Web Services

Web services are self-contained, modular applications that can be published, located, and invoked across the Web from any location, allowing you to transcend hardware and operating system boundaries. An implementation of SOA, Web services use a standard interface technology to make the services available.

Web service applications perform discrete functions ranging anywhere from simple requests to complicated business processes that combine information from multiple sources. Web services can be developed and componentized internally, or brought in and reused from the outside. To create a robust business solution, you assemble Web services like building blocks into a cohesive entity, mixing and matching software components as you need.

Conceptually, Web services are similar to any application services. What makes them unique is that they describe themselves to the outside world, revealing what functions they perform, how they can be accessed, and what kinds of data they require. Today, this is accomplished with the following standards:

- Self-description is accomplished via Web Service Description Language (WSDL). Using WSDL, you provide information on all publicly available functions, address information for locating the service, and so on.

- Services are published and found in the white, yellow, and green pages of a business registry conforming to the Universal Description, Discovery, and Integration (UDDI) specification.

- All communications to a Web service are encoded in eXtensible Markup Language (XML), using SOAP. SOAP is used to describe how to instantiate a remote procedure call, including the passed data and the returned results.

2.6.1 Providing Web Services from COBOL

For those using the J2EE platform, we offer a native Java interface that encapsulates the ACUCOBOL-GT runtime in a JAR file. By invoking the Java class contained in this archive, a Web service running on J2EE can start the runtime and run your COBOL program. Your program becomes a service to another program or service.

Consumers of the COBOL service use their own tools to incorporate a client proxy into their application. On the WebLogic platform, for instance, they use the Web Services Tool Kit (WSTK) of WebLogic Workshop to update their Web service.

As with any Web service, the consumer application must also contain logic for connecting to the COBOL server. It must also be programmed to issue a Web service request to the specified URL via XML/SOAP.

2.6.2 Consuming Web Services in COBOL

To enable a COBOL program to consume a Web service, we provide the C$JAVA library routine. The Java programmer packages the WSDL from the Web service in a WAR file with all the necessary resources including a Java client proxy, and the COBOL programmer invokes the service by calling the C$JAVA routine and naming the proxy as a USING parameter.

3 Working with Windows Technologies

Key Topics

3.1 COBOL and Windows

ACUCOBOL-GT® is a graphical COBOL language that is uniquely suited for graphical Windows environments. ACUCOBOL-GT not only supports all major Windows technologies, but it runs on most 32-bit and 64-bit versions of the Windows operating system.

In this chapter, you will learn how to call dynamic link libraries (DLLs), access the Windows Application Programming Interface (API), use Visual C++ .NET, and enable Windows features like ".wav" files and print spoolers. You'll also learn how to interoperate with applications and data sources that conform to the Open Database Connectivity (ODBC) standard.

Component Object Model (COM), ActiveX®, and .NET are large enough subjects to warrant their own chapters. To learn how to leverage COM and ActiveX in your COBOL programs, refer to Chapter 4 of this guide. To learn to interact with .NET assemblies, refer to Chapter 5.

For information on graphical user interface (GUI) programming (including concepts such as windows, handles, events, methods, menus, and color mapping), please refer to *ACUCOBOL-GT User Interface Programming*.

3.2 Calling COBOL From Other Windows Programs

To simplify the process of calling an ACUCOBOL-GT program from other programming languages in a Windows environment, the ACUCOBOL-GT 32-bit Windows runtime is encapsulated in a DLL file. This DLL file has been further encapsulated in a COM server.

To call ACUCOBOL-GT from other Windows programs, you can do one of two things:

1. Create an object for the ACUCOBOL-GT COM server in the source program and call the methods of that object. This is covered in section 3.2.1.
2. Call the ACUCOBOL-GT runtime DLL in the source program. This call requires special declarations, and is explained in section 3.2.2.

The advantage of using the COM server is that you can treat the ACUCOBOL-GT system as a COM object. You do not need to insert declarations into the source code of the other programming language, you can operate in a multi-threaded environment, and the development environment is more intuitive. Using the ACUCOBOL-GT runtime DLL instead of the COM server can provide slightly improved performance and makes application distribution smaller and installation easier. However, the DLL can be called only from a single thread of execution. For example, if you call a COBOL program from a user interface control's event procedure, and the event procedure is called again before the COBOL program returns, you must detect this case and either wait or inform the user of the error.

Regardless of which approach you choose, when a program written in another language calls an ACUCOBOL-GT program, the data is passed as a pointer to a variant type for each parameter. The ACUCOBOL-GT program receives a handle for each parameter and uses a library routine to convert the data to COBOL types. When ACUCOBOL-GT data items are passed back to this program, they pass through another library routine that converts the data back into variant types. The C$GETVARIANT and C$SETVARIANT library routines are detailed in Appendix I in *ACUCOBOL-GT Appendices*.

The parameters of the COM server methods, or exported DLL functions, are all null-terminated strings, integers, or variant type variables. In some programming languages, such as Visual Basic (VB), the variant type is used by default for any variables that have not been assigned a data type. Because the variant type is used to represent many different types of data, you generally don't have to convert these types of data when they are assigned to a variant variable. The programming language automatically performs any necessary conversion. Because the ACUCOBOL-GT COM server and runtime DLL routines take variant type parameters, it is easy to receive variables from, or return variables to, other languages.

Note: All the examples in this section use Visual Basic as the source language for code samples. Microsoft conventions for object description language are used for the descriptions of method usage.

3.2.1 Using the ACUCOBOL-GT COM Server

For ease of use in Windows environments, the ACUCOBOL-GT runtime is available as a COM server. With the COM server, you can treat the ACUCOBOL-GT system as a COM object and include it in applications that support COM.

To use the COM server in other programs:

1. Register the ACUCOBOL-GT COM server. Registration occurs automatically when you load the ACUCOBOL-GT runtime using the setup program that comes with the software.

   When you distribute your application, if you are not using the ACUCOBOL-GT setup program, you will have to install and register the COM server on each user's machine. If you are using it as a remote server, you must install and register the ACUCOBOL-GT runtime with the COM server option on the server machine. Register the ACUCOBOL-GT COM server by running it with no command-line options or with the "/RegServer" option. This command-line option is not case sensitive.

   The ACUCOBOL-GT COM server executable is in the ACUCOBOL-GT bin directory after installation. This file is named "AcuGT.exe". The ACUCOBOL-GT COM server requires the same files as the ACUCOBOL-GT runtime, except for "wrun32.exe". Two additional files, "AcuGT.exe" and "AcuGT.tlb", must be installed on the machine in a single directory. For the COM server to work, the runtime DLL "wrun32.dll" must either be in the same directory as "AcuGT.exe" or somewhere else in the Windows DLL search path. If you move "AcuGT.exe" to a different directory, you must register it again from the new location.

   Note: If you ever need to unregister the ACUCOBOL-GT COM server, run "AcuGT /UnregServer".

2. Start the other programming language's development environment and add the current "ACUCOBOL-GT Library" to your project references.

3. Add code to declare and create the AcuGT object. For example, in Visual Basic you could enter:

   ```
   Dim cblObj As Object
   Set cblObj = New AcuGT
   ```

4. Control the ACUCOBOL-GT COM server using the “Initialize”, “Call”, “Call50”, “Cancel”, and “Shutdown” methods described in section 3.2.1.1. For example, from Visual Basic you would enter:

   ```
   cblObj.Initialize "-d"   ' Start ACUCOBOL-GT in debug mode
   retVal = cblObj.Call(programName, arg0, arg1, arg2)
   cblObj.Shutdown
   ```

   or use the “With” construct. For example:

   ```
   With cblObj
     .Initialize "-e @myserver:\myprogs\errorfile"
     .Call "*myserver:\myprogs\program1.acu", "call1", 1.2, 37
     .Call "*myserver:\myprogs\program1.acu", "call2", 2.3, 38
     .Call "*myserver:\myprogs\program1.acu", "call3", 3.4, 39
     .Cancel "*myserver:\myprogs\program1.acu"
   End With
   ```

If you don’t explicitly call “Initialize”, the COM server calls it for you, passing an empty command-line parameter. Likewise, if you don’t explicitly call “Shutdown”, the COM server calls it for you when the object is destroyed.

In this example, after the AcuGT object is created in Visual Basic, “Initialize” is called automatically. Then, when the AcuGT object is destroyed at the end of the subroutine, the “Shutdown” method is called automatically:

```
Private Sub Command1_Click()
  Dim cblObj As Object
  Set cblObj = New AcuGT
  cblObj.Call "program"
End Sub
```

If you have several COBOL calls to make, it is much more efficient to create the AcuGT object as a Public variable in the module, class, or form initialization. For example, this may be done using the Visual Basic CreateObject function:

```
Dim cblObj As Object
Set cblObj = CreateObject("AcuGT.Application");
```

The Visual Basic CreateObject function takes an optional second parameter—the name of the network server where the object is created. For example, if you want to run the COBOL program on a remote machine named MOOSE, use the following syntax:

```
Set cblObj = CreateObject("AcuGT.Application", "MOOSE");
```

The COM server sets the “current directory” for COBOL programs to the directory containing “AcuGT.exe”. This allows you to use relative directory paths when you specify file names. For example, suppose you have installed the COM server in C:\AUTOSRV\BIN, the COBOL programs and configuration files you want to use in C:\AUTOSRV\PROGRAMS and the data files in C:\AUTOSRV\DATA. You could then call the “Initialize” method with “-c ..\PROGRAMS\CONFIG”, set CODE_PREFIX to “..\PROGRAMS” and set FILE_PREFIX to “..\DATA”.

The ACUCOBOL-GT COM server is thread-safe, meaning that you can run COBOL programs asynchronously. To do this, you must create a new thread and a new AcuGT object in that thread. Then you call the COBOL program from that thread.

For an example of how to create new threads in Visual Basic, see “Creating a Multithreaded Test Application” in the Visual Basic documentation.

It is a good idea to trap errors and handle them with your own Visual Basic error handler. For example:

```
   On Error GoTo ErrHandler
   cblObj.Call "program"
Exit Sub

ErrHandler:
   myval = MsgBox(Err.Description, vbOKOnly,
      "Call not successful")
End Sub
```

3.2.1.1 Methods of the COM server object

The ACUCOBOL-GT COM server object has the following methods:

- Initialize
- Shutdown

- Call
- Call50
- Cancel

Initialize

Initializes the ACUCOBOL-GT runtime

Usage

```
HRESULT Initialize([in] VARIANT *cmdLine)
```

Return value

"Initialize" returns one of the following result codes (note that these values are given in hexadecimal format):

Name	Value	Description
S_OK	0	Call succeeded
ACUGT_E_UNEXPECTED	80040200	Unexpected error
ACUGT_E_INITIALIZE	80040201	COM initialization failed. Make sure that the COM libraries are the correct version.

Shutdown

Shuts down the runtime

Usage

```
void Shutdown(void)
```

Call

Calls the runtime

Usage

```
HRESULT Call([in] VARIANT *name,
   [in, out, optional] VARIANT *arg0,
   [in, out, optional] VARIANT *arg1,
   [in, out, optional] VARIANT *arg2,
   [in, out, optional] VARIANT *arg3,
   [in, out, optional] VARIANT *arg4,
   [in, out, optional] VARIANT *arg5,
   [in, out, optional] VARIANT *arg6,
   [in, out, optional] VARIANT *arg7,
   [in, out, optional] VARIANT *arg8,
   [in, out, optional] VARIANT *arg9,
   [in, out, optional] VARIANT *arg10,
   [in, out, optional] VARIANT *arg11,
   [in, out, optional] VARIANT *arg12,
   [in, out, optional] VARIANT *arg13 )
```

Return value

"Call" returns one of the following result codes (note that these values are given in hexadecimal format):

Name	Value	Description
S_OK	0	Call succeeded
ACUGT_E_UNEXPECTED	80040200	Unexpected error
ACUGT_E_MULTIPLE_THREADS	80040203	Call (or Call50) has been called in multiple threads (see "Calling the Runtime DLL").
ACUGT_E_INITIALIZE_FAILED	80040204	Initialize failed. (Initialize cannot be called after Shutdown in a single process.) (See "Calling the Runtime DLL.")
ACUGT_E_PROGRAM_MISSING	80040205	Program missing or inaccessible

Name	Value	Description
ACUGT_E_NOT_COBOL	80040206	Not a COBOL program
ACUGT_E_CORRUPTED	80040207	Corrupted program
ACUGT_E_INADEQUATE_MEMORY	80040208	Inadequate memory available
ACUGT_E_UNSUPPORTED	80040209	Unsupported version of object code
ACUGT_E_PROGRAM_IN_USE	8004020A	Program already in use
ACUGT_E_TOO_MANY	8004020B	Too many external segments
ACUGT_E_CONNECTION_REFUSED	8004020C	Connection refused; perhaps AcuConnect® is not running.

Call50

"Call50" calls the runtime in the same way as "Call", except that you may have up to 50 optional parameters. Substitute "Call50" for the word "Call" in the Visual Basic syntax. The return values are exactly the same.

Note: When you call using "Call" or "Call50", you may not exceed 14 parameters for "Call" or 50 parameters for "Call50".

Cancel

Cancels a COBOL program

Usage

```
void Cancel([in] VARIANT *program )
```

3.2.2 Calling the Runtime DLL

For 32-bit Windows users, the ACUCOBOL-GT runtime is available in a DLL file.

To call the runtime DLL from another programming language, you must add certain declarations to the source program. This example shows what you would use in Visual Basic:

```
Declare Function AcuInitialize Lib "wrun32.dll" _
   (Optional ByVal cmdLine As String) As Integer

Declare Sub AcuShutdown Lib "wrun32.dll" ()

Declare Function AcuCall Lib "wrun32.dll" _
   (ByVal name As String, _
     Optional param1, _
     Optional param2, _
     Optional param3, _
     Optional param4, _
     Optional param5, _
     Optional param6, _
     Optional param7, _
     Optional param8, _
     Optional param9, _
     Optional param10, _
     Optional param11, _
     Optional param12, _
     Optional param13, _
     Optional param14) As Integer

Declare Function AcuCall50 Lib "wrun32.dll" _
   (ByVal name As String, _
     Optional param1, _
...
     Optional param50) As Integer
```

Note: Two declarations are shown. "AcuCall" supports 14 optional parameters, and "AcuCall50" supports 50 optional parameters. "AcuCall50" has the same format as "AcuCall", except that you must include the full list of 50 parameters in your declaration. The code example for "AcuCall50" was abbreviated.

```
Declare Function AcuGetCallError Lib"wrun32.dll" () As Integer

Declare Sub AcuCancel Lib "wrun32.dll" (ByVal name As String)
```

After you add the declarations, you initialize the runtime, call the COBOL program(s) passing the program name and parameters, and shut down when you are finished. For example, in Visual Basic you would perform the following steps:

1. Call “AcuInitialize” to pass the runtime’s command-line options. For example:

   ```
   returnValue = AcuInitialize("-c myconfig -le myerrors")
   ```

   “AcuInitialize” returns a value of “0” on success and “-1” on failure. You can safely call “AcuInitialize” multiple times. The command line from the first call is used and is ignored on subsequent calls.

   You may use the runtime “-d” option to debug your ACUCOBOL-GT program. Specify “-d” in the command line to “AcuInitialize”, and the debugger window appears when you actually call a COBOL program.

2. Then call “AcuCall” or **“**AcuCall50”, passing the program name and parameters. For example, to call the program “vb-test” with “AcuCall”, enter:

   ```
   returnValue = AcuCall("vbtest.acu", testNum,
      testStr, testLongNum, testFloat)
   ```

   “AcuCall” returns “0” on success and “-1” on failure. If “AcuInitialize” hasn’t been called yet, “AcuCall” calls it, passing an empty command line. If “AcuCall” returns “-1”, you may call “AcuGetCallError” to get the error code. The error codes are as follows:

-4	“AcuCall” (or “AcuCall50”) has been called in multiple threads.
-3	“AcuInitialize” failed. (“AcuInitialize” cannot be called after “AcuShutdown” in a single process.)
1	Program missing or inaccessible
2	Not a COBOL program
3	Corrupted program

4	Inadequate memory available
5	Unsupported version of object code
6	Program already in use
7	Too many external segments
25	Connection refused; perhaps AcuConnect is not running
27	Program contains object code for a different processor.
28	Incorrect serial number
30	License error

Note: Two calls are available to you. “AcuCall” supports 14 optional parameters, and “AcuCall50” supports for up to 50 optional parameters. “AcuCall50” calls the runtime in the same way as “AcuCall”, just use the appropriate name in the Visual Basic syntax. The return values are exactly the same. The declaration is similar, but you must declare all 50 parameters.

---

3. Call “AcuShutdown” after you are completely finished using COBOL in your Visual Basic application. It is absolutely essential to call “AcuShutdown” from the VB program after the final call to COBOL. Failure to call “AcuShutdown” before the VB program ends will likely cause the Visual Basic integrated environment to crash, resulting in the loss of any unsaved VB program changes.

Note: If a COBOL program issues a STOP RUN, “AcuCall” returns, the runtime environment is initialized, and the calling process continues running. This sequence occurs so that STOP RUN does not shut down the calling application or development environment you are using.

---

4. To cancel a COBOL program, call “AcuCancel” passing the name of the program. For example:

```
AcuCancel("vbtest.acu")
```

3.3 Calling DLLs from COBOL

ACUCOBOL-GT programs can call native code subroutines located in DLLs. This includes files with a ".dll" extension and Windows system files with a ".drv" or ".ocx" extension. You do not need to relink the runtime to access the routines in DLLs.

To use a routine located in a DLL, you must first load the DLL in one of three ways:

- Using the CALL statement
- Using configuration variables
- Using the "-y" runtime option

Note: If you are loading a DLL that contains Windows API functions, please refer to section 3.5. This section contains some specific guidelines and procedures that you should know when calling Windows API functions.

3.3.1 Loading DLLs with the CALL Statement

To load the dynamic link library "mylib.dll", you would CALL either "mylib.dll" or simply "mylib". The runtime automatically appends ".dll" when searching for a DLL. The runtime searches for DLLs after it has searched for COBOL programs. It first searches the paths defined in the CODE_PREFIX configuration variable, and then in the System and Windows folders, respectively (actually, folder names vary depending on the specific version of Windows and user customizations). If you do not want the runtime to look in the System or Windows folders, set the DLL_USE_SYSTEM_DIR configuration variable to "0" (off, false, no).

Note: By default, the runtime also recognizes ".drv" and ".ocx" files, which enables you to load these file types just as you would a ".dll". For backwards compatibility, you can turn this feature off by setting the runtime configuration variable "USE_WINSYSFILES" to "0" (off, false, no). Then, only calls to ".dll" files are supported.

The CALL statement that loads a DLL simply makes the routines contained in the DLL available to the COBOL program. The one exception to this is if the DLL contains a routine whose name is the same as the DLL. In this case, the routine is immediately called. For example, if the DLL "mylib.dll" contains a routine called "mylib", then

```
CALL "mylib"
```

both loads the DLL and executes the MYLIB routine. To load the library and avoid calling the subroutine of the same name, specify ".dll" explicitly in the CALL statement, as shown below:

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

Thin client applications may call DLLs on the display host (client) by adding "@[DISPLAY]:" to the beginning of the CALL name. For example, to call a DLL named "mylib" on the client, you would use the following code:

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

For complete information, see section 6.5.6, "Calling Dynamic Link Libraries (DLLs)," in the *AcuConnect User's Guide*.

Once the library has been loaded, all of the routines it contains can be called. For example:

```
CALL "funcA"
```

Loaded DLLs are searched immediately prior to searching for COBOL programs on disk. Routines contained in a DLL are called using either the direct C interface or the Pascal/WINAPI interface. As a result, you may pass parameters BY VALUE if that is required by the routine.

Routines called by this method are assumed to use either the cdecl (standard C) or stdcall (Pascal/WINAPI) parameter passing conventions. Most of the Windows API library functions are stored in DLLs and must be called using the stdcall (Pascal/WINAPI) calling convention.

By default, the runtime uses the value of the DLL_CONVENTION configuration variable to determine the calling convention. A maximum of 16 parameters can be passed using the stdcall DLL_CONVENTION. A maximum of eight parameters can be passed using the cdecl (standard C) DLL_CONVENTION (the default). See Appendix H in *ACUCOBOL-GT Appendices* for more details. If you attempt to call a routine that uses a different calling convention, the results are undefined (and usually fatal).

You can override the calling convention for an individual function by specifying it after the function name in the CALL statement.

To specify the stdcall calling convention, append one of the following strings to the function name: @, @1, @WINAPI, @__stdcall. (The @__stdcall string has two underscores.) For example:

```
CALL "funcA@"
```

or

```
CALL "funcA@1"
```

or

```
CALL "funcA@WINAPI"
```

or

```
CALL "funcA@__stdcall"
```

calls funcA using the stdcall calling convention.

To specify the cdecl calling convention, append one of the following strings to the function name: @0, @STDC, @__cdecl. (The @__cdecl string has two underscores.) For example:

```
CALL "funcB@0"
```

or

```
CALL "funcB@STDC"
```

or

```
CALL "funcB@__cdecl"
```

calls funcB using the cdecl calling convention.

The runtime uses the specified calling convention and ignores the value of the DLL_CONVENTION configuration variable. Note that you cannot specify the calling convention for a DLL when specifying a DLL name in a CALL statement. You can, however, specify the calling convention for DLLs using configuration variables. Specifying conventions for individual functions in the CALL statement overrides any other conventions specified for the DLL name.

If desired, you could specify the calling convention for individual functions in the runtime configuration file instead of in the CALL statement. To do this, set the CODE_MAPPING variable to "1". If you use the following configuration entries:

```
CODE_MAPPING=1
funcA=funcA@__stdcall
funcB=funcB@__cdecl
```

then

```
CALL "funcA"
```

calls funcA using the stdcall calling convention and

```
CALL "funcB"
```

calls funcB using the cdecl convention.

If you have access to a library (".lib") file for the DLL, you can determine the calling convention for a particular function using the Microsoft COFF Binary File Dumper utility. Run "dumpbin /exports <library name>". If the function name is preceded by an underscore and followed by an "at" sign ("@") and a decimal number, the function expects to be called using the stdcall calling convention. If the name of the function is not followed by an at sign, then the function expects to be called using the cdecl convention.

Like other programs that are loaded with a CALL statement, you can unload a CALLed DLL with a CANCEL statement. When you CANCEL a DLL, you may no longer call its component libraries. Also, unless the *logical*

cancels feature is enabled, all memory used by the program is released. For information about runtime memory management and the logical cancels feature, see section 6.3, "Memory Management," in the *ACUCOBOL-GT User's Guide*.

Note: The CANCEL_ALL_DLLS configuration variable can be used to control whether CANCEL ALL frees DLLs. See Appendix H in *ACUCOBOL-GT Appendices* for more details.

3.3.2 Loading DLLs with Configuration Variables

Another way to load DLLs is to list them in the SHARED_LIBRARY_LIST configuration variable. SHARED_LIBRARY_LIST lets you specify a list of libraries to be automatically loaded by ACUCOBOL-GT at run time. It can be set in the environment, in the runtime configuration file, or programmatically with the SET ENVIRONMENT statement.

The library names in SHARED_LIBRARY_LIST are delimited by spaces or colons on UNIX and spaces or semicolons on Windows (like directories specified in the PATH variable). You can also 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 example, to specify the stdcall calling convention for "mylib.dll", you might include the following in your configuration file:

```
SHARED_LIBRARY_LIST=mylib.dll@__stdcall
```

This indicates that every function in "mylib.dll" should be called using the stdcall convention, regardless of the DLL_CONVENTION setting.

To specify the cdecl calling convention, you might include:

```
SHARED_LIBRARY_LIST=mylib.dll@__cdecl
```

This indicates that every function in "mylib.dll" should be called using the cdecl convention, regardless of the DLL_CONVENTION setting.

If you have access to a library (".lib") file for the DLL, you can determine the calling convention for a particular function using the Microsoft COFF Binary File Dumper utility, as described in the previous section. Run "dumpbin /exports <library name>". If the function name is preceded by an underscore and followed by an at sign and a decimal number, the function expects to be called using the stdcall calling convention. If the name of the function is not followed by an at sign, then the function expects to be called using the cdecl convention.

Note that 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. You can, however, call DLLs on the server using SHARED_LIBRARY_LIST.

Refer to Appendix H in *ACUCOBOL-GT Appendices* for more information on any of the configuration variables discussed in this section.

Note: To use shared libraries without CALLing them first, you can use the "-y" runtime option to place shared libraries on the command line. The runtime then uses the usual search logic to find the specified libraries when trying to resolve CALLs.

3.3.3 Loading DLLs with the "-y" Runtime Option

You can load DLLs by specifying "-y" on the command line at run time. You enter "-y" followed by the DLL name to load. Specifying:

```
-y lib1.dll -y lib2.dll -y lib3.dll
```

is equivalent to specifying

```
SHARED_LIBRARY_LIST=lib1.dll;lib2.dll;lib3.dll
```

You can also specify the calling convention in the "-y" argument. For example:

```
wrun32 ... -y lib1.dll@_stdcall -y lib2.dll@_cdecl
```

Note that the "-y" runtime option 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. You can, however, call DLLs on the server using the "-y" option.

Refer to section 2.2 of the *ACUCOBOL-GT User's Guide* for more details on the "-y" runtime option.

3.4 Working With Open Database Connectivity (ODBC)

To provide flexible interaction between your COBOL applications and data and popular commercial applications and data in the Windows environment, we offer two different ODBC interface technologies.

The first, Acu4GL® for ODBC, is designed to give your ACUCOBOL-GT applications access to ODBC-compliant data sources such as Microsoft Access. You can use standard COBOL I/O statements to access these data sources, locally or remotely.

The second, AcuXDBC™, gives ODBC-enabled applications such as Microsoft Word, Excel, and Access access to Vision data. Business Intelligence tools such as Crystal Reports® Professional, and custom applications developed in ODBC supported environments such as Visual Basic® are supported as well. AcuXDBC also works with Java Database Connectivity (JDBC).

This section provides general information about using our ODBC interfaces. For detailed instructions, please refer to the *Acu4GL User's Guide* and the *AcuXDBC User's Guide*.

3.4.1 What Is ODBC?

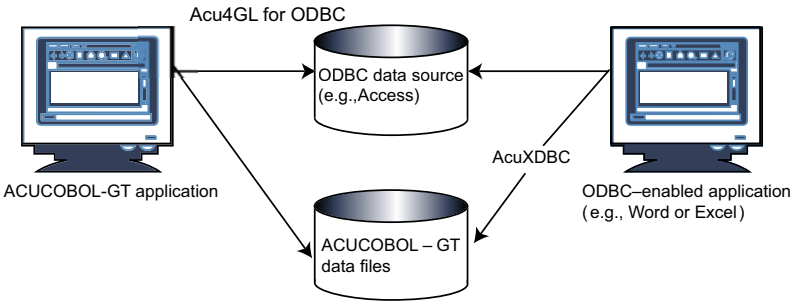
Developed by Microsoft, ODBC is a library of standardized data access functions used to connect Windows applications to relational and non-relational databases. It is intended to give programmers a way to access

and manipulate data in a variety of dissimilar data sources. ODBC can access a wide variety of file systems because it takes advantage of their common properties and common standards.

With traditional call-level interfaces, you need to learn the API for each data source and application. If you want to extend an application to support an additional data source, or port an application from one data source to another, you must write a complete new access module.

ODBC was designed expressly to provide access to any ODBC-compliant data source. It offers a standard API that can be used to manipulate data in a Microsoft Access database on your PC, or connect from your PC to an Oracle database on a UNIX host. It can even access files that are not databases, such as Excel spreadsheets.

Our two ODBC products, AcuXDBC and Acu4GL for ODBC, are designed to ease the use of ACUCOBOL-GT applications and data in a Microsoft Windows environment. Together, these ODBC interfaces give our customers a broad level of application and data flexibility.



The front-end ODBC product is known as Acu4GL for ODBC. ACUCOBOL-GT uses Acu4GL libraries to access information stored in relational database management systems (RDBMSs). Data dictionaries generated by the compiler guide the libraries in mapping the field names and data types that are passed between COBOL and the database engine. The essence of Acu4GL libraries is that standard COBOL I/O statements are used to access databases. Acu4GL dynamically generates industry-standard SQL from the COBOL I/O statements. As the ACUCOBOL-GT runtime module

is executing your COBOL application, Acu4GL is running "behind the scenes" to match up the requirements and rules of both COBOL and the RDBMS to accomplish the task set by your application.

AcuXDBC can be used as a back-end ODBC product. AcuXDBC gives ODBC-enabled Windows applications seamless access to ACUCOBOL-GT Vision files. AcuXDBC Server is an add-on to AcuXDBC that supports remote processing on a Windows NT/2000/2003/2008, UNIX, or Linux server. Whether your Vision data is on a Windows machine, network server, or UNIX server, AcuXDBC can make it accessible to many popular Windows applications, including Excel, Word for Windows, Access, and Microsoft Query.

3.5 Accessing the Windows API

The Windows API lets you develop applications that take advantage of the features and capabilities of the Windows operating system. In essence, the Windows API provides building blocks that can be used by applications written for Windows, regardless of the language those applications are written in.

If desired, you can call Windows API functions from ACUCOBOL-GT, giving your COBOL program features and functions that are unique to Windows. You can give your application a graphical user interface; display graphics and formatted text; and manage system objects such as memory, files, and processes. Literally hundreds of Windows API functions are available.

Windows API functions are contained in DLLs. To access a Windows API function from ACUCOBOL-GT, you load the DLL and then call the function. (The procedure for this is outlined in section 3.5.3, subsection "Procedure.")

Before you get started, you should become familiar with the documentation that is available, the DLLs that you'll use most often, and some guidelines specific to ACUCOBOL-GT and data mapping. This information is provided in sections 3.7.1 through 3.7.3.

3.5.1 Microsoft Documentation

Microsoft Corporation documents the Windows APIs in two main places: in the Microsoft Software Developer's Kit (SDK), and on the Web site, msdn.microsoft.com. Both of these are excellent sources of information on the various Windows API functions. In order to learn how to access and use a Windows API function, you need to refer to these documents frequently.

In the documentation for each function, you can learn the DLL name that contains the function (the DLL name to call) and whether or not ANSI and Unicode versions are available (for functions that handle text). All of this information is listed in the requirements table for the function.

If both Unicode and ANSI versions of the function are available, you must append an "A" or "W" to the function name when you call it in ACUCOBOL-GT. "A" stands for ANSI, "W" for wide code/Unicode. For instance, the GetUserName function is implemented as GetUserNameW (Unicode) and GetUserNameA (ANSI). In some cases, the requirements table supplies the function names for Unicode and ANSI. In other cases it simply states:

```
Unicode:  Implemented as Unicode and ANSI versions
```

Either way, be sure to include the "A" or "W" in the function name to indicate which version you want to call.

Note that Windows API function names are case sensitive. You must adhere to case sensitivity of the function name when you call it.

Finally, the Microsoft documentation defines the parameters for the Windows API functions. You will create variables for each of these parameters in your program's Working-Storage Section.

3.5.2 Useful Windows API DLLs

Although many Windows API DLLs are available, the following three may be used frequently:

- "user32.dll", for clipboard functions

- "kernel32.dll", for Getxxx functions
- "mpr.dll", for network functions

To see a list of the functions available in these or any DLLs, place the DLL in your \system directory then run the utility known as "depends.exe". The "depends.exe" file is in the \tools directory of the Microsoft SDK.

3.5.3 Calling a Windows API function from ACUCOBOL-GT

The following sections provide some general rules and guidelines for calling a Windows API function from ACUCOBOL-GT.

General rules

1. You must load the DLL to call it's function. (Even if it is already loaded by the system, you must implement it.) There are three ways to load a DLL in ACUCOBOL-GT: you can CALL it, use the "-y" runtime option, or use the SHARED_LIBRARY_LIST configuration variable. These are described fully in section 3.3 of this chapter.

2. If you CALL a DLL, you can enter any of the following:

   ```
   CALL "MyDll"
   CALL "MyDll.dll"
   CALL "C:\MyDir\MyDll.dll"
   ```

   We recommend the middle option. The last won't work if your system is mobile. If you set CODE_PREFIX, the runtime looks for the DLL in the CODE_PREFIX directory. If it is not found, it also tries the Windows system directory.

3. You must indicate the DLL calling convention in your COBOL program. The vast majority of the time, the convention for Windows API DLLs is WINAPI also known as *stdcall*. You can specify calling conventions in ACUCOBOL-GT in several ways:

   - Using the DLL_CONVENTION variable. To indicate WINAPI/stdcall, you would set DLL_CONVENTION to "1".

- Using the SHARED_LIBRARY_LIST variable
- In the CALL statement for individual library functions
- Setting the CODE_MAPPING variable to “1”, then using configuration entries to specify the calling convention for individual functions
- Using the “-y” runtime option

Section 3.3.1 describes the nuances of specifying calling conventions.

4. You must indicate “A” or “W”, ANSI/wide, in the function name if both ANSI and Unicode versions of the function are available. See section 3.5.1 for a more detailed description.
5. You must adhere to case sensitivity of the function name.
6. You must terminate any strings that you pass to a function. Use x“00” (LOW-VALUES) as in the following examples:

   ```
   STRING "My example" LOW-VALUES DELIMITED BY SIZE INTO
   target-string.
   ```

   or

   ```
   INSPECT target-string REPLACING TRAILING SPACES BY
   x"00".
   ```

7. Do not use literals with Windows functions.
8. You must map your ACUCOBOL-GT handle to the Windows handle. For example:

   ```
   INQUIRE acuhandle SYSTEM HANDLE IN syshandle
   ```

   Then “syshandle” can be passed on to the Windows API function. You declare the handle in Working-Storage like this:

   ```
   77 syshandle USAGE PIC X(4) COMP-N.
   ```

9. You should cancel the DLL when finished using it. For example:

   ```
   CANCEL "mydll.dll".
   ```

   Note that DLLs loaded with “-y” or SHARED_LIBRARY_LIST cannot be cancelled or unloaded.

10. Always pass string variables BY REFERENCE. In C, BY REFERENCE is indicated by an "&". You can also use BY REFERENCE when the data type is prefixed with LP, as in LPSTR.

    You can pass a numeric value BY REFERENCE or BY VALUE, depending on how the function is implemented. (Refer to the Microsoft documentation.)

    CALL a function BY CONTENT to make a copy of it and provide an address to the copy. This lets users read, not modify, content.

11. Most Windows API functions return only "true" or "false" when they succeed or fail. You must call GetLastError for specific reasons.

12. Whatever memory you allocate via ACUCOBOL-GT's M$ALLOC routine, it is your responsibility to release/free via M$FREE.

Data mapping

1. When mapping Windows API data types to COBOL, you can use COMP-5 or COMP-N as shown below:

Windows	**COBOL**	**Data Size**
char[*n*], str[*n*], tchar[*n*]	PIC X(*n*)	*n* bytes
long, int	PIC X(4) COMP-N or PIC S9(9) COMP-5	4 bytes
dword, ulong, lp*xxx*, float	PIC X(4) COMP-N or PIC 9(9) COMP-5	4 bytes
short	PIC X(2) COMP-N or PIC S9(5) COMP-5	2 bytes
word, ushort	PIC X(2) COMP-N or PIC 9(5) COMP-5	2 bytes

We recommend using COMP-N for most cases, although COMP-N data types are HEX, unsigned, and thus difficult for negative numbers.

If you want to use COMP-5, be sure to use the "--TruncANSI" compiler option. This causes truncation in binary to the capacity of the allocated storage for COMP-5 items. (By default, ACUCOBOL-GT truncates in decimal to the number of digits given in the PICTURE clause on arithmetic and non-arithmetic stores into COMP-5 items.) Note that the "-Dz" option overrides "--TruncANSI".

2. When defining C data types in COBOL, remember to maintain the data size shown in the table. It is imperative to make items the same, static size.

3. A structure is a group of data, a virtual container for many different C data items. To include a C structure (struct) in ACUCOBOL-GT, ignore the first and last line in the structure, and create a COBOL group item as shown in the following example:

C structure to include:

```
typedef struct _WIN32_FIND_DATA {
  DWORD     dwFileAttributes;
  FILETIME  ftCreationTime;
  FILETIME  ftLastAccessTime;
  FILETIME  ftLastWriteTime;
  DWORD     nFileSizeHigh;
  DWORD     nFileSizeLow;
  DWORD     dwReserved0;
  DWORD     dwReserved1;
  TCHAR     cFileName[ MAX_PATH ];
  TCHAR     cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA;
```

Resulting ACUCOBOL-GT group item:

```
01 WIN32-FIND-DATA.
  03 dwFileAttributes   PIC X(4) COMP-N.
  03 ftCreationTime.
     05 dwLowCreateDT   PIC X(4) COMP-N.
     05 dwHighCreateDT  PIC X(4) COMP-N.
  03 ftLastAccessTime.
     05 dwLowAccessDT   PIC X(4) COMP-N.
     05 dwHighAccessDT  PIC X(4) COMP-N.
  03 ftLastWriteTime.
```

```
           05 dwLowWriteDT    PIC X(4) COMP-N.
           05 dwHighWriteDT   PIC X(4) COMP-N.
        03 nFileSizeHigh      PIC X(4) COMP-N.
        03 nFileSizeLow       PIC X(4) COMP-N.
        03 dwReserved0        PIC X(4) COMP-N.
        03 dwReserved1        PIC X(4) COMP-N.
        03 cFileName          PIC X(260).
        03 cAlternateFileName PIC X(14).
```

Limits

ACUCOBOL-GT does not support:

- Callback functions
- Microsoft Foundation Class (MFC) objects

In addition, the GetMessage function causes the ACUCOBOL-GT runtime to stop while the function waits for a message. Use PeekMessage instead. If no message is sent, the function returns.

Procedure

To call a Windows API function:

1. Determine the name of the desired function and the DLL containing the function by looking at Microsoft's documentation. Also determine if both ANSI and Unicode versions are supported. For example:

   ```
   Unicode:  Implemented as GetUserNameW (Unicode) and
   GetUserNameA (ANSI)
   Required DLL:  AdvAPI32.DLL
   ```

2. Find the parameters for the desired function in the documentation. For example, the parameters for GetUserName are:

   lpBuffer: Pointer to the buffer to receive the null-terminated string containing the user's logon name

   nSize: On input, this variable specifies the size of the lpBuffer buffer, in TCHARs. On output, the variable receives the number of TCHARs copied to the buffer, including the terminating null character.

3. In your ACUCOBOL-GT program's Working-Storage Section, create a variable for each of the Windows API function parameters. For example:

```
77 user-name PIC X(40).
77 var-size PIC 9(9) COMP-5.
```

4. Set the DLL calling convention for the function in your program's Procedure Division. For example:

```
set environment "dll_convention" to 1.
```

5. Load the DLL containing the Windows API function. For example:

```
Call "advapi32.dll"
On exception go to err-load
```

6. Call the Windows API function. For example:

```
Set var-size to size of user-name.
call "GetUserNameA" using
by reference user-name
by reference var-size
```

   Other functions have parameters requiring the use of BY VALUE instead of BY REFERENCE. Read the Microsoft documentation carefully to make sure you pass parameters the proper way.

7. Terminate any strings that will be returned by the function. For example:

```
inspect user-name replacing all low-values by space.
```

8. Cancel or unload the DLL if appropriate. For example:

```
cancel "advapi32.dll"
```

   Note that DLLs loaded with "-y" or SHARED_LIBRARY_LIST cannot be cancelled or unloaded.

These general steps should be enough to get you started.

3.6 Using Visual C++ .NET

This section describes how to use ACUCOBOL-GT in conjunction with Visual C++ .NET. Visual C++ .NET, together with Visual Studio .NET, provides a 32-bit Windows *software development kit*.

Generally speaking, you use Visual C++ .NET to write various C subroutines that provide a specialized interface between your COBOL applications and the Windows operating system. You then build a new ACUCOBOL-GT runtime that contains these C subroutines, and you make calls to these routines from your COBOL application. You can also use this procedure to call code produced by code generators.

The remainder of this section assumes you are familiar with C and Visual C++ .NET. For information on including .NET assemblies in your COBOL program, or calling COBOL from a .NET application, refer to Chapter 5.

3.6.1 Building a New Runtime

Chapter 6 covers the details of writing C subroutines and passing parameters between those routines and COBOL. You may use any of the interface methods described there to call your C routines.

After you've written your routines, you'll need to link them into the runtime system. This process builds a new "wrun32.dll" that contains your routines.

In order to build a new runtime, you must have the ACUCOBOL-GT Windows runtime and Visual C++ .NET installed. The files that you need to rebuild the runtime can be found in the "library" subdirectory of the main ACUCOBOL-GT directory. To link, load "wrun32.sln" into Visual C++ .NET and perform a build. For more information, see Chapter 6.

Important tips before you rebuild the runtime

You may add your C routines directly to the "sub.c" or "mswinsub.c" files provided with ACUCOBOL-GT. More likely, you'll want to create your own files that hold your routines. If you do this, add the files to the "wrun32" project.

The file “mswinsub.c” contains useful declarations that you may want to use in your C subroutines. In particular, you will find variables for the “instance” and “main window” handles used by ACUCOBOL-GT. Also, you will find a start-up routine to which you can add your initialization code.

3.6.2 User Interface Approaches

You can use Visual C++ .NET for many things that don’t directly affect the user interface. For example, you can add dynamic data exchange with another application. If you want to add user interface code, however, you must decide whether to build your user interface using a mix of COBOL and C, or whether to use C alone. This decision affects how your code interacts with the runtime system.

Using C only, no COBOL

If you build your interface entirely in C, then you have complete flexibility in how the interface works. In this case, you either want to run the runtime system with the “-b” command-line option, or set the configuration variable NO_CONSOLE to “1”. When you do this, the runtime won’t create its own application window. Instead, your C code must build its own window. When you take this approach, you may not use ACCEPT or DISPLAY verbs in your COBOL program (except for those that don’t interact with the screen or keyboard). This approach also works well with a user interface created by a code-generating tool.

Using C and COBOL

If you want to use COBOL in conjunction with C, you must take care to cooperate with the runtime system in how the screen is displayed.

In some cases, you don’t need to worry about the runtime system, because 32-bit Windows manages everything. Generally speaking, this occurs when your C code displays data in its own window. For example, you can display and accept data from a dialog box without interacting with the runtime system (all you need is the handle of the runtime’s window, which you have in “mswinsub.c”).

In other cases, you'll need to cooperate with the runtime's message handler. For example, if you want to display a graphical object in the main application window, you must monitor "paint" messages to the runtime system and draw your object when appropriate. The general technique for doing this is called "subclassing." When you subclass a window, you instruct 32-bit Windows to pass all of its messages to your own message handler. Typically, your message handler acts on one or more messages and then passes all the messages to the original message handler. For detailed instructions on subclassing, see any 32-bit Windows programming text. The following is an example of a typical case.

Suppose that you want to intercept messages to the runtime system and pass them to a routine called "MyMsgHandler". You would first declare "MyMsgHandler" as a function designed to be called from 32-bit Windows:

```
LRESULT CALLBACK MyMsgHandler( HWND, UINT, WPARAM, LPARAM );
```

Next, in your start-up code, you would get the address of the ACUCOBOL-GT message handler and then direct 32-bit Windows to send messages to your handler instead. The code reads like this:

```
FARPROC  lpfnMyMsgHandler, lpfnAcuWndProc;

lpfnMyMsgHandler = MakeProcInstance((FARPROC) MyMsgHandler, hAcuInstance );
lpfnAcuWndProc = (FARPROC) GetWindowLong( hAcuWnd, GWL_WNDPROC );
SetWindowLong( hAcuWnd, GWL_WNDPROC, (long) lpfnMyMsgHandler );
```

At this point, all messages that 32-bit Windows would normally direct to the ACUCOBOL-GT main window procedure are instead received by "MyMsgHandler". Your message handler should intercept and act on the messages it cares about. At the end, it should pass each message on to the original message handler and return the result. This is usually done with a line that reads like this:

```
return CallWindowProc( lpfnAcuWndProc, hWnd, iMsg, wParam, lParam );
```

For reference, ACUCOBOL-GT for 32-bit Windows currently acts on the following messages:

WM_ACTIVATE	WM_LBUTTONUP
WM_ACTIVATEAPP	WM_MBUTTONDBLCLK
WM_CHAR	WM_MBUTTONDOWN

WM_CLOSE	WM_MBUTTONUP
WM_COMMAND	WM_MEASUREITEM
WM_CREATE	WM_MOUSEMOVE
WM_CTLCOLOR	WM_NCLBUTTONDBLCLK
WM_CTLCOLORBTN	WM_NCLBUTTONDOWN
WM_CTLCOLOREDIT	WM_NCPAINT
WM_CTLCOLORLG	WM_PAINT
WM_CTLCOLORLISTBOX	WM_PALETTECHANGED
WM_CTLCOLORMSGBOX	WM_QUERYDRAGICON
WM_CTLCOLORSCROLLBAR	WM_QUERYENDSESSION
WM_CTLCOLORSTATIC	WM_QUERYNEWPALETTE
WM_DESTROY	WM_RBUTTONDBLCLK
WM_DRAWITEM	WM_RBUTTONDOWN
WM_ENDSESSION	WM_RBUTTONUP
WM_ERASEBKGND	WM_SETCURSOR
WM_GETMINMAXINFO	WM_SETFOCUS
WM_HSCROLL	WM_SIZE
WM_INITMENU	WM_SIZING
WM_INITMENUPOPUP	WM_SYSCHAR
WM_KEYDOWN	WM_SYSCOLORCHANGED
WM_KILLFOCUS	WM_SYSCOMMAND
WM_LBUTTONDBLCLK	WM_TIMER
WM_LBUTTONDOWN	WM_VSCROLL

See the Visual C++ .NET documentation for details about these messages.

3.7 Windows-specific Features of ACUCOBOL-GT

The following sections describe ACUCOBOL-GT behaviors that are unique to 32-bit Windows environments.

Message Boxes

ACUCOBOL-GT applications deployed in 32-bit Windows environments can make use of the native Windows message box facility. A message box is a simple pop-up window that provides information to the user. In some instances it accepts a response (such as Yes or No) from the user. For further details, see DISPLAY MESSAGE BOX in *ACUCOBOL-GT Reference Manual*, Chapter 6.

Hardware Error Handling

Use the WIN_ERROR_HANDLING configuration variable to determine how hardware errors are handled.

When this variable is set to "1" (the default), certain errors are handled directly by 32-bit Windows, 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 click Retry. If the user clicks Cancel, then your program receives the file error that it would have normally received.

If you set WIN_ERROR_HANDLING to "0", then the dialog box is not shown and your program receives the error directly.

Special Characteristics of 32-bit Windows

In many ways, the 32-bit Windows environment operates quite differently from other environments. Included in these differences are memory access methods, interfaces to other languages, file extensions, and hardware error messages. Some of these differences could affect the way your programs execute.

Memory

The ACUCOBOL-GT runtime for 32-bit Windows is designed to handle memory management issues for you automatically.

SYSTEM library routine

The SYSTEM routine (described in Appendix I in *ACUCOBOL-GT Appendices*) can be used to initiate 32-bit Windows and console-mode programs. In either case, the program you start runs in its own window. While the called program is running, the window containing the calling program does not accept input from the user.

When CALL "SYSTEM" is used to initiate a program, it looks only for files with a ".exe" extension. If you want to call a ".com" or ".bat" file, you must explicitly add that extension in your code. For example:

```
CALL "SYSTEM" USING "MYBATCH.BAT"
```

C$SYSTEM library routine

The C$SYSTEM routine (also described in Appendix I in *ACUCOBOL-GT Appendices*) allows you to run other programs from inside a COBOL application by combining the functionality of the SYSTEM and C$RUN routines.

ACCEPT SYSTEM-INFORMATION FROM SYSTEM-INFO

ACUCOBOL-GT for 32-bit Windows returns "WIN/NT" as its host operating system when you use the ACCEPT FROM SYSTEM-INFO verb. This allows you to easily code 32-bit Windows-specific sections in your programs. The file "acucobol.def" contains SYSTEM-INFORMATION data items.

Assembly routines

Assembly routines cannot be linked into the 32-bit Windows runtime.

C$CHAIN library routine

The 32-bit Windows version of C$CHAIN behaves differently than it does in most other environments. A mechanism for "chaining" programs is not provided in 32-bit Windows. Instead, ACUCOBOL-GT initiates the chained-to program as an independent program and then halts the chained-from program.

Although this has the correct effect, 32-bit Windows causes the active program to shift from the chained-to program to some other program when the chained-from program halts. The runtime cannot determine which program is selected by 32-bit Windows as the active one.

The net effect is that the chained-to program pops up, but then becomes inactive and must be reactivated with either the mouse or the keyboard in order for the user to enter data. A mechanism for suggesting which program to activate when a program halts does not exist in 32-bit Windows.

3.7.1 Windows-specific Library Routines

ACUCOBOL-GT offers several Windows-specific library routines.

- WIN$PLAYSOUND lets you play a ".wav" file on Microsoft Windows machines.
- WIN$PRINTER enhances COBOL's ability take advantage of the Windows print spooler.
- WIN$VERSION lets you retrieve version information from Windows host machines.
- A number of library routines enable you to create and query Windows Registry keys.

For more information on these routines, please refer to Appendix I in *ACUCOBOL-GT Appendices*.

4 Using ActiveX Controls and COM Objects

Key Topics

4.1 Leveraging Ready-made Controls

You can use many different ActiveX® controls and COM objects in your ACUCOBOL-GT® program. When you add an ActiveX control or COM object to your program, whether it is graphical or non-graphical, the control or object becomes part of the development and run-time environment and instantly provides the application with new functionality.

COM objects are application components that perform well-defined functions, often involving a graphical user interface. ActiveX controls are COM objects that subscribe to the ActiveX component model originally developed by Microsoft. As such, they behave in a manner that developers can predict, they are reusable, and they are toolable. Pre-programmed controls (such as calendars, clocks, and gauges) are sold by third-party vendors along with the licensing rights to use them in your Windows application. Non-graphical controls and objects such as spell checkers and COM servers are also available, widening the opportunities for your COBOL application even further.

For your convenience, several Microsoft controls have been included with the ACUCOBOL-GT Windows runtime. You can use these controls in your program and redistribute them to your end users as needed. Refer to section 4.10 for a list of these controls.

By supporting ActiveX controls and COM objects, ACUCOBOL-GT allows you to take advantage of existing software functionality, as well as create applications that conform to the Windows standard.

This chapter provides a general overview of ActiveX and COM programming, including information on how to add ActiveX and COM objects to your program, avoid name clashes, distribute a program that contains an ActiveX or COM object, respond to events, handle errors, and more. Reference material is also provided in the ACUCOBOL-GT documentation set:

- The ACCEPT, CREATE, MODIFY, INQUIRE, and USE statements are described in Chapter 6 in *ACUCOBOL-GT Reference Manual*, along with common screen options.

- ActiveX terms, methods, events, control types, and color settings are described in *ACUCOBOL-GT User Interface Programming*.

- ActiveX configuration variables and library routines are described in *ACUCOBOL-GT Appendices*.

Note: Please note that ACUCOBOL-GT does not support windowless ActiveX controls. If you want to use a windowless control, see if it is available as a .NET assembly, then use our .NET interface to invoke the control. Refer to section 5.5.1, "Using .NET assemblies in COBOL" for details.

4.2 Adding ActiveX Controls or COM Objects to Your COBOL Program

You add ActiveX controls and COM objects to your ACUCOBOL-GT program using the utility program called **AXDEFGEN**. **AXDEFGEN** is provided on the ACUCOBOL-GT installation media for your convenience. You can launch it from Windows Explorer or AcuBench®, whichever you choose. Refer to section 4.20, "AXDEFGEN Utility Reference," for details.

The AXDEFGEN Utility

The role of **AXDEFGEN** is to locate the names of ActiveX controls and COM objects currently registered on the system and generate a COBOL COPY file for the control or object that you select. This COPY file is used by the ACUCOBOL-GT compiler for syntax and parameter type checking as well as efficient code generation.

The COPY file contains a Special-Names Class definition for the ActiveX control/COM object. After generating the COPY file with **AXDEFGEN**, you must copy the COPY file into the COBOL program's Special-Names paragraph in the Environment Division/Configuration Section. Please note that the Special-Names paragraph *must* end with a period. However, the COPY file generated by **AXDEFGEN** does not end with a period. This is so other definitions can be made in the Special-Names paragraph after the COPY statement that copies this COPY file. To complete the paragraph, you must add a period.

The Special-Names Class definitions in the COPY file generated by **AXDEFGEN** contain all of the information the compiler needs to know about the ActiveX control or COM object. This eliminates the need for the compiler to read the system registry or instantiate the ActiveX control during compilation of the COBOL program. As a result, the COBOL program can be compiled later on a UNIX or VMS machine, or on a Windows machine that does not have the particular ActiveX control or COM object registered.

In order for the control to function in your COBOL program, you must add it to your program. For ActiveX, you typically modify the Screen Section to include the name of the control's primary interface. This name can be determined by looking at the COPY file section "***Primary Interface***." Occasionally, you may want to create an ActiveX control in your Procedure Division. In this case, you use the DISPLAY statement.

Note: If you are using the AcuBench integrated development environment (IDE), you can drag the ActiveX control from the Screen Component Toolbox and drop it onto your screen in the Screen Designer. When run from AcuBench, **AXDEFGEN** automatically populates the toolbox with the ActiveX controls of your choice. Refer to the *AcuBench User's Guide* for details.

To create a COM object, you add a CREATE statement to your program's Procedure Division. Unlike ActiveX controls, COM objects cannot be created using the Screen Section or DISPLAY statement.

To Add an ActiveX Control or COM Object to Your ACUCOBOL-GT Program

1. If it is not installed already, install and register the ActiveX control or COM object of interest on your development system.

   Complex controls may come with their own setup programs, licensing, and registration wizards. Look for a setup program or "readme" file in the directory containing the control. Run the setup program if you find one. This program will likely install and register the control for you. Read the "readme" file, if one exists, for any installation and registration instructions. If you cannot find any instructions on the control, you can also visit the control vendor's Web site.

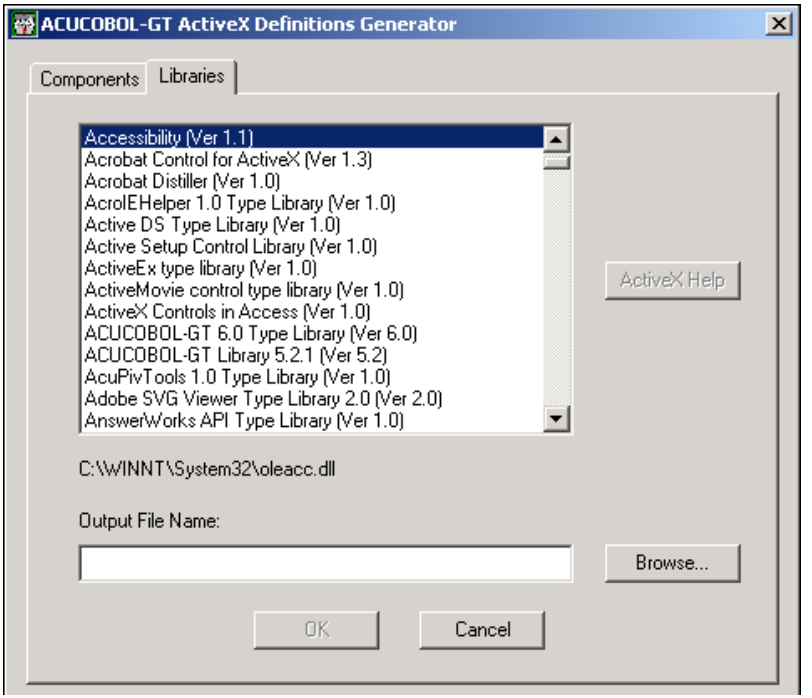
For simple controls, you can usually accomplish installation and registration by copying the ActiveX control files (at least a ".ocx" or ".dll" file) to your hard disk and executing the following command:

```
regsvr32 <ocx or dll name>
```

The "regsvr32.exe" file is normally located in your \windows\system directory. Do not assume that it is in your search path. Even if the control is already installed on your machine (for instance, if it came with other software that you've purchased) you may still need to register the control with "regsvr32.exe".

2. Use Windows Explorer to locate "axdefgen.exe". You'll find it in the \AcuGT\bin directory wherever you installed ACUCOBOL-GT on your machine (C:\Program Files\Acucorp\Acucbl8xx\AcuGT\bin by default).

3. Double-click the AXDEFGEN icon to start the utility. A list of all the controls and COM objects currently in the local machine's Windows registry appears. (If desired, you can launch "AXDEFGEN" from the command line. Refer to section 4.20 of this book for applicable command-line options.)



4. Select the ActiveX control or COM object that you'd like to include in your COBOL program, then specify an output path and filename for the COPY file. Click **OK** when done. The utility automatically generates a COPY file for the chosen control and appends the ".def" filename extension to the file.

   Many ActiveX controls and COM objects have documentation available. If they do, the ActiveX Help push button on the right side of this box is enabled. Click the button to read the help file for the selected ActiveX control or COM object. If the ActiveX Help push button is disabled, then **AXDEFGEN** could not locate a help file for the selected control.

5. In a code editor, open your ACUCOBOL-GT program and go to its Environment Division/Configuration Section.

6. Copy the new COPY file into the COBOL program's Special-Names paragraph. If you are adding several controls, copy several COPY files into this paragraph. Add a period at the end of the paragraph. For example:

   ```
   SPECIAL-NAMES
   COPY "calendar.def".
   COPY "chart.def".
   .
   ```

   ACUCOBOL-GT includes an "acuclass.def" file, which contains stock class definitions for ActiveX. We recommend that you also copy it into your program's Special-Names paragraph, as in:

   ```
   COPY "acuclass.def".
   ```

7. Add the control to your program. If you are adding a COM object, you add the object using the CREATE statement in your program's Procedure Division. For example:

   ```
   *Create an instance of the Outlook application
   CREATE Application OF Outlook
        HANDLE IN myOutlook
        EVENT PROCEDURE OUTLOOK-EVENTS.
   ```

   If you are adding an ActiveX control, however, you can do one of two things:

   a. Add the new control to your screen in your program's Screen Section. For example:

      ```
      SCREEN SECTION
      ...
      03 calendar-item Calendar column 5,
         line 5, size 60, lines 20
      ...
      ```

      or

   b. Create the control using the DISPLAY statement in your program's Procedure Division. For example:

      ```
      *Declare the calendar control's handle.
      ```

```
       77 CALENDAR-1 USAGE IS HANDLE OF Calendar.
       ...
    *Create an instance of the calendar control
       DISPLAY Calendar line 4, column 6,
          lines 10, size 40
          EVENT PROCEDURE IS CALENDAR-EVENT-HANDLER
          HANDLE IN CALENDAR-1.
```

You can determine the name of the control by opening the COPY file (".def" file) and looking at the section called "***Primary Interface***." Look for the name of the control after the word "CLASS" in the primary interface definition.

If you set a property of the ActiveX control in the control property panel (thus storing it in the resource file) and that property has a corresponding ACUCOBOL-GT property (e.g., ENABLED), then the COBOL program must explicitly set the corresponding ACUCOBOL-GT property in the Screen Section or Procedure Division. For example, if the ActiveX ENABLED property is set to "false" in the ActiveX control property panel, then a line must be included in the initialization code of the COBOL program to set the ACUCOBOL-GT ENABLED property to "false", as in:

```
MODIFY ActiveXcontrol ENABLED FALSE
```

Note: The runtime ignores events from all controls while it is creating an ActiveX control. 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 runtime configuration variable to "1" (on, true, yes). Alternatively, you could avoid creating an ActiveX control when you are expecting an event.

---

8. If desired, you can modify a control's properties or invoke methods using the MODIFY verb as in the following example:

```
MODIFY ActiveXcontrol Method ("parameters").

MODIFY ActiveXcontrol Property-name = property-value.
```

or

```
MODIFY ActiveXcontrol PROPERTY 37 = ("parameters").
```

Note that 37 is the ActiveX or COM "property" number of Property-name or Method. You can determine the property number by opening the COPY file and searching for the name of the property or method. The "property" number, also known as the dispatch id or dispid, precedes the name in the COPY file. The equal sign is optional.

To disable an ActiveX control, you use the MODIFY statement:

```
MODIFY ActiveXcontrol ENABLED FALSE.
```

If the ActiveX control has its own property named ENABLED, the MODIFY statement must explicitly set both the ACUCOBOL-GT ENABLED property and the ActiveX property (@ENABLED; the "@" symbol signifies that the property is a property of the ActiveX control). For example:

```
MODIFY ActiveXcontrol ENABLED FALSE @ENABLED FALSE.
```

To enable the control:

```
MODIFY ActiveXcontrol ENABLED TRUE @ENABLED TRUE.
```

To inquire about the value of one of a control's properties, use the INQUIRE verb, as in:

```
INQUIRE ActiveXcontrol Property-name IN value-item.
```

or

```
INQUIRE ActiveXcontrol PROPERTY 37 IN value-item.
```

9. If desired, modify your program to respond to one of the control's events (e.g., a mouse click). For example:

```
calendar-1-event.
   evaluate event-type
      when msg-ax-event
         evaluate event-data-2
            when CalendarClick
               Perform ...
         end-evaluate.
      ...
   end-evaluate.
```

You can view a list of control events by opening the COPY file created by AXDEFGEN and looking at the section "Event Interface for the *xxx* Control."

10. Compile and run your modified program.

Note: In order for the control to work on your end user's machine, it must be installed and registered. If you are licensed to do so, you can distribute the control along with your application. Refer to section 4.10, "Distributing Applications Containing ActiveX Controls," for information about distributing an application that contains an ActiveX object.

4.3 Properties, Styles, and Methods

ActiveX controls have properties and styles just like ACUCOBOL-GT controls. In addition, ActiveX controls provide functions called *methods*. To set a property or style or to invoke (call) a method, you use the MODIFY verb. Use INQUIRE to get the value of a property or to get the style flags.

ActiveX methods can take any number of parameters or no parameters. They can also take optional parameters (i.e., parameters that can be omitted). You specify the parameters in COBOL by enclosing them in parentheses. If there are no parameters, include empty parentheses (). Optional parameters are always last. For example, if a method has three parameters, one of which is optional, the first two are required. The last one is optional and therefore may be omitted. For more information on ActiveX methods, refer to section 4.5 of *ACUCOBOL-GT User Interface Programming*.

Note that ActiveX properties and methods should always be prepended with an "@" sign in case they clash with COBOL reserved words or ACUCOBOL-GT graphical control property and style names. The "@" symbol identifies the relationship of the name to ActiveX. The same holds true for ActiveX enumerators, described in section 4.14.

When programming with ActiveX controls and COM objects, you can create simpler statements by using named parameters. ActiveX controls and COM objects provide the option of using named parameters as a shortcut for typing parameter values in MODIFY and INQUIRE statements. With named

parameters, you can provide any or all of the parameters, in any order, by assigning a value to the named parameter. This is especially useful when an ActiveX/COM method or property has several optional arguments that you do not always need to specify.

Note: You cannot use named parameters to avoid entering required parameters. You can omit optional parameters only.

Generally, if the COBOL program passes a parameter in binary or numeric form with no decimal point, the runtime creates a variant of type VT_I4 (4-byte signed integer) before passing it to the property or method. If the program passes the parameter in numeric with a decimal point or in floating point, then the runtime creates a variant of type VT_R8 (8-byte floating point number). And if the COBOL program passes an alphabetic or alphanumeric parameter, then the runtime creates a variant of type VT_BSTR (Unicode string) before passing it to the property or method.

Note that you do not have to make a conversion from an ACUCOBOL-GT Working Storage item to a variant datatype in order to pass data to an ActiveX/COM property or method. The runtime handles all conversion that is required for you. For example, if you have a method in your definition file like this:

```
METHOD, 10, @Send,
     "VARIANT" @From, TYPE 12,
     "VARIANT" @To, TYPE 12,
     "VARIANT" @Subject, TYPE 12,
     "VARIANT" @Body, TYPE 12,
     "VARIANT" @Importance, TYPE 12
        OPTIONAL5
```

You do not have to convert your data before passing it. Rather, all you need to consider is what data to pass. (The runtime does the conversion so focus on the content.) In this case, you would pass sender, recipient, subject, message, and importance.

If the ActiveX or COM object with which your program is interacting expects the parameter of a different variant type, you must tell the runtime by using the AS *type-num* phrase in the parameter expression of the MODIFY

or INQUIRE statement (where *type-num* indicates the variant type to pass). The runtime then converts the parameter to the variant type that you specify before passing it to the object.

You can tell from the object's documentation and the name of the parameter whether the object expects a particular variant type, such as boolean.

4.3.1 Passing COBOL Data to Methods or Properties as SAFEARRAYs

When programming with ActiveX controls and COM objects, you can pass one- or two-dimensional COBOL tables to methods or properties that expect SAFEARRAY parameters. The runtime automatically converts a one- or two-dimensional COBOL table to a COM SAFEARRAY, as long as it contains only one elementary item that is USAGE HANDLE or USAGE HANDLE OF VARIANT.

The COM SAFEARRAY data type can contain elements of any type. Therefore, you must convert your COBOL data into variant type data before adding it to the array. Use the C$SETVARIANT library routine to create a new variant that stores the data if the initial value of the handle item passed to it is LOW-VALUES or SPACES. You need to free this variant using the DESTROY verb.

To use SAFEARRAYs, you should do the following:

1. Declare a table in Working-Storage that has one or two OCCURS clauses. If specified, the second OCCURS clause must be on an item that is subordinate to the item with the first OCCURS clause. The table must contain only one elementary item that is USAGE HANDLE or USAGE HANDLE OF VARIANT. (OF VARIANT is optional but makes the code more readable.)

2. Call C$SETVARIANT for each handle item in the table to convert the COBOL data to variant type data.

3. Use the name of this table wherever a property or method requires a SAFEARRAY.

For example, Microsoft Chart Control has a property called ChartData. The value of this property is a SAFEARRAY. Each element of the array is a data point value for the chart.

```
01  myTable.
    03  filler occurs 5 times.
        05  chart-data     usage handle of variant.

screen section.
01  screen-1.
    03  mschart-1
        MSChart
        line 3, column 5, size 50, lines 16.

    03  my-button push-button, "E&xit Program",
        ok-button,
        line 32, cline 23, column 27, size 13.

procedure division.
Main-Logic.

    perform varying col-number from 1 by 1
      until col-number > 5
        call "c$setvariant"
            using col-number, chart-data(col-number)
    end-perform.

    display standard graphical window,
        title "ActiveX Table MSChart Sample - tblchart.cbl"
        lines 37, size 66, background-low.

    display screen-1.

    modify mschart-1
           ChartData = myTable.

    perform, with test after, until exit-button-pushed
        accept screen-1
    end-perform.

    perform varying col-number from 1 by 1
       until col-number > 5
        destroy chart-data(col-number)
    end-perform.
    destroy screen-1.
```

```
    stop run.
```

Notice that the initial values of the chart-data table elements are spaces. When C$SETVARIANT is called with the chart-data items set to spaces, it creates new variant handles and sets the chart-data item to the variant handle.

The DESTROY statement destroys these handles and releases the associated memory. The DESTROY statement also sets the chart-data item to low-values to allow multiple destroys of the same handle item without any negative effects.

The following code is an example of a table with two OCCURS clauses passed as a two-dimensional SAFEARRAY to the ChartData property. Microsoft Chart Control takes the "string" elements of this array to be the *x*-axis labels and the numeric elements to be two series of chart data.

```
77  col-label       pic x(20).
77  series-2-data   pic 99.

01  myTable.
    03  filler occurs 5 times.
        05  filler occurs 3 times.
            07  chart-data   usage handle of variant.

screen section.
01  screen-1.
    03  mschart-1
        MSChart
        line 3, column 5, size 50, lines 16.

    03  my-button push-button, "E&xit Program",
        ok-button,
        line 32, cline 23, column 27, size 13.

procedure division.
Main-Logic.

    perform varying col-number from 1 by 1
       until col-number > 5
        string "Label " delimited by size
            col-number delimited by size
            into col-label
        call "c$setvariant"
            using col-label, chart-data(col-number,1)
```

```
            call "c$setvariant"
                using col-number, chart-data(col-number,2)
            multiply col-number by 2 giving series-2-data
            call "c$setvariant"
                using series-2-data, chart-data(col-number,3)
        end-perform.

        display standard graphical window,
            title "ActiveX Table MSChart Sample - tblchart.cbl"
            lines 37, size 66, background-low.

        display screen-1.

        modify mschart-1
               ChartData = myTable.

        perform, with test after, until exit-button-pushed
            accept screen-1
        end-perform.

        perform varying col-number from 1 by 1
           until col-number > 5
            destroy chart-data(col-number,1)
            destroy chart-data(col-number,2)
            destroy chart-data(col-number,3)
        end-perform.
        destroy screen-1.
        stop run.
```

Some ActiveX and COM objects that use SAFEARRAYs accept that some items in the array may be empty (for example, Microsoft ADO). Items that are empty are normally passed using the VT_EMPTY variant type.

If an element in an array that you want to pass is optional (i.e., it may sometimes be empty), you must tell the runtime this by coding one of the following:

```
    01 VariantParam.
       03 VariantTable usage handle of variant occurs 5.

       call "c$setvariant" USING x"00" VariantTable(1)
```

or

```
       call "c$setvariant" USING "" VariantTable(1)
```

This tells the runtime to convert the content of VariantTable(1) to VT_EMPTY. If you use the latter approach, you may receive and ignore compiler warnings that an empty literal was encountered.

4.3.2 Using COBOL Data Types as ActiveX and COM Object Parameters

When programming with ActiveX controls and COM objects, you can MODIFY property values using COBOL data items, literals, and figurative constants. You can also pass them as parameters to methods using MODIFY.

When passing parameters to an ActiveX control or COM object method, the ACUCOBOL-GT runtime converts the COBOL parameters into variant parameters. Variant parameters have a data type associated with them. More than 40 different variant types exist. Each type name starts with "VT_". For example, VT_I4 is a 4-byte signed integer. VT_R8 is an 8-byte real (floating point) number. VT_BSTR is a string. The rules applied by the runtime to determine the variant type are given at the end of this section.

A method or property can act differently depending on the type of data passed to it in a variant. For example, a type of variant parameter called VT_UNKNOWN is commonly used to represent COM objects. A property or method might expect either VT_I4 or VT_UNKNOWN (i.e., IUnknown pointer) and will act differently depending on which one it receives (see section 4.20.1, "AXDEFGEN COPY Files," for more information about IUnknown). If the runtime converts a COM object handle to a VT_I4 instead of a VT_UNKNOWN, the property or method being called might not act as expected. The runtime determines which of the two variant types to pass based on the USAGE parameters of the COBOL data item.

Some ActiveX and COM methods and properties take VT_VARIANT parameters, a generic variant type. This usually means that these methods accept more than one type of variant parameter. For example, you could pass a VT_I4, VT_R8, or a VT_BSTR as a VT_VARIANT parameter. The ActiveX or COM property or method could convert the passed parameter into a specific variant type. It could also act differently depending on the type of variant that was passed. For example, suppose an ActiveX or COM object had a table where rows could be referred to by a character string name or by their numeric index. A method that returns a row in the table could take

either the name or index of the row as a parameter. The method could assume that if the parameter is a VT_BSTR, it is a row name; and if the parameter is a VT_I4, it is a numeric index.

If the ActiveX control or COM object property or method parameter is one of the standard variant types, ACUCOBOL-GT attempts to convert the COBOL data to the expected type. For example, if an ActiveX control property is type VT_I4 (i.e., 4-byte integer), and the COBOL data type is PIC X(10), ACUCOBOL-GT tries to convert the value of the PIC X(10) item into a number and pass it as a VT_I4 type variant.

If the property or method parameter type is VT_VARIANT, then ACUCOBOL-GT converts the COBOL data item into a specific variant type parameter using the following rules:

1. The PICTURE and USAGE clauses determine the type.
2. If the data item is alphabetic, alphanumeric, or alphanumeric edited, it is passed as a VT_BSTR.
3. If the data item is USAGE HANDLE or POINTER, and the property or method parameter type is VT_VARIANT, it is passed as a VT_UNKNOWN. If the data item is another binary (USAGE COMP-..., BINARY, HANDLE, POINTER, etc.), it is passed as a VT_I4.
4. If the data item is USAGE FLOAT or DOUBLE, it is passed as a VT_R8.
5. If the data item is another numeric type, it is passed as a VT_I4.

Note that the compiler generates an error message if a figurative constant is passed as a parameter where the method or property expects a “by reference” parameter. That error message is, “Illegal parameter: literal”. This is the same error message you get when passing a figurative constant as a USING parameter in a CALL statement. One way to tell that the ActiveX/COM method expects a “by reference” parameter is by viewing the entry in the COPY file for that control or object. If the type has “BYREF” or if the numeric value divided by 16384 is odd, then you may not pass a figurative constant.

4.4 ActiveX and COM Events

ActiveX control and COM object events can have any number and type of associated parameters or no parameters. The event parameters are used to provide information about the event to the program. They can also be used to get information from the program in response to the event.

When an ActiveX control or COM object event occurs, the control or object invokes its event procedure. The EVENT-STATUS data item reflects the invoking event. EVENT-TYPE is either CMD-GOTO, CMD-HELP, MSG-VALIDATE, or MSG-AX-EVENT. For a description of these events, refer to section 6.3 of ACUCOBOL-GT User Interface Programming.

MSG-AX-EVENT (value 16436) occurs when an ActiveX control or COM object has “fired” an event. EVENT-DATA-2 contains the control’s event type. For COM objects, you can use the C$SETEVENTDATA and C$GETEVENTDATA library routines to set and get event the event parameters for the current event.

For ActiveX controls, however, you can use C$GETEVENTDATA/ C$SETEVENTDATA, or you can use the C$GETEVENTPARAM/ C$SETEVENTPARAM routines to get and set individual event parameters.

For example:

```
01 DATE-1
    03 MONTH PIC 99.
    03 FILLER PIC X VALUE '/'.
    03 DAY PIC 99.
    03 FILLER PIC X VALUE '/'.
    03 YEAR PIC 99.
77 KEY-ASCII PIC X USAGE COMP-X.
77 KEY-CHAR PIC X REDEFINES KEY-ASCII.
...

* Handle events
...
 CALENDAR-EVENT-HANDLER.
      EVALUATE EVENT-TYPE
         WHEN MSG-AX-EVENT
              EVALUATE EVENT-DATA-2
                 WHEN CalendarBeforeUpdate
* Don’t allow years >= 2020
```

```
                    INQUIRE EVENT-CONTROL-HANDLE
                       Value IN DATE-1
                    IF YEAR OF DATE-1 >= 2020
* Cancel the update (set the 'Cancel' parameter to 1)
                        CALL "C$SETEVENTPARAM" USING
                         EVENT-CONTROL-HANDLE, "Cancel", 1
                    END-IF
                WHEN CalendarKeyPress
* Stop run if the user presses 'X'
                    CALL "C$GETEVENTPARAM" USING
                         EVENT-CONTROL-HANDLE, "KeyAscii",
                            KEY-ASCII
                    IF KEY-CHAR = 'X' STOP RUN END-IF
...
```

Note that the CalendarBeforeUpdate event has one parameter, CANCEL. (See the CalendarBeforeUpdate definition in the control's COPY file.)

In this example, EVENT-CONTROL-HANDLE contains the handle of the control that fired the event (e.g., CALENDAR-1). C$SETEVENTPARAM is used to set the CANCEL parameter to "1" in response to a CalendarBeforeUpdate event when the year is 2020 or later. C$GETEVENTPARAM is used in the handling of the CalendarKeyPress event to get the key value and stop the runtime if it is "X".

For another example, suppose you have displayed an ActiveX control called "AX" whose handle is in AX-1. Further suppose that this control fires an event called AxEventOne, which has three parameters. You would use the following COBOL syntax to get the event parameters, add "2" to each one, and set the event parameters to their new values:

```
evaluate event-type
    when w-event
        evaluate event-data-2
            when AxEventOne
              call "c$geteventdata"
                 using event-control-handle,
                   param-1, param-2, param-3
              add 2 to param-1
              add 2 to param-2
              add 2 to param-3
              call "c$seteventdata"
                 using event-control-handle,
                   param-1, param-2, param-3
```

To use C$GETEVENTPARAM and C$SETEVENTPARAM, you must know the actual names of the parameters. You can determine these names by reading the ActiveX control's documentation or by looking at the definitions in the COPY file for the ActiveX control.

Note: Using the C$SETEVENTPARAM approach, you do not need to pass all of the event parameters. You need to specify only the name of the particular parameter you want to set. With C$SETEVENTDATA you don't need to specify parameter names, but you must pass an ordered parameter list up to the parameter you want to set.

An event commonly receives many parameters. C$GETEVENTPARAM and C$SETEVENTPARAM allow you to get and set the values of only the parameters you care about. Suppose in the above example that PARAM-1 and PARAM-2 contain information about the event and that only PARAM-3 is meant to be set by the event procedure. Because PARAM-3 is the third parameter, to set it you would have to pass two "dummy" parameters to C$SETEVENTDATA. For example,

```
call "c$seteventdata" using event-control-handle,
      0, 0, param-3.
```

Suppose you determined that the name of PARAM-3 in the ActiveX control was "Param3". You could then use C$SETEVENTPARAM to accomplish the task in our example in a more elegant and readable way. For example:

```
call "c$seteventparam" using event-control-handle,
      "param3", param-3.
```

In the Calendar example, you would use:

```
call "c$seteventparam" using event-control-handle,
      "cancel", 1
```

instead of:

```
call "c$seteventdata" using event-control-handle, 1
```

And you would use:

```
call "c$geteventparam" using event-control-handle,
      "KeyAscii", key-ascii
```

instead of:

```
call "c$geteventdata" using event-control-handle, key-ascii
```

Using these routines can make your code more readable. The object code will be a little larger, and your program will run slightly slower. However, these differences may be unnoticeable and the benefits of readable code can outweigh the performance and size considerations.

To determine in which specific window and control the event occurred, you can use the EVENT-WINDOW-HANDLE, EVENT-CONTROL-HANDLE, and/or EVENT-CONTROL-ID fields in the event procedure.

- EVENT-WINDOW-HANDLE holds the handle of the floating window in which the event occurred. If the event occurred in a control, this item is the handle of the floating window that contains the control.
- EVENT-CONTROL-HANDLE holds the handle of the control in which the event occurred. If the event did not occur in a control, this item is set to NULL.
- EVENT-CONTROL-ID holds the ID of the control in which the event occurred. IDs are assigned by the application when each control is created. If the event did not occur in a control, this item has the value zero.

During an ActiveX or COM event, you can refer to the control which “fired” the event using the EVENT-CONTROL-HANDLE item.

For more information on the C$GETEVENTDATA, C$SETEVENTDATA, C$GETEVENTPARAM, and C$SETEVENTPARAM library routines, please refer to Appendix I in *ACUCOBOL-GT Appendices*.

4.4.1 Event Timing

In order to properly handle ActiveX and COM events, it is important to understand how the runtime behaves in certain situations. Following are default runtime behaviors. If you want to change the behavior, you can add configuration variable(s) to your runtime configuration file.

- **The runtime ignores events from all controls while it is creating an ActiveX control or COM object.** 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 runtime configuration variable to "1" (on, true, yes).

- **The runtime suspends ActiveX and COM events when the application is not processing an ACCEPT statement; in other words, it suspends events in between ACCEPT statements.** If your control does not support the suspend/resume behavior and you are using it with the ACUCOBOL-GT Thin Client, this can be problematic. To prevent problems, set the TC_RESTRICTS_AX_EVENTS thin client configuration variable to "1" to mimic the runtime behavior.

- **The runtime allows events to trigger while it is processing other events.** If this is not handled properly, the runtime could execute the same code simultanously in two threads on the same data. Set NESTED_AX_EVENTS to "0" (off, false, no) if you do not want event procedures to be nested. Be aware that this option may cause you to lose certain events (typically events triggered by modifications made in the event procedure).

For details on these configuration variables, please refer to Appendix H of *ACUCOBOL-GT Appendices*.

4.5 ACTIVE-X Control Type

To use an ActiveX control in your COBOL program, we recommend that you define a new control type with properties, methods and events using the SPECIAL-NAMES CLASS clause. You can do this with the ActiveX definitions file generator, **AXDEFGEN**.

ACUCOBOL-GT defines a control type named "ACTIVE-X" that it uses internally whenever you CREATE, MODIFY, INQUIRE, or DESTROY an ActiveX control. For a complete description of the ACTIVE-X control type, refer to section 5.3 in *ACUCOBOL-GT User Interface Programming*.

4.6 Name Clashes

An ActiveX control may have property, method, or event names that are the same as COBOL reserved words or ACUCOBOL-GT standard property or style names. This situation creates ambiguity for the compiler. In addition, because ActiveX class names are used in the USAGE HANDLE clause of data description entries, in Screen Section items, and in DISPLAY statements, they may also cause ambiguities with COBOL reserved words.

To avoid these ambiguities, **AXDEFGEN** prepends an "at" sign character ("@") to every class, property, method, and event name in the generated COPY file.

In addition, ambiguity may occur with event names when two or more ActiveX controls define the same event name. To reduce this possibility, **AXDEFGEN** prepends the control name to each event name. For example, if an ActiveX control named "MyControl" has an event called "RightMouseButtonClick", **AXDEFGEN** names the control "@MyControl" and the event "@MyControlRightMouseButtonClick".

The "@" sign is not *required* unless ambiguities exist in the meaning in a certain context. However, to guard against unanticipated name conflicts and to ensure clarity in the reading and maintenance of the source code, we strongly recommend that you always use "@" when referring to an ActiveX property, style, or method in your source code.

If you do not use an "@" sign and a clash occurs, a compiler error results. For example, if you had a component named Editor that has a method Open, the following would cause a compiler error because Open clashes with ACUCOBOL-GT syntax:

```
...
77 hDocument HANDLE OF Editor.
...
PROCEDURE DIVISION.
MAIN.
     DISPLAY Editor LINE 1, COLUMN 1, LINES 10, SIZE 80,
          HANDLE IN hDocument.
     MODIFY hDocument Open("myfile.txt").
...
```

To address this, prepend the method name with an "@" as shown below:

```
...
77 hDocument HANDLE OF Editor.
...
PROCEDURE DIVISION.
MAIN.
     DISPLAY Editor LINE 1, COLUMN 1, LINES 10, SIZE 80,
          HANDLE IN hDocument.
     MODIFY hDocument @Open("myfile.txt").
...
```

If a class name immediately follows the level number in a Screen Section item, you must either use the "@" prefix or specify FILLER between the level number and class name.

4.7 Useful Files

We provide two files that are useful to developers involved in ActiveX and COM programming. The "activex.def" file contains useful definitions for ActiveX. We recommend that you copy it into your program's Working-Storage section.

The "acuclass.def" file contains stock class definitions for ActiveX. We recommend that you copy it into your program's Special-Names paragraph.

4.8 Multiple Object Interfaces

Some ActiveX controls are designed with multiple (object) interfaces. For example, "Microsoft Chart Control, version 6.0 (OLEDB)" has 42 public interfaces. Each interface is equivalent to a new object definition. In order to access the full feature set of the Microsoft Chart control, ACUCOBOL-GT must allow the property modification and method invocation of 42 different objects. For example, to set the Microsoft Chart legend, you get the value of the Legend property. This value is an object that you may then modify to change the legend. The Legend object has properties whose values are other objects, and so on.

Here's how you would set the text and backdrop parameters for a chart legend in Visual Basic.

```
Private Sub Command1_Click()
   With MSChart1.Legend
      ' Make Legend Visible.
      .Location.Visible = True
      .Location.LocationType = VtChLocationTypeRight
      ' Set Legend properties.
      .TextLayout.HorzAlignment = _
      VtHorizontalAlignmentRight   ' Right justify.
      ' Use Yellow text.
      .VtFont.VtColor.Set 255, 255, 0
      .Backdrop.Fill.Style = VtFillStyleBrush
      .Backdrop.Fill.Brush.Style = VtBrushStyleSolid
      .Backdrop.Fill.Brush.FillColor.Set 255, 0, 255
   End With
End Sub
```

In ACUCOBOL-GT this task is accomplished in a similar way with the USE verb, "^" and "::" operators. For example:

```
MODIFY MS-CHART-1 Legend::Location::Visible = 1.
MODIFY MS-CHART-1 Legend::Location::LocationType =
       VtChLocationTypeRight.
MODIFY MS-CHART-1 Legend::TextLayout::HorzAlignment =
       VtHorizontalAlignmentRight.
MODIFY MS-CHART-1 Legend::VtFont::VtColor::
       Set ( 255, 255, 0 ).
MODIFY MS-CHART-1 Legend::BackDrop::Fill::Style =
       VtFillStyleBrush.
MODIFY MS-CHART-1 Legend::BackDrop::Fill::Brush::Style =
       VtBrushStyleSolid.
MODIFY MS-CHART-1
       Legend::BackDrop::Fill::Brush::FillColor::
             Set ( 255, 0, 255 ).
```

or:

```
USE MS-CHART-1 Legend
MODIFY ^Location::Visible = 1
       ^Location::LocationType = VtChLocationTypeRight
       ^TextLayout::HorzAlignment =
              VtHorizontalAlignmentRight
       ^VtFont::VtColor::Set ( 255, 255, 0 )
       ^BackDrop::Fill::Style =
             VtFillStyleBrush
       ^BackDrop::Fill::Brush::Style =
             VtBrushStyleSolid
```

```
       ^BackDrop::Fill::Brush::FillColor::
             Set ( 255, 0, 255 )
END-USE
```

This syntax can be described as follows. In this format, the word following MODIFY must always be a control handle or "^". Each property or method name can be followed by "::" and then another property or method name to invoke methods inline. "MethodName1::MethodName2" means invoke the method "MethodName1" of the current object and set the current object to the return value. When a property or method name is followed by a token other than '::', then it means to actually invoke the method on the current object passing the specified arguments or set the property to the specified value and reset the current object to null.

For example, the following code:

```
MODIFY MS-CHART-1
       Legend::BackDrop::Fill::Brush::FillColor::
             Set ( 255, 0, 255 ).
```

can be broken down as follows:

```
MODIFY
MS-CHART-1
Legend
::
BackDrop
::
Fill
::
Brush
::
FillColor
::
Set
( 255, 0, 255 ).
```

which means MODIFY MS-CHART-1 in the following ways:

1. Set the current object to the chart control.
2. Invoke the "Legend" method of the current object (the chart control).
3. Release the current object.

4. Set the current object to the value returned by Legend.
5. Invoke the “BackDrop” method of the current object (the Legend object).
6. Release the current object.
7. Set the current object to the value returned by BackDrop.
8. Invoke the “Fill” method of the current object (the BackDrop object).
9. Release the current object.
10. Set the current object to the value returned by Fill.
11. Invoke the “Brush” method of the current object (the Fill object).
12. Release the current object.
13. Set the current object to the value returned by Brush.
14. Invoke the “FillColor” method of the current object (the Brush object).
15. Release the current object.
16. Set the current object to the value returned by FillColor.
17. Invoke the “Set” method of the current object (the FillColor object) passing (255, 0, 255) as arguments.
18. Release the current object.

4.9 ActiveX Library Routines

ACUCOBOL-GT includes the following six library routines related to ActiveX:

C$GETEVENTDATA	retrieves all event parameters from the control’s event procedure
C$GETEVENTPARAM	retrieves specific event parameters from the control’s event procedure

C$EXCEPINFO	retrieves information about an object exception that has been raised
C$RESOURCE	loads a control "state" resource file and either retrieves or destroys the resource handle associated with it
C$SETEVENTDATA	sets all event parameters to be sent back to the control when the control's event procedure exits
C$SETEVENTPARAM	sets specific event parameters to be sent back to the control when the control's event procedure exits

These routines are documented in Appendix I in *ACUCOBOL-GT Appendices*.

4.10 Distributing Applications Containing ActiveX Controls

To distribute your application, you need the object files and the resources they use. If you have added ActiveX controls to your application, you need to distribute files associated with the ActiveX controls along with the bitmap, ".wav", XFD, and configuration files required by your application. In addition, you need to modify your installation program to install and register the controls on the end-user's machine. Typically, you can accomplish this by copying the ActiveX control files (at least a ".ocx" or ".dll" file) to the directory where your application will be installed on the user's hard disk and calling the following command from your install shield script or batch file:

```
regsvr32 <ocx or dll name>
```

The "regsvr32.exe" file is normally located in the user's \windows\system directory. It is also included on the ACUCOBOL-GT Windows installation CD. Do not assume that it is in the user's search path or that it is pre-installed on Windows 98 and Windows NT 4.

Note: We recommend that you test your component installation by using the declaratives section to catch exceptions when creating an instance of an ActiveX control. See C$EXCEPINFO in Appendix I in *ACUCOBOL-GT Appendices* for details. In addition, note that "regsvr32**"** can be used to unregister a control as well as register it. This may be useful if you want to test whether or not your installation script is properly registering the control. When you type "regsvr32" with no command-line options, a list of available options is displayed.

For your convenience, several Microsoft controls have been included with the ACUCOBOL-GT Windows runtime. You can use these controls in your program and redistribute them to your end users as needed. The runtime checks out the license key automatically so you don't have to provide one. These controls include:

Microsoft Chart Control
Microsoft Comm Control
Microsoft DTPicker Control
Microsoft ImageCombo Control
Microsoft ImageList Control
Microsoft Internet Control
Microsoft ListView Control
Microsoft Mail Control (MapiSession, MapiMessage)
Microsoft MaskEdit Control
Microsoft Monthview Control
Microsoft Progressbar Control
Microsoft RichTextBox Control
Microsoft Slider Control
Microsoft Statusbar Control
Microsoft SystemInfo Control
Microsoft Tabstrip Control
Microsoft Toolbar Control
Microsoft TreeView Control
Microsoft UpDown Control

The ".ocx" files for these controls reside in the \ms\ocx subdirectory of the ACUCOBOL-GT installation CD for Windows. There, you will find documents called "list.txt" and "procedure.txt", which describe how to install

and register the Microsoft components onto the target machine(s). Note that for network installations, you must install the components on each workstation that will use the software (including thin client terminals). It is not sufficient to install the components on the server. For information on the controls themselves, refer to Microsoft's Controls Reference on msdn.microsoft.com.

Of course, you can use any ActiveX control in your ACUCOBOL-GT application, not just the ones that we provide on disk.

Complex controls may have more complicated installation and registration procedures. If this is the case, control vendors typically provide instructions on distributing their controls. Look for instructions in the form of "readme" files or online help files in the directory where your control is stored. You can also refer to a vendor's Web site for instructions.

When an ActiveX control requires a license, the distributor of the control provides a license key. This license key is a text string. To use a license with an ActiveX control, you set the value of the LICENSE-KEY property for the control to this license key, thereby embedding this license key in your COBOL program. Note that an ActiveX control is often delivered with two keys—one for development and one for distribution. If this is the case, the ActiveX vendor will notify you of this fact.

Once you've set the LICENSE-KEY property, when your COBOL program creates an instance of the control, the license key is passed to the ActiveX control for verification. Please note that if the license key is WideChar (WCHAR) (Doublebyte) such as "0x0067 0x01a2 0x00dd 0x0134 0x0167," you must take some additional steps to ensure that the license code is passed. If the control's license is missing or invalid, the following message displays:

```
Class is not licensed for use.
COBOL error at xxxxx in xxxxx.
```

Refer to section 5.3.2 of *ACUCOBOL-GT User Interface Programming* for more information on using the LICENSE-KEY property and passing WideChar keys.

4.11 Deployment Guidelines

Listed below are some guidelines to consider when deploying a COM object or ActiveX control:

1. Is there an end-user license?

   - Check with the vendor.
   - Make sure the LICENSE-KEY phrase is filled in with any necessary license information.
   - If you are using the Microsoft common controls included on the product CD, you do not need to worry about licensing. The end-user license is automatically checked out by the ACUCOBOL-GT runtime.
   - Licensing implications are discussed in section 4.10.

2. Are there any file dependencies?

   - What files are required (if any) to execute the control? The control vendor can tell you this. Don't ship the entire developer's installation unless you must, because this typically includes unnecessary components.
   - If external files are required, what versions should they be?
   - Does the dependent file have a dependency? If your control is made with any version of Visual Studio, it may depend on the presence of the Microsoft Visual C runtime library, or the Microsoft foundation classes of a particular version. Normally you don't have to think about this, because the COBOL Virtual Machine™ already depends on these and provides them. Make a note of it though, because your component may rely on a different version and may behave oddly in the case of a mismatch.
   - For the Microsoft common controls shipped on the product CD, we provides a text file describing the control dependencies. This file is stored in the same folder as the controls.

3. When should you install component and additional files?

- ACUCOBOL-GT's Declaratives section covers object exceptions, or cases where an ActiveX control or COM object either did not display or terminated during execution. You can trap these kinds of events in the Object Exception part of the Declaratives. You should do this to make sure that you can control the full execution of the control, and if nothing else, make a graceful termination for your application. See section 4.16 for additional details.

- With the Object Exception section in place, you can use the standard DISPLAY or CREATE verb to determine if the control you are about to use is already installed. If you have an Object Exception section and create an object that is not installed, your program is thrown into the Declaratives. Using the C$EXCEPINFO library function, you can determine the cause of the failure.

- If it is determined that the control is not installed, you should copy the files into your application directory (where you have "wrun32.exe" installed) to avoid interfering with other software and to ensure the possibility of an easy cleanup and uninstall.

- If the control is not installed and it has a ".dll" extension, you can install it from within your COBOL application. You must copy the component files to the /bin directory and then register the files via:

  ```
  DllRegisterServer
  ```

  and

  ```
  DllUnregisterServer
  ```

  Alternatively, you can use "regsvr32" like this:

  ```
  regsvr32 activexfilename
  ```

  If you want to unregister, use this:

  ```
  regsvr32 activexfilename /U
  ```

Note: If the control has the extension ".ocx", you may rename it to ".dll" in order to install it in ACUCOBOL-GT. The control's behavior is the same. The runtime does not recognize an ".ocx" as a ".dll". Once renamed, a control should not be renamed again.

For instructions on how to deploy an ActiveX control or COM object, see section 4.2.

4.12 Creating COM Objects on Remote Network Servers

ACUCOBOL-GT includes a special verb to accommodate COM objects: CREATE. The Format 1 CREATE statement creates a new instance of a COM object. (Use the Screen Section or Format 14 DISPLAY to create an instance of an ActiveX control.) Refer to Chapter 6 in the *ACUCOBOL-GT Reference Manual* for a detailed description of the CREATE statement.

If you have access privileges to do so, you can create a COM object on a remote-networked computer by passing the name of the computer in SERVER-NAME. That name is the same as the machine-name portion of a share name. For example, for a share named \\MyServer\Public, server name is MyServer. Note that CREATE cannot be used to create an object on a UNIX or VMS server. SERVER-NAME must be the name of a Windows machine.

The following code returns the version number of an instance of Excel running on a remote computer named MyServer:

```
CREATE Application of Excel
    SERVER-NAME is "MyServer"
    HANDLE in xl-app.
INQUIRE xl-app Version in xl-vers.
DISPLAY xl-vers.
```

If the remote server does not exist or is unavailable, then an exception is raised and xl-app is set to NULL.

You might think of using CREATE if you want to use an ActiveX object that does not have a user interface. However, the CREATE verb does not create ActiveX controls. If you want to create an ActiveX control that does not have a visual representation on the screen, set VISIBLE = 0 in the DISPLAY statement or Screen Section item. If you do not want any screen at all, create the initial window also with VISIBLE = 0.

The CREATE statement can be used with thin client applications to create instances of an object on the client or on remote Windows servers.

For instance, you can use the CREATE statement to provide Microsoft Office functions such as spell check and mail merge to end users. The clients must be Windows workstations, but they can be naked of all software except for Windows and the ACUCOBOL-GT Thin Client. The server running AcuConnect can be Windows, UNIX, or Linux. Note that clients must have execution privileges on the computer hosting Office, and use of the Office technology is subject to Microsoft licensing terms.

You use the Format 1 CREATE statement to create a remote instance of the desired application on the Windows computer hosting Office. When executed, your thin client application CREATEs the Office object on the Windows host, sends it instructions and data, gets results, then either displays results on the client or performs further processing on them.

Although CREATE cannot be used to create an object on a UNIX or VMS server, non-Windows servers running AcuConnect can provide connectivity to Windows servers in a multiple-tier configuration.

4.13 Qualified ActiveX Control and Object Names

In a Screen Section item, Format 14 DISPLAY statement, and a CREATE statement, “control-type-name” or “object-name” can be a qualified or unqualified name of an ActiveX control or COM object. ActiveX control and COM object names are defined in the Special-Names paragraph. The AcuBench Screen Designer or **AXDEFGEN** generates a COPY file for each ActiveX control or COM object type. You should copy this COPY file into the Special-Names paragraph. In rare cases, two different ActiveX controls or COM objects may have the same name. To use both in a single COBOL source file, you must qualify the names using “IN” or “OF” followed by the name of the “root” object. The “root” object name is defined in the ActiveX COPY file following the word OBJECT.

For example, if a single COBOL source file invokes methods in the Application object of both Microsoft Word and Microsoft Excel, it must qualify the Application object name as in the following two CREATE statements:

```
CREATE Application OF Word HANDLE IN WORD-HANDLE.
CREATE Application OF Excel HANDLE IN EXCEL-HANDLE.
```

4.14 Enumerators

The CLASS clause is also used to define enumerations. For example, the VtBrushStyle class is defined by the Microsoft Chart Control in the COPY file as follows:

```
* Brush Styles
* VtBrushStyle
     CLASS @VtBrushStyle
          CLSID, B8CC5B99-BD29-11D1-B137-0000F8753F5D
          NAME, "VtBrushStyle"
* long VtBrushStyleNull
          ENUMERATOR, @VtBrushStyleNull, 0
* long VtBrushStyleSolid
          ENUMERATOR, @VtBrushStyleSolid, 1
* long VtBrushStylePattern
          ENUMERATOR, @VtBrushStylePattern, 2
* long VtBrushStyleHatched
          ENUMERATOR, @VtBrushStyleHatched, 3
```

Enumerators are used just like level-78 data items.

Note that ActiveX enumerators should always be prepended with an "@" sign in case they clash with COBOL reserved words or ACUCOBOL-GT graphical control property and style names. The "@" character identifies the relationship of the name to ActiveX. The same holds true for ActiveX properties and methods.

4.15 ActiveX Color Representation

Many ActiveX controls use a special type named OLE_COLOR to represent colors. Methods and properties that accept a color specification of OLE_COLOR type expect that specification to be a number representing an RGB color value. For specific information regarding the OLE_COLOR type, refer to section 9.10, "ActiveX Color Settings," in *ACUCOBOL-GT User Interface Programming*.

4.16 ActiveX Error Handling

The runtime handles errors that occur during operations involving ActiveX controls similar to the way it handles errors during file I/O or during transactions. If an error cannot be naturally understood and dealt with by the COBOL program by looking at the return value or out-values of a statement, an exception is raised. The system then searches for a USE After EXCEPTION On OBJECT statement in the Declaratives section. If such a statement is found, the search stops, and the error handler is executed. If the program has not been terminated, program execution continues after the statement that raised the error. If no USE After EXCEPTION On OBJECT statement is found, the runtime determines the action. Usually, a message is presented and the program halts. Refer to Chapter 6 in *ACUCOBOL-GT Reference Manual* for more information about USE.

4.17 ActiveX Debugging

If you encounter problems when trying to run a program that contains ActiveX or COM objects, you should first try to determine whether the failure is related to the object or to the COBOL program. Try to run the ActiveX or COM object in another ActiveX container program to see if it works there. For instance, try to run it using the AcuBench Screen Designer. If you have Visual Basic on your machine, you can try to run the ActiveX or COM object in Visual Basic, or you can simply insert it on a Web page and try to run it in a browser. In addition, you can try to run the ActiveX or COM object on a different machine using ACUCOBOL-GT. If it works in other ActiveX containers or on other machines, then you have isolated the problem to your local environment or program. Check the COBOL syntax surrounding the control.

If necessary, you can use the screen trace option in the debugger to get more information about an error and to help debug your program. To enable the screen trace feature, type “ts” at the debugger command prompt. Refer to section 3.1.5, “Screen Tracing,” in *ACUCOBOL-GT User’s Guide* for more information on using this option.

4.18 ActiveX Troubleshooting

Symptom	Cause
Control does not work within AcuBench (e.g., you receive a message such as “Can’t create OCX.MSFlexGrid Control”).	Control is not properly installed or registered on the development system. Refer to section 4.2 for instructions on installing and registering controls.
Control does not appear within the **AXDEFGEN** list.	Control is not properly installed or registered on the development system. Refer to section 4.2 for instructions on installing and registering controls.
Control does not work correctly with the ACUCOBOL-GT runtime on the end-user’s machine.	Control is not properly installed or registered on the end-user’s system. Refer to section 4.10 for instructions on distributing ActiveX files along with your application.

4.19 ActiveX Examples

Use of the Windows Media Player control is demonstrated in an AcuBench sample project located in the Support area of the Micro Focus Web site. To download the project,go to: http://supportline.microfocus.com/examplesandutilities/index.asp. Select **Examples and Utilities > Graphical User Interface Sample Programs > Media_Player.zip.**

Following is an excerpt of the COPY file generated by **AXDEFGEN** for Microsoft’s Calendar Control 8.0. The COPY file includes object names so that the compiler can distinguish between two classes with the same name in different objects. For example, you might want to create an “Application of Word” and an “Application of Excel” in the same COBOL program. In the Calendar COPY file, the line “OBJECT @MSACAL” specifies the object name.

```
* CAL.DEF - ActiveX control definitions for MSACAL
* Generated: Tuesday, June 22, 1999
```

```
    OBJECT @MSACAL

* Calendar control

*** Primary Interface ***

* Calendar
     CLASS @Calendar
         CLSID, 8E27C92B-1264-101C-8A2F-040224009C02
         NAME, "Calendar"
         PRIMARY-INTERFACE
         ACTIVE-X-CONTROL
         DEFAULT-INTERFACE, "ICalendar"
         DEFAULT-SOURCE, "DCalendarEvents"
* BackColor
         PROPERTY-GET, -501, @BackColor
                 RETURNING "OLE_COLOR"
* BackColor
         PROPERTY-PUT, -501, @BackColor,
             "OLE_COLOR (Property Value)"
* Day
         PROPERTY-GET, 17, @Day
                 RETURNING "short"
* Day
         PROPERTY-PUT, 17, @Day,
             "short (Property Value)"
* DayFont
         PROPERTY-GET, 1, @DayFont
                 RETURNING "IFontDisp*"
* DayFont
         PROPERTY-PUT, 1, @DayFont,
             "IFontDisp* (Property Value)"
* DayFontColor
         PROPERTY-GET, 2, @DayFontColor
                 RETURNING "OLE_COLOR"
* DayFontColor
         PROPERTY-PUT, 2, @DayFontColor,
             "OLE_COLOR (Property Value)"
* NextDay
         METHOD, 22, @NextDay
* NextMonth
         METHOD, 23, @NextMonth
* NextWeek
         METHOD, 24, @NextWeek
* NextYear
```

```
        METHOD, 25, @NextYear
* PreviousDay
        METHOD, 26, @PreviousDay
* PreviousMonth
        METHOD, 27, @PreviousMonth
* PreviousWeek
        METHOD, 28, @PreviousWeek
* PreviousYear
        METHOD, 29, @PreviousYear
* Refresh
        METHOD, -550, @Refresh
* Today
        METHOD, 30, @Today
* AboutBox
        METHOD, -552, @AboutBox
* Click
        EVENT, -600, @CalendarClick
*           No Parameters
* DblClick
        EVENT, -601, @CalendarDblClick
*           No Parameters
* KeyDown
        EVENT, -602, @CalendarKeyDown
*           2 Parameters
                short* KeyCode
                short Shift
* KeyPress
        EVENT, -603, @CalendarKeyPress
*           1 Parameter
*               short* KeyAscii
* KeyUp
        EVENT, -604, @CalendarKeyUp
*           2 Parameters
*               short* KeyCode
*               short Shift
* BeforeUpdate
        EVENT, 2, @CalendarBeforeUpdate
*           1 Parameter
*               short* Cancel
* AfterUpdate
        EVENT, 1, @CalendarAfterUpdate
*           No Parameters
* NewMonth
        EVENT, 3, @CalendarNewMonth
*           No Parameters
```

```
* NewYear
        EVENT, 4, @CalendarNewYear
*           No Parameters
```

After generating the COPY file, you may create and use a Calendar control. You can add the control to your Screen Section as described in section 4.2, or you can use the DISPLAY statement in the Procedure DivisioN to create the control. Note that the following two statements are equivalent:

```
    DISPLAY Calendar HANDLE IN my-cal.
    DISPLAY Calendar OF MSACAL HANDLE IN my-cal.
```

Here is an example of how you might update your Procedure Division to create, and then modify, the control:

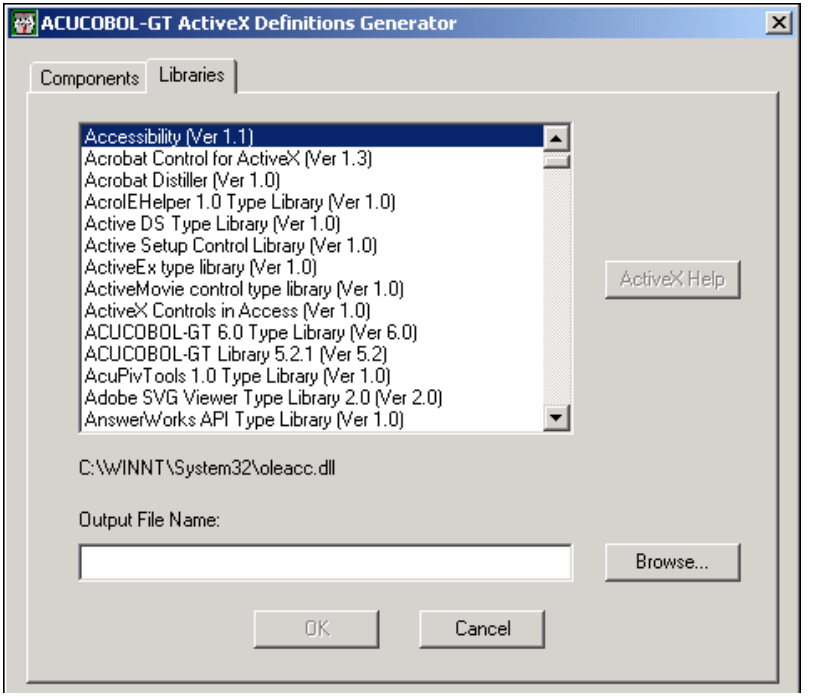
```
* Declare the calendar control's handle.
77 CALENDAR-1 USAGE IS HANDLE OF Calendar.
...
* Declare the @KeyPress event parameter
77 KEY-ASCII PIC X USAGE COMP-X.
77 KEY-CHAR PIC X REDEFINES KEY-ASCII.
...
* Calendar @Value is the current date
01 DATE-1
    03 MONTH PIC 99.
    03 FILLER PIC X VALUE '/'.
    03 DAY PIC 99.
    03 FILLER PIC X VALUE '/'.
    03 YEAR PIC 9999.
...
* Create an instance of the calendar control
       DISPLAY Calendar LINE 4 COLUMN 6 LINES 10 SIZE 40
               EVENT PROCEDURE IS CALENDAR-EVENT-HANDLER
               HANDLE IN CALENDAR-1.
...
* Set the calendar's "day" font to Courier 10 pt Bold
       USE CALENDAR-1 DayFont
           MODIFY Name = "Courier" Size = 10 Bold = 1.
...
* Set the calendar’s year to 2000
       MODIFY CALENDAR-1 Year = 2000.
...
* Invoke the NextDay method
       MODIFY CALENDAR-1 NextDay.
...
```

```
* Invoke the PreviousDay method
       MODIFY CALENDAR-1 PreviousDay.
...
* Handle events
...
CALENDAR-EVENT-HANDLER.
       EVALUATE EVENT-TYPE
           WHEN MSG-AX-EVENT
               EVALUATE EVENT-DATA-2
                   WHEN CalendarBeforeUpdate
* Don't allow years >= 2000
                       INQUIRE EVENT-CONTROL-HANDLE
                          @Value IN DATE-1
                       IF YEAR OF DATE-1 >= 2000
* Cancel the update (set the 'Cancel' parameter to 1)
                           CALL "C$SETEVENTDATA" USING
                           EVENT-CONTROL-HANDLE, 1
                       END-IF
                   WHEN CalendarKeyPress
* Stop run if the user presses 'X'
                       CALL "C$GETEVENTDATA" USING
                       EVENT-CONTROL-HANDLE, KEY-ASCII
                   IF KEY-CHAR = 'X' STOP RUN END-IF
...
```

4.20 AXDEFGEN Utility Reference

The ActiveX Definitions Generator (**AXDEFGEN**) utility is a dialog-based application designed to facilitate the addition of ActiveX controls and COM objects to Windows-based ACUCOBOL-GT programs. As mentioned in section 3.4.1, the role of **AXDEFGEN** is to locate the names of ActiveX controls and COM objects currently registered on the system and generate a COBOL COPY file for the control or object that you select. This COPY file is used by the ACUCOBOL-GT compiler for syntax and parameter type checking as well as efficient code generation.

AXDEFGEN is located in the \AcuGT\bin directory wherever you installed ACUCOBOL-GT on your machine. If you run **AXDEFGEN** without command-line parameters, a dialog box appears.



The dialog box has two tabs: Components and Libraries. The Components tab lists all ActiveX controls and other COM objects that have registered type libraries. The items on this list are derived from the HKEY_CLASSES_ROOT\CLSID registry key and then matched by the Globally Unique IDentifier (GUID) of the type library against the HKEY_CLASSES_ROOT\TypeLib registry entries. The Libraries tab contains a list of libraries derived directly from the HKEY_CLASSES_ROOT\TypeLib registry key. Therefore, some items may appear under both tabs, and some items in the Libraries list may represent collections of items from the Components list. **AXDEFGEN** generates a COPY file for the entire type library, whether you select the library name from the Libraries list or the name of an individual component from the Components list.

Select the name of an ActiveX control or COM object that you want to include in your COBOL program, then browse to choose an output path and filename for the COPY file. (The extension ".def" is automatically appended to the filename you specify.) When done, click **OK** to generate the COPY file.

Note: To prevent compiler errors, **AXDEFGEN** automatically converts any embedded spaces that it finds to hyphens when generating the COPY file, including spaces found in object methods, events, properties, parameters, and enumerators.

Many ActiveX controls and COM objects have documentation available. If they do, the ActiveX Help push button on the right side of this box is enabled. Click the button to read the help file for the selected ActiveX control or COM object. If the ActiveX Help push button is disabled, it means that **AXDEFGEN** could not locate a help file for the selected control.

If desired, you can execute AXDEFGEN from the command line. It takes two optional command line parameters. The first is the registry name of an ActiveX control or other COM object. The second is the name of the COPY file you want AXDEFGEN to create. For example:

```
AXDEFGEN MSCAL.Calendar cal.def
```

creates a COPY file named "cal.def" in the current directory containing the definitions for the Microsoft Calendar Control, which is an ActiveX control registered as "MSCAL.Calendar" in the system registry. Likewise:

```
AXDEFGEN Word.Application word.def
```

creates a COPY file named "word.def" in the current directory containing the definitions for Microsoft Word, which is registered as Word.Application in the system registry.

For more information on ActiveX and COM programming, including instructions on what to do with the COPY files generated by **AXDEFGEN**, refer to section 4.2, "Adding ActiveX Controls or COM Objects to Your COBOL Program."

4.20.1 AXDEFGEN COPY Files

Following is a list of TYPE codes that you might find in a COPY file generated by **AXDEFGEN**. The TYPE names are acronyms and/or abbreviations for the type descriptions. VT stands for Variant Type. For example, VT_I2 is a type of variant that contains a 2-byte signed integer. VT_UI2 is a 2-byte unsigned integer. VT_R4 is a 4-byte real number (floating point). For a more complete reference of these codes, refer to the Microsoft Developer's Network at msdn.microsoft.com.

Note that you may pass an alphanumeric data item or literal as a parameter to a property or method that expects a numeric item. The runtime automatically "parses" the alphanumeric string and extracts a number from it if possible.

In general, if a parameter is passed by reference (usually an I/O or output parameter), then it has 16384 (hex 4000) added to its value in the COPY file. For example a boolean output parameter would be 16384 + 11 (the value for VT-BOOL), or 16395. For this reason, if the TYPE in the COPY file is between 16384 and 16456, then you can subtract 16384 to find the associated TYPE code.

This table also includes possible C types that you may find in **AXDEFGEN** COPY files along with the corresponding COBOL data class.

AXDEFGEN COPY File Type Codes

TYPE Code	TYPE Name	C Type	COBOL Data Class	Description of Data
	VT_EMPTY	void	N/A	Nothing
1	VT_NULL	null	numeric	SQL style Null; any numeric data item, 0, zero, null, or low-values
2	VT_I2	short	numeric	2-byte signed int; a 16-bit signed integer, any numeric data item or literal

AXDEFGEN COPY File Type Codes

TYPE Code	TYPE Name	C Type	COBOL Data Class	Description of Data
3	VT_I4	int	numeric	4-byte signed int; a 32-bit signed integer, any numeric data item or literal
4	VT_R4	single	numeric	4-byte real; a single (4-byte) floating point number, any numeric data item or literal, typically USAGE FLOAT
5	VT_R8	double	numeric	8-byte real; a double (8-byte) floating point number, any numeric data item or literal, typically USAGE DOUBLE
6	VT_CY	CURRENCY	numeric	Currency; a currency value, any numeric data item or literal, usually containing a decimal point
7	VT_DATE	DATE	alphanumeric	Date; a date in either numeric or alphanumeric form
8	VT_BSTR	BSTR	alphanumeric	COM Automation string; a character string, any data item or literal, typically USAGE DISPLAY
9	VT_DISPATCH	IDispatch	numeric	IDispatch; a pointer to the IDispatch interface, a USAGE POINTER, or USAGE HANDLE item
10	VT_ERROR	SCODE	numeric	SCODE; a 32-bit unsigned integer, any numeric data item or literal

AXDEFGEN COPY File Type Codes

TYPE Code	TYPE Name	C Type	COBOL Data Class	Description of Data
11	VT_BOOL	boolean	numeric	True=-1, False=0; any alphanumeric or numeric data item or literal, pass 1, -1 or “True” for True, 0 or “False” for False
12	VT_VARIANT	VARIANT	any	VARIANT; any alphanumeric or numeric data item or literal
13	VT_UNKNOWN	IUnknown	numeric	IUnknown; a pointer to the IUnknown interface, a USAGE POINTER, or USAGE HANDLE item
14	VT_DECIMAL	N/A	numeric	16-byte fixed point
16	VT_I1	char	alphanumeric	Signed char; a single 8-bit signed character, any data item or literal, typically USAGE DISPLAY
17	VT_UI1	unsigned char	numeric	Unsigned char; a single 8-bit unsigned character, any numeric data item or literal
18	VT_UI2	unsigned short	numeric	Unsigned short; a 16-bit unsigned integer, any numeric data item or literal
19	VT_UI4	unsigned long	numeric	Unsigned long; a 32-bit unsigned integer, any numeric data item or literal
20	VT_I8	int64	numeric	Signed 64-bit int; a 64-bit signed integer, any numeric data item or literal

AXDEFGEN COPY File Type Codes

TYPE Code	TYPE Name	C Type	COBOL Data Class	Description of Data
21	VT_UI8	uint64	numeric	Unsigned 64-bit int; a 64-bit unsigned integer, any numeric data item or literal
22	VT_INT	int	numeric	Signed machine int; a 32-bit signed integer, any numeric data item or literal
23	VT_UINT	unsigned int	numeric	Unsigned machine int; a 32-bit unsigned integer, any numeric data item or literal
24	VT_VOID	void	N/A	C style void
25	VT_HRESULT	HRESULT	numeric	Standard return type; a 32-bit unsigned integer, any numeric data item or literal
26	VT_PTR	PTR	numeric	Pointer type; a USAGE HANDLE or USAGE POINTER data item
27	VT_SAFEARRAY	SAFEARRAY	table	One- or two-dimensional table with one USAGE HANDLE or USAGE HANDLE OF VARIANT elementary item
28	VT_CARRAY	CARRAY	N/A	C style array
29	VT_USERDEFINED	USERDEFINED	N/A	User-defined type
30	VT_LPSTR	LPSTR	alphanumeric	Null terminated string; any alphanumeric data item or literal
31	VT_LPWSTR	LPWSTR	alphanumeric	Wide null terminated string; any alphanumeric data item or literal
36	VT_RECORD		N/A	User-defined type

AXDEFGEN COPY File Type Codes

TYPE Code	TYPE Name	C Type	COBOL Data Class	Description of Data
64	VT_FILETIME	FILETIME	N/A	FILETIME
65	VT_BLOB	BLOB	N/A	Length prefixed bytes
66	VT_STREAM	STREAM	N/A	Name of the stream follows.
67	VT_STORAGE	STORAGE	N/A	Name of the storage follows.
68	VT_STREAMED _OBJECT	STREAMED _OBJECT	N/A	Stream contains an object.
69	VT_STORED _OBJECT	STORED _OBJECT	N/A	Storage contains an object.
70	VT_BLOB_OBJECT	BLOB_OBJECT	N/A	Blob contains an object.
71	VT_CF	CF	N/A	Clipboard format
72	VT_CLSID	CLSID	alphanumeric	A Class ID; any alphanumeric data item or literal

If DATE is in numeric form, days are represented by whole number increments starting with 30 December 1899, midnight as time zero. Hour values are expressed as the absolute value of the fractional part of the number. For example:

0.00 is 30 December 1899, 12:00 A.M.

5.25 is 4 January 1900, 6 A.M.

5.875 is 4 January 1900, 9 P.M.

If DATE is in alphanumeric form, the date can be in a variety of formats. For example, the following are all valid formats:

“25 January 1996”

“8:30:00”

"20:30:00"

"January 25, 1996 8:30:00"

"8:30:00 Jan. 25, 1996"

"1/25/1996 8:30:00"

You may pass a USAGE POINTER item that was filled in by a prior method or property call, or you may pass a USAGE HANDLE item which contains a handle to an ActiveX control or COM object. You may not pass a Screen Section item name as a handle of an ActiveX control. Instead, use the Format 11 SET verb to get a handle to the ActiveX control (e.g., SET my-handle to HANDLE OF screen-section-item). The IUnknown interface is part of the Microsoft COM standard. Any COM object or ActiveX control exports interfaces that are used to create, use, and destroy objects. Each interface is based on a single interface called IUnknown. This means that you may pass a pointer to any of these interfaces (objects) to a method or property that expects an IUnknown*.

The runtime automatically converts a handle to an ActiveX control or COM object to the IUnknown* type when you pass it to a method or property.

Any of these types may be followed by an asterisk to indicate that the parameter will be passed "by reference". This means that the ActiveX control or COM object method or property may modify the contents of the passed data item.

Any type name other than those in the list is a user-defined type. User-defined types are those that are created by the ActiveX control or COM object programmer or vendor. They always resolve to one of the types in the list but have different names to indicate their functions. For example, OLE_COLOR is a user-defined type that is commonly used to represent colors in ActiveX controls and COM objects. It resolves to a unsigned long which is a 32-bit unsigned integer. You must read the programmer's documentation of the particular ActiveX control or COM object in order to determine how to use user-defined types. For example, after reading about OLE_COLOR you may learn a formula to allow you to construct an OLE_COLOR if you know the red (0-255), green (0-255), and blue (0-255) components of the color you are trying to represent.

Another common user-defined type is IFontDisp*. This type is used to represent fonts. Some ActiveX controls and/or COM objects have properties whose values are fonts, or methods whose parameters are fonts. You may use INQUIRE to get a IFontDisp* into a HANDLE OF IFontDisp item. IFontDisp is defined in "acuclass.def". Then you may modify the NAME, SIZE, BOLD, ITALIC, UNDERLINE, STRIKETHROUGH, WEIGHT, or CHARSET properties of the IFontDisp item using the MODIFY verb. For example:

```
copy "acuclass.def"
...
77  my-font-disp  usage handle of IFontDisp
...
INQUIRE Calendar-1 DayFont in my-font-disp.
MODIFY my-font-disp Name = "Courier New"
     @Size = 15, Bold = 1.
```

Note: Because SIZE is a common ACUCOBOL-GT property name, @Size uses the "at" sign to distinguish it as an ActiveX property name.

Alternatively, you may use the double colon ("::") operator to set these properties in a single MODIFY statement. For example:

```
MODIFY Calendar-1 DayFont::Name = "Courier New"
    DayFont::Size = 15, DayFont::Bold = 1.
```

In this case, you do not use a HANDLE OF IFontDisp item. Instead, the runtime creates a temporary HANDLE OF IFontDisp item, does the INQUIRE, and sets the properties "behind the scenes" in the processing of the MODIFY statement.

Another user-defined type that you might see is DataSource*, which is sometimes used as the value of a DataSource property in an ActiveX control. It resolves to the IUnknown * type. For example, to use the Microsoft DataGrid control and the Microsoft ADO control together, set the DataGrid control's DataSource property to the IUnknown* of the ADO control. As stated above, to pass an ActiveX control or COM object as a IUnknown* you must pass the handle of the control or COM object.

For example:

```
01  main-screen.
    03 adoctrl, Adodc,
```

```
       COL 14 LINE 21 LINES 2.20 CELLS
       SIZE 29.00 CELLS
       LICENSE-KEY "C4145310-469C-11d1-B182-00A0C922E820".
    03 testgrid, Datagrid,
       COL 14 LINE 9 LINES 10 CELLS
       SIZE 28 CELLS
       LICENSE-KEY "CDE57A55-8B86-11D0-b3C6-00A0C90AEA82".

    03 PUSH-BUTTON LINE 27 COL 23 TITLE "Exit"
       CANCEL-BUTTON LINES 4 CELLS SIZE 10 CELLS.

...

DISPLAY main-screen.

MODIFY adoctrl ConnectionString = "DSN=Customers".
MODIFY adoctrl DatasourceName = "Customers".
MODIFY adoctrl RecordSource = "Select * from publishers".

MODIFY testgrid Caption = "Test".
SET adoctrl-handle TO HANDLE OF adoctrl.
MODIFY testgrid DataSource = adoctrl-handle.

DISPLAY testgrid.
```

The handle of the “adoctrl” is obtained with the SET verb. It is then passed as the value of the DataSource property in the MODIFY statement that follows.

5 Working With .NET Assemblies

Key Topics

5.1 COBOL and .NET

ACUCOBOL-GT includes many facilities for interoperating with Microsoft® .NET technologies.

It includes compiler options that generate Microsoft Intermediate Language (MSIL) objects capable of calling and running your COBOL program. These objects can be managed and run by the .NET Common Language Runtime (CLR) and instantiated by a .NET assembly.

It includes a .NET interface that lets .NET programmers interact with the COBOL program at the API level. A .NET version of the ACUCOBOL-GT® programs can be included in any Visual Studio .NET project.

ACUCOBOL-GT also includes a utility, NETDEFGEN, that translates .NET assemblies into COBOL COPY files, making it easy to invoke .NET assemblies from your COBOL program.

Web services are supported via .NET client control proxies. These proxies are simple .NET controls that interact with Web services and communicate with an ACUCOBOL-GT program using native .NET event declarations, methods, and properties.

UNIX users can interact with .NET assemblies using our Thin Client technology. If you want, you can mix Win32, ActiveX, and .NET graphical controls on the same ACUCOBOL-GT screen.

All of these options are discussed in this chapter.

5.2 What Is .NET?

.NET is a set of Microsoft software technologies designed to facilitate the development and execution of large, interoperable Web-based, desktop, distributed, and server applications, both for e-commerce and global electronic businesses. .NET is the Microsoft solution for Web services. Like other Web service environments, it relies heavily on eXtensible Markup Language (XML) and SOAP.

.NET has been incorporated across Microsoft's clients, servers, services, and tools. For developers, .NET is manifested in the programming model delivered in the Microsoft .NET Framework. .NET is based on the reuse of services. Microsoft defines *services* as small, discrete, building-block applications that connect to each other as well as to other, larger applications via the Internet or an intranet.

.NET supports many different programming languages. Normally .NET applications are compiled into Microsoft intermediate language (MSIL) and run in a sort of virtual operating system, the CLR. The compiled code is called *managed* code, because it is managed by the CLR.

Applications compiled in other languages can also run in .NET environments under the Windows operating system, and Microsoft has provided many facilities to allow such programs to interoperate with managed code.

5.3 What Is an Assembly?

In .NET parlance, all controls or programs are called *assemblies*. An assembly is a single file or a group of files that comprise a program, including the security management, versioning, sharing, and deployment information as well as the individual services that the program utilizes.

Services perform discrete functions ranging anywhere from simple requests to complicated business processes that combine information from multiple sources. A .NET assembly is a group of services that can be *assembled* like building blocks into a cohesive business application, or it can be as simple as an individual program that performs some discrete functions.

An assembly may appear as a single ".dll" or ".exe" file. Assemblies can be listed in a Global Assembly Cache (GAC) on an end user's machine.

5.4 Calling COBOL from .NET

There are three ways to call and interact with an ACUCOBOL-GT program from a .NET assembly. You can:

- Use ACUCOBOL-GT's .NET compiler options to package your ACUCOBOL-GT program as a .NET assembly. .NET programmers can then invoke these objects as they would any other .NET assembly.

  Though perhaps easiest, this option limits data passing as discussed in section 5.4.1.

  Note: To use the .NET compiler options, you must have Microsoft .NET Development Framework Version 2.0. However, you can create Version 1.1 and 2.0 assemblies.

- Use our .NET interface assembly, "wrunnet.dl". This gives .NET programmers more direct access to and control of the ACUCOBOL-GT Windows runtime module. "wrunnet.dll" contains a singleton class, CVM, that encapsulates the ACUCOBOL-GT runtime. With the CVM, the .NET programmer can programmatically instantiate the ACUCOBOL-GT runtime and invoke a COBOL program without a COM interface or knowledge of the Windows API or .NET PINVOKE.

  .NET programmers can include "wrunnet.dll" in a Visual Studio .NET project and take advantage of the native .NET development environment. This interface gives them lower-level control over the COBOL program, and is very similar to the ACUCOBOL-GT C and Java interfaces.

- Use ACUCOBOL-GT's COM server technology. The COM server is a COM object containing the ACUCOBOL-GT Windows runtime DLL. When the COM server is added to a .NET project, a proxy is created to communicate with the ACUCOBOL-GT runtime. The proxy provides all interface and data marshalling between .NET and the COM server. (Marshalling is the process of gathering data and transforming it into a standard format before transmitting it over a network.)

5.4.1 Using the .NET MSIL Compiler Options

The easiest way to provide COBOL services to a .NET assembly is to use one of ACUCOBOL-GT's .NET compiler options to generate managed objects that can call your ACUCOBOL-GT program. These objects are compiled in Microsoft Intermediate Language (MSIL), so they can be managed directly

by the .NET Common Language Runtime (CLR). They are MSIL stubs capable of calling the ACUCOBOL-GT runtime and executing your ".acu" object.

There are two compiler options that you can use for this purpose:

Compiler Option	Description
--netexe	Generates a .NET executable file from your COBOL source. By default this command generates a .NET Version 1.1-compatible assembly or the latest version if 1.1 is not installed. You can also specify the desired version by adding the appropriate qualifier to the option. For example: "--netexe:2.0" verifies that .NET Version 2.0 is installed and generates a 2.0-compatible assembly if so. Valid versions are 1.1 and 2.0.
--netdll	Generates a .NET dynamic link library (DLL) file from your COBOL source. By default this command generates a .NET Version 1.1-compatible assembly or the latest version if 1.1 is not installed. You can also specify the desired version by adding the appropriate qualifier to the option. For example: "--netdll:2.0" verifies that .NET Version 2.0 is installed and generates a 2.0-compatible assembly if so. Valid versions are 1.1 and 2.0.

5.4.1.1 –netexe

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

-int:
-string:

-uint:
-short:
-ushort:
-float:
-double:
-long:
-ulong:
-byte:

For example, the compiler command:

```
ccbl32 --netexe MyCobol.cbl
```

results in two files being created: the COBOL object, “MyCobol.acu,” and the .NET executable, “MyCobol.exe.”

You can execute the .NET version from the command line with “MyCobol.exe” or you can include command line options and linkage section parameters, as in:

```
MyCobol.exe -d -int:1234 -string:"Enter customer name"
```

5.4.1.2 –netdll

The “--netdll” compiler option generates a .NET dynamic link library (DLL) that gives .NET assemblies—both executables and DLLs—a programmatic interface to your COBOL program. All COBOL entry points are exposed as .NET methods along with ACUCOBOL-GT runtime properties and methods. This allows .NET programmers to set ACUCOBOL-GT command options and call runtime interfaces from their .NET assembly.

By referring to an ACUCOBOL-GT .NET DLL in a project solution, .NET programmers can view ACUCOBOL-GT runtime properties, runtime initialization and control methods, COBOL entry points, the main COBOL entry point, and Linkage Section parameters in the Visual Studio .NET object browser.

Three files are created when you compile a COBOL program using the “--netdll“ option. All three begin with the program file name. For example, the compiler command:

```
ccbl32 --netdll MyProgram.cbl
```

results in “MyProgram.dll,” “MyProgram.netmodule,” and “MyProgram_CVM.dll.” .NET programmers would reference “MyProgram.dll” and “MyProgram_CVM.dll” in their project.

“MyProgram_CVM.dll” contains all the COBOL program entry points and ACUCOBOL-GT runtime interfaces exposed as .NET methods. ACUCOBOL-GT runtime options are exposed as properties.

The class in “MyProgram.dll” derives from class CVM which resides in “MyProgram_CVM.dll” allowing the instantiation of namespace class “MyProgram.MyProgram myPgm = new MyProgram.MyProgram()”. All methods and properties in “MyProgram_CVM.dll” class CVM are exposed to object “myPgm”.

“MyProgram.netmodule” contains ACUCOBOL-GT setup routines that are automatically executed during object instantiation. “MyProgram.dll” works in conjuction with “MyProgram.netmodule” to perform this task. The namespace and class are always generated using the COBOL file name without the extension. In this case the namespace and class are “MyProgram.MyProgram”.

As mentioned previously, COBOL entry names and the main COBOL program entry point generate .NET methods. Method parameters are generated when entry statements contain USING parameters or a Procedure Division statement contains USING parameters. There are two additional parameters added to each generated .NET method. They follow all the COBOL USING parameters for the entry name or Procedure Division statement. The first parameter, string, is for program execution command parameters. They are “-d” (debug) and “-cache”. All other runtime command options must be set via properties before calling AcuInitialize or in the string parameter of AcuInitialize.

The second parameter is for a return code. This is the return code from the COBOL program. The method return code is from the COBOL Virtual Machine interface and is documented in section 5.4.2.1. You can also view the values using the Visual Studio .NET object browser under ErrorTypes in class CVM.

Note: All of the criteria that apply to the CVM class also apply to the .NET component. Refer to section 5.4.2.1 for details on these criteria.

5.4.1.3 Data passing limitations

The compiler options limit data passing to the following data types: integer, string, unsigned integer, short, unsigned short, float, double, long, unsigned long, and byte.

5.4.1.4 Example

COBOL Source

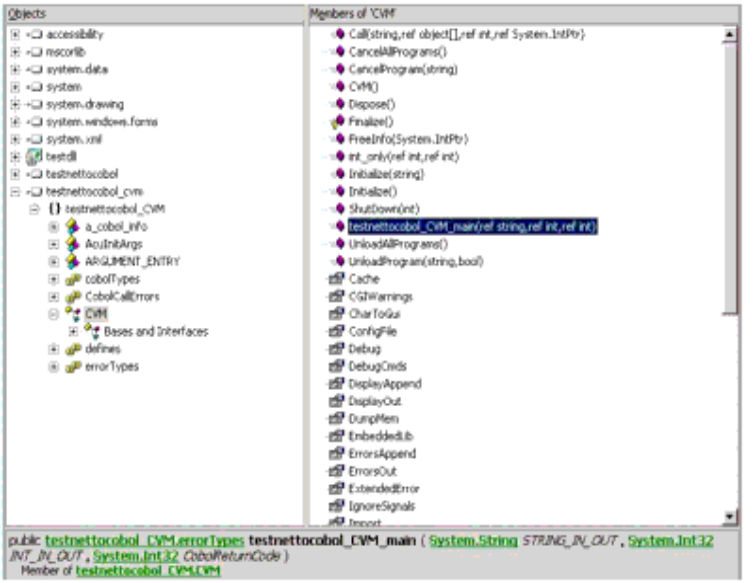
Following is the COBOL source for a sample program, "TestNetToCobol.cbl".

```
identification division.
program-id. TestNetToCobol.
environment division.
configuration section.
data division.
working-storage section.

linkage section.
77 string-in-out          pic x(32) value spaces.
77 int-in-out             USAGE IS SIGNED-INT.

procedure division using string-in-out, int-in-out.
main-logic.

     move "hey whats doin" to string-in-out.
     entry "int-only" using int-in-out.
     move 9999 to int-in-out.
     exit program.
```

View of Managed Code in Visual Studio .NET Object Browser

Following is a screen that shows what "TestNetToCobol_CVM" looks like to a .NET programmer in the Visual Studio .NET object browser (once you've generated a .NET DLL for this program with the "--netdll" option). Notice

that there are methods other than the COBOL program entry points and main Procedure Division paragraph in this browser. These ACUCOBOL-GT runtime interfaces are documented in section 3.5.3.2.



C# Source

Here is a C# program that makes reference to “TestNetToCobol_CVM”.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using testnettocobol;
//using testnettocobol_CVM;

namespace TestDll
{
     /// <summary>
     /// Summary description for Form1.
     /// </summary>
     public class Form1 : System.Windows.Forms.Form
     {
          testnettocobol.testnettocobol cblObj;
```

```
private System.Windows.Forms.Button button1;
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1()
{
     //
     // Required for Windows Form Designer support
     //
     InitializeComponent();

     //
     // TODO: Add any constructor code after InitializeComponent call
     //
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
     if( disposing )
     {
          if (components != null)
          {
               components.Dispose();
          }
     }
     base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
     this.button1 = new System.Windows.Forms.Button();
     this.SuspendLayout();
     //
     // button1
     //
     this.button1.Location = new System.Drawing.Point(104, 24);
     this.button1.Name = "button1";
     this.button1.TabIndex = 0;
     this.button1.Text = "Test";
```

```
     this.button1.Click += new System.EventHandler(this.button1_Click);
     //
     // Form1
     //
     this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
     this.ClientSize = new System.Drawing.Size(292, 101);
     this.Controls.Add(this.button1);
     this.Name = "Form1";
     this.Text = "Form1";
     this.ResumeLayout(false);

}
#endregion

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
     Application.Run(new Form1());
}

private void button1_Click(object sender, System.EventArgs e)
{
    testnettocobol_CVM.errorTypes err = testnettocobol_CVM.errorTypes.CS_OK;
     IntPtr pInfo = IntPtr.Zero;
     int cblReturn = 0;
     string err_msg;
     int some_int = 777;
     string some_string;

     // Instantiate COBOL object and CVM, COBOL VIRTUAL MACHINE
     cblObj = new testnettocobol.testnettocobol();
     // Set ACUCOBOL-GT runtime options (properties)
     cblObj.RunPath = "D:\\branch_7_1\\cobolgt\\bin";
     cblObj.Debug = true;
     cblObj.LinkageLength = true;
     //cblObj.Cache = true;
     // Initialize the ACUCOBOL-GT runtime
     cblObj.Initialize();

     try
     {

          //  The second to last parameter is for options specific
          //  to a method call.  They may also be set via properties
          //  before the method call is executed.
          //  The last parameter is the return code from the COBOL program.
```

```
            // call an ENTRY in the ACUCOBOL-GT program
            some_int = 1111;
            cblReturn = 0;

            err = cblObj.int_only(ref some_int, null, ref cblReturn);

            // call the main ENTRY, 1st COBOL line in PROCEDURE DIVISION
            some_int = 1422;
            some_string = "The hills are Alive             ";

            err = cblObj.testnettocobol_CVM_main(ref some_string,
                                    ref some_int,
                                    null,
                                    ref cblReturn);
        }
        catch (System.Exception e2)
        {
            Exception innerE = e2.InnerException;
            if ((innerE != null) && (innerE.Message.Length > 0))
                err_msg = innerE.Message;
            else
            {
                if (e2.Message.Length > 0)
                    err_msg = e2.Message;
                else
                    err_msg = "AcuNet Temp Object Create Error";
            }
            MessageBox.Show(err_msg);
            return;
        }
        // test runtime return code
        if (err != testnettocobol_CVM.errorTypes.CS_OK)
        {
            // get error text property
            MessageBox.Show(cblObj.LastErrorMsg);
        }
        cblObj.ShutDown(0);
    }
}

     }
```

5.4.2 Using the .NET Interface Assembly, "wrunnet.dll"

A more precise way to call COBOL from .NET is using the application programming interface (API) contained in the dynamic link library, "wrunnet.dll". .NET developers can use this interface to call COBOL functionality (programs, entry points, etc.) from their .NET class.

The API consists of one .NET class: CVM. CVM class allows .NET developers to programmatically manage the ACUCOBOL-GT runtime, giving them low-level control of COBOL objects from .NET.

Note that you can call COBOL from .NET locally or remotely. You can even have the runtime execute remotely without a COBOL object executing on the client. All you need on the client is a .NET program and a runtime. For this to work, you set CODE_PREFIX in the configuration file that you provide with the runtime initialization to point to a remote server hosting your COBOL application. The remote server must also be running AcuConnect. AcuConnect is able to execute a COBOL object remotely and share data with the local runtime. For more information on executing remote COBOL programs with AcuConnect, please refer to the *AcuConnect User's Guide.*

5.4.2.1 CVM class

CVM is a .NET class representing the ACUCOBOL-GT runtime. The CVM class exposes public methods for setting runtime options, calling and cancelling programs, getting object libraries, and more.

CVM contains the following methods within the name space "acucobol":

- Initialize
- Call
- CancelProgram
- CancelAllPrograms
- UnloadProgram
- UnloadAllPrograms

- ShutDown

Each of these methods corresponds to the runtime interfaces *acu_initv*, *acu_cobol*, *acu_cancel*, *acu_unload*, and *acu_shutdown*. More details are provided in the following sections.

Initialize

Initializes the COBOL Virtual Machine (runtime) with command line options before calling CallAcuCobol. Initialize is optional; Call will call Initialize with defaults if it has not been previously called.

Usage:

```
public unsafe bool Initialize (string cmdline)
public bool Initialize ()
```

Where:

Variable	Definition
cmdline	Includes ACUCOBOL-GT runtime options. You can set runtime options using key value pairs or by setting properties. See “Properties” below.

Use “public bool Initialize ()” to set defaults.

Call

Executes an ACUCOBOL-GT program.

Usage:

```
public unsafe errorTypes Call (string        pgmName,
                               ref object[]  CobolParams,
                               ref byte[]    CobolTypes,
                               string        CallOptions,
                              ref int       ProgramReturnCode)
```

Where:

Variable	Definition
pgmName	Is the file path of an ACUCOBOL-GT program.
CobolParams	Is an array of parameter objects. They must match the Procedure Division USING parameters and be native types int, uint, short, ushort, long, ulong, float, double, char, byte, and/or string. Parameters should correspond one-to-one with the Linkage Section of the COBOL program.
CobolTypes	Is an array of COBOL types. (See "CompilerTypes" later in this section.) This field is optional, meaning it can be a null reference. When used, it must be allocated with the same number of entries as CobolParams, one type corresponding to each parameter. These entries are needed when a COBOL program has a mix of unicode, double byte, and ANSI strings. .NET treats all strings as unicode and "wrunnet", by default, converts them to ANSI strings. In order for "wrunnet" to convert the unicode to a wide character, double byte, or pass it as a unicode string to a COBOL program, the corresponding entry in CobolTypes must be set. If all strings in the COBOL program are unicode or all are double byte, a property may be set. This field is automatically generated when using the compiler option, "--netdll". The applicable fields are NAT, NATJ, NATE, WID, WIDJ and EWID.

Variable	Definition
CallOptions	The following runtime options can be set using the Call method: • "-d", debug • "-show", display error text • " -uni", unicode • "-wide", double byte characters • "-cache", cache program These are the only options that can be set after Initialize is called. They may also be set using a property assignment before a Call is executed. For example: "-d 1" or "-d 0" turns debug mode on in the first instance and off in the second instance. This is the only place where a boolean option can be set on and off in this manner. All other boolean options set via Initialize can be turned only on. In this case the syntax does not include a "1" or "0".
ProgramReturnCode	Upon return from Call, ProgramReturnCode contains the COBOL program return code.

CancelProgram/CancelAllPrograms

Cancels program(s) and resets working storage. Optional.

Usage:

```
public unsafe bool CancelProgram (string name)
public unsafe bool CancelAllPrograms ()
```

Where:

Variable	Definition
name	The COBOL program to cancel.

"public unsafe bool CancelAllPrograms ()" cancels all programs.

UnloadProgram/UnloadAllPrograms

Unloads a cached program from memory. Optional.

Usage:

```
public unsafe bool UnloadProgram (string name,  bool subprograms)
public unsafe bool UnloadAllPrograms ()
```

Where:

Variable	Definition
name	The COBOL program to unload.
subprograms	When true, unloads all subprograms

"public unsafe bool UnloadAllPrograms ()" unloads all cached programs from memory.

ShutDown

ShutDown terminates the ACUCOBOL-GT runtime and is sometimes called when the runtime is no longer needed by the application . Please note that ShutDown should rarely be executed in a .NET application. This is because CVM provides an interface to the ACUCOBOL-GT runtime that is a standard Windows DLL, and there is only one instance of the DLL in the application. Even though you might destroy the instance of the CVM assembly, Windows keeps the DLL loaded until the application terminates. If you later create a new instance of CVM, Windows gives it a handle to the loaded DLL. Once an instance of CVM executes the ShutDown method, then any current or subsequent instances will not work because the DLL is now in a shutdown state and can not be reactivated.

ShutDown is automatically called when CAcuCobol is destoyed. This method is useful in other contexts as well and is provided to maintain compatibility.

Usage:

```
public void ShutDown(int msg)
```

Where:

Variable	Definition
msg0	Is the default return code

FreeInfo de-allocates IntPtr pInfo control blocks that were allocated during a call to CallAcuCobol. If this is not called the blocks will be de-allocated when “wrunnet” is removed from the system or when class CVM is destroyed.

5.4.2.2 Properties

The following table contains a description of each property and its associated method. Please note that the get property, LastErrorMsg, returns the string of the last error message. The set properties set the string or property value.

When “wrunnet.dll” is referenced in a Visual Studio .NET project or the compiled COBOL program used the “--netdll” option, properties can be viewed using the object browser of Visual Studio .NET. When the “--netdll” option is used, properties are inluded in the “ProgramName_CVM.dll”.

String Types

Property Name	Get /Set	Description	Command Option	Platform	Method
ConfigFile	Set	Alternate configuration file	-c	Win, UNIX	Initialize
DebugCmds	Set	File containing debugger commands	-r	Win, UNIX	Initialize
DisplayAppend	Set	File to append display output	+o	Win, UNIX	Initialize
DisplayOut	Set	File for display output	-o	Win, UNIX	Initialize
EmbeddedLib	Set	Load configuration file from COBOL object library	--embedded	Win, UNIX	Initialize
ErrorsAppend	Set	Append to error messages file	+e	Win, UNIX	Initialize

String Types

Property Name	Get /Set	Description	Command Option	Platform	Method
ErrorsOut	Set	Error messages file	-e	Win, UNIX	Initialize
Import	Set	A variable for importing graphical screens	-import	Win	Initialize
KeyFile	Set	Keyboard input file	-i	Win, UNIX	Initialize
LastErrorMsg	Get	Returns the last error message string		Win, UNIX	No restriction
ObjLib	Set	Object file library	-y	Win, UNIX	Initialize
Plays	Set	File of input keystroke script	-k	Win, UNIX	Initialize
RunPath	Set	Folder where “wrun32.dll” is located	-runpath:	Win, UNIX	Initialize
Switches	Set	List of Special Names switches to turn on	-#	Win, UNIX	Initialize
TerminalOut	Set	Capture terminal output to a file	-t	Win, UNIX	Initialize

BOOL Types

Property Name	Get/ Set	Description	Command Option	Platform	Method
Cache	Set	Leave program in cache after execution	-cache	Win, UNIX	Call
CGIWarnings	Set	Suppress warning messages in CGI programs	-f	Win	Initialize
CharToGui	Set	Convert character screens to GUI equivalent	--CharToGui	Win	Initialize
Debug	Set	Execute ACUCOBOL-GT debugger	-d	Win, UNIX	Call
DumpMem	Set	Dump memory for memory access violations	-z	Win, UNIX	Initialize

BOOL Types

Property Name	Get/ Set	Description	Command Option	Platform	Method
ExtendedError	Set	Display extended error codes for file error “30”	-x	Win, UNIX	Initialize
IgnoreSignals	Set	Ignore terminal hang-up signals	-h	UNIX	Initialize
LinkageLength	Set	Disable Linkage item length test	-u	Win, UNIX	Initialize
ListConfig	Set	List contents of configuration file	-l	Win, UNIX	Initialize
NonNumeric	Set	Suppress warnings when non-numeric data is used as numeric data	-w	Win, UNIX	Initialize
NoSaveDebug	Set	Prevent debugger from reading and writing adb	--no-save-debug	Win, UNIX	Initialize
SafeMode	Set	Run in safe mode	-s	UNIX	Initialize
ShowError	Set	Display error message text in a MesssageBox	-show	Win	Call
TerminalInit	Set	Inhibit terminal initialization	-b	UNIX	Initialize
Unicode	Set	Pic X() parameters passed as unicode	-uni	Win, UNIX	Call
Wide	Set	Pic X() parameters passed As DBC strings	-wide	Win, UNIX	Call
ZipErrorFile	Set	Suppress warning messages in CGI programs	-g	Win, UNIX	Initialize

5.4.2.3 Error codes

Call returns the following error codes to the .NET environment. The .NET programmer can refer to the error code, an enumerator, to get more information.

When ShowError is “true”, Call displays the message text. The property, LastErrorMsg, is also available to retrieve the last error text string. When “wrunnet.dll” is referenced in a Visual Studio .NET project or the compiled COBOL program used the “--netdll” option, error codes can be viewed using the object browser of Visual Studio .NET. When the “--netdll” option is used, errorTypes is inluded in the “ProgramName_CVM.dll”.

```
public enum errorTypes : int
{
/* runtime is not thread-safe */
CS_MULTIPLE_OS_THREADS = -4,
CS_WIN_INIT_FAILED     = -3,
CS_STOP_RUN            = -2,
CS_CONT                = -1,
CS_OK                  = 0,  /* program executed with no errors */
CS_MISSING             = 1,  /* Program missing or inaccessible */
CS_NOT_COBOL           = 2,  /* Not a COBOL program */
CS_INTERNAL            = 3,  /* Corrupted program */
CS_MEMORY              = 4,  /* Inadequate memory available */
CS_VERSION             = 5,  /* Unsupported version of object code */
CS_RECURSIVE           = 6,  /* Program already in use */
CS_EXTERNAL            = 7,  /* Too many external segments */
CS_LARGE_MODEL         = 8,  /* Large-model program not supported */
CS_JAPANESE            = 14, /* Japanese extensions not supported */
CS_MULTITHREADED       = 22, /* Multithreaded CALL RUN illegal */
CS_AUTHORIZATION       = 23, /* Access denied */
CS_CONNECT_REFUSED     = 25, //Connection refused
//Program contains object code for a different processor
CS_MISMATCHED_CPU      = 27,
CS_SERIAL_NUMBER       = 28  /* Incorrect serial number */
//user count exceeded on remote server
CS_USER_COUNT_EXCEEDED = 29,
CS_LICENSE             = 30, /* License error */
CS_UNSUPPORTED_PARAM   = 31, /* unsupported parameter */
CS_COBOL_SIGNAL        = 65,
CS_COBOL_FATAL_ERROR   = 66
};
```

5.4.2.4 CompilerTypes

When “wrunnet.dll” is referenced in a Visual Studio .NET project or the compiled COBOL program used the “--netdll” option, CompilerTypes can be viewed using the object browser in Visual Studio .NET. When the “--netdll” option is used, CompilerTypes is included in the “ProgramName_CVM.dll”.

```
public enum CompilerTypes : byte
```

```
{
    NSE= 0,
    NSEZ= 1,
    CONS= 2,
    ANL= 3,
    ABS= 4,
    ABSJ= 5,
    PARAM= 6,
    GRPL= 7,
    ANSE= 8,
    GRPVL= 9,
    ANS= 10,
    ANSJ= 11,
    GRP= 12,
    GRPV= 13,
    NPU= 14,
    NNCU= 15,
    NSU= 16,
    NNU= 17,
    NSS= 18,
    NSSL= 19,
    NCU= 20,
    NBU= 21,
    NCS= 22,
    NNCS= 23,
    NIS= 24,
    NISL= 25,
    NPS= 26,
    NFP= 27,
    TEMP= 28,
    NPP= 29,
    NBS= 30,
    NNS= 31,
    NAT= 32,  /* codes are for double byte */
    NATJ= 33, /* codes are for double byte */
    NATE= 34, /* codes are for double byte */
    WID= 35,  /* codes are for double byte */
    WIDJ= 36, /* codes are for double byte */
    EWID= 37, /* codes are for double byte */
    TMP_PFX= 38,
    NEFP= 39,
    NBFP= 40
};
```

5.4.3 Using the ACUCOBOL-GT COM Server

An alternate way to provide COBOL services to a .NET assembly is through the ACUCOBOL-GT COM server.

The COM server is a COM object containing the ACUCOBOL-GT Windows runtime DLL. It provides a COM interface between the ACUCOBOL-GT runtime and programs running outside the runtime. For more information on the COM server, refer to section 3.2.1 of this guide.

When the COM server is added to a .NET project, .NET creates a proxy object, "Interop.AcuGTObjects.dll", that provides an interface between .NET and the ACUCOBOL-GT COM server object. The .NET proxy gathers data intended for use by the COBOL program and packages it in COM variant formats before sending it to the ACUCOBOL-GT COM server. This process is known as data marshalling. The COBOL program uses C$GETVARIANT and C$SETVARIANT to retrieve and update data held by the ACUCOBOL-GT COM server object. The proxy also unmarshals data received from ACUCOBOL-GT COM server before delivering it back to the .NET program. The proxy is designed to manage the interchange between the .NET and COBOL worlds.

Invoke an ACUCOBOL-GT program from a .NET assembly as follows:

1. Install and register the ACUCOBOL-GT COM server on the system running .NET. Install the COBOL object file there as well. Instructions are provided in section 3.2.1, "Using the ACUCOBOL-GT COM Server."

   Once the COM server is registered on the .NET system, the ACUCOBOL-GT runtime is then exposed to any program running in that environment.

2. To use the ACUCOBOL-GT COM Server, select a .NET application that requires the ACUCOBOL-GT interface. (The …\sample\dotnet\NetToAcuCobol directory contains a sample that you can use.)

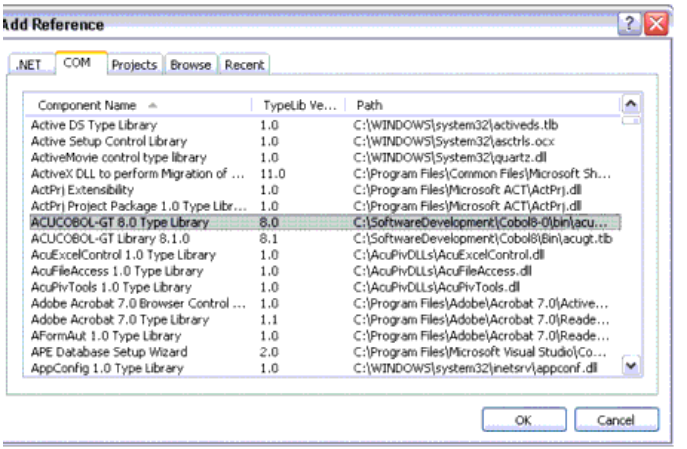
   a. In Visual Studio .NET, right-click the project name in the Solution Explorer window to display a selection menu.

b. Select **Add Reference** from the resulting pop-up menu to display the Add Reference dialog box.

c. Select the **COM** tab from the Add Reference dialog box.

d. Select ACUCOBOL-GT 8.0 Type Library (Acugt.tlb) from the list and click **OK**.

Visual Studio generates a .NET proxy module to interface with your COM object identified as “Interop.AcuGTObjects.dll”.



Visual Studio .NET Add Reference dialog box

3. Add the proxy interface to the .NET application source. For example, if you are using Visual Basic or C#, you would add the following:

   Visual Basic:

```
Dim AcugtInterface As Object
AcugtInterface = New AcuGTObjects.AcuGTClass
```

   C#:

```
AcuGTObjects.AcuGTClass AcugtInterface = new
AcuGTObjects.AcuGTClass();
```

4. Run the .NET service and it will invoke the COBOL program as necessary.

Note: The …\sample\dotnet\NetToAcuCobol directory contains a "ReadMe.txt" file that describes using the example with the "AcuGTObjects.AcuGTClass".

5.5 Calling .NET from COBOL

Use ACUCOBOL-GT's .NET bridging interface, "AcuToNet.dll", if you want to include .NET assemblies in your COBOL programs. This interface gives you access to .NET assemblies, both graphical and non-graphical, including user controls and Windows forms controls. You do not need to know object-oriented programming or the NET Framework.

This interface includes a graphical utility known as the .NET Definitions Generator, or "NETDEFGEN.exe". This utility allows you to view and select .NET assemblies for translation to a COBOL COPY file. COBOL programmers already familiar with the DISPLAY, CREATE, MODIFY, INQUIRE, and DESTROY statements can immediately write COBOL programs that consume .NET assemblies.

At run time, "AcuToNet.dll" starts the .NET CLR, loads and executes the requisite .NET components, and returns the results to the COBOL program.

The next Section provides details on adding and using .NET assemblies in COBOL programs by using NETDEFGEN and AcuToNET.dll.

Note: To execute NETDEFGEN, the samples, "AcuToNet.dll", and other items, you must have Microsoft .NET Development Framework Version 2.0. However, with NETDEFGEN, you can process Version 1.1 and 2.0 assemblies.

5.5.1 Using .NET assemblies in COBOL

The .NET bridging interface includes a graphical utility known as **NETDEFGEN**. This utility allows you to view and select .NET assemblies for translation to a COBOL COPY file. It is similar to the **AXDEFGEN** utility that ACUCOBOL-GT provides for programming with ActiveX and COM objects. However, **NETDEFGEN** is designed to work with .NET assemblies.

The COPY files generated by **NETDEFGEN** supply the ACUCOBOL-GT compiler with all the necessary information for interfacing with .NET assemblies. When you include the COPY files in your COBOL program, you can interact with the .NET assembly via these COBOL statements: DISPLAY, CREATE, MODIFY, INQUIRE, and DESTROY.

Invoke .NET assemblies from your ACUCOBOL-GT program as follows:

1. If it is not installed already, install and register the .NET framework on your development system.

2. Run “netdefgen.exe”. It is located in the \AcuGT\bin directory wherever you installed ACUCOBOL-GT on your machine. **NETDEFGEN** locates the .NET assemblies in the Global Assembly Cache (GAC) on the machine and lists them. (For more information on the **NETDEFGEN** utility, refer to the NETDEFGEN Utility Reference later in this section.)

3. From the list of cached assemblies displayed in the **NETDEFGEN** dialog box, select the .NET assembly that you want to include in your COBOL program, then select the NameSpace class or classes that you specifically want to access.

   Note: If the assembly of interest is not in the GAC, use the **Browse** button to navigate to the directory where the assembly resides.

   ---

4. Specify an output path and filename for the COBOL COPY file that will be generated. Click **Generate** when done.

   The utility automatically generates a COPY file for the chosen assembly. For information on the contents of **NETDEFGEN** COPY files, refer to the NETDEFGEN Utility Reference.

The utility also generates something known as an event DLL. The event DLL is named after the NameSpace class found in the COPY file, appended by the suffix ".dll".

5. In a code editor, open your ACUCOBOL-GT program and go to its Environment Division/Configuration Section.

6. In the COBOL program's Special-Names paragraph, enter a COPY statement for the COPY file that you specified in step 4. If you are adding several .NET assemblies, copy several COPY files into this paragraph. Add a period at the end of the paragraph. For example:

```
SPECIAL-NAMES
COPY "netcontrol1.def".
COPY "netcontrol2.def".
.
```

7. Add the .NET control to your program. Minimally, you must include the ASSEMBLY-NAME, NAMESPACE, and CLASS-NAME parameters that are found in the COPY file.

   If the .NET control that you are adding is graphical, (i.e., it has the word "Visual" in the COPY file NameSpace class definition), you can add it to your program in one of two ways:

   a. Go to the Screen Section of your program and add the new control to your screen. For example:

```
screen section.
01  screen-1.
    03 SOME-NETCONTROL, "@My.Assembly",
       LINE 1, COL 2,
       NAMESPACE IS "My.Test.Namespace",
       CLASS-NAME IS "MyGUIClass",
       CONSTRUCTOR IS CONSTRUCTOR2(PARM1, PARM2, PARM3,
       PARM4, PARM5, PARM6, PARM7),
       EVENT PROCEDURE IS USERCONTROL-EVENTS.
```

   b. Go to your program's Procedure Division and create the control using the DISPLAY statement. For example:

```
DISPLAY "@My.Assembly"
     NAMESPACE IS "My.Test.Namespace"
     CLASS-NAME IS "MyGUIClass"
     EVENT PROCEDURE IS MY-EVENT-PROCEDURE
     HANDLE IS MY-GUI-HANDLE.
```

If the .NET assembly that you are adding is non-graphical, use the CREATE statement to add it to your program as shown below:

```
CREATE "@My.Assembly"
     NAMESPACE IS "My.Test.Namespace"
     CLASS-NAME IS "MyClass"
     HANDLE IS MY-NONGUI-HANDLE.
```

Note that .NET properties, methods, and events should always be prepended with an "@" sign in case they clash with COBOL reserved words or ACUCOBOL-GT graphical control property and style names. The "@" character identifies the relationship of the name to .NET or ActiveX.

8. Add any optional .NET parameters to the Screen Section, DISPLAY, or CREATE statement. For example, add the FILE-PATH parameter to point to assemblies that will not be placed in the end-user's GAC (you must first create an XML file containing the file path), or add the CONSTRUCTOR parameter to instantiate a class. Valid .NET parameters are described in section 5.5.2.2.

9. Perform whatever functions you want on the control. If desired, you can modify a .NET control's properties or invoke methods using the MODIFY statement. For example:

   ```
   MODIFY MY-NONGUI-HANDLE printIteration = NUMBRPRINTS.
   MODIFY MY-NONGUI-HANDLE lastName = LAST-NAME.
   MODIFY MY-NONGUI-HANDLE "ToLog"("Hello From COBOL", 99,
   "It's a Good Thing").
   ```

   If you want to retrieve a property value, you can use the INQUIRE statement. For example:

   ```
   INQUIRE MY-NONGUI-HANDLE printIteration IN QPRINTS.
   INQUIRE MY-NONGUI-HANDLE lastName IN LAST-NAME.
   ```

10. To destroy .NET controls, use the DESTROY statement. Ultimately, all controls that you instantiated in step 7 should be destroyed. You can use DESTROY ALL to destroy all controls in the Screen Section or created with a DISPLAY statement. However, if the control was created with a CREATE statement, then it must be destroyed with a Format 1 DESTROY handle-name statement.

Note: For details on using the CREATE, DISPLAY, MODIFY, INQUIRE, and DESTROY statements, refer to section 6.6 in *ACUCOBOL-GT Reference Manual*. A complete sample program is provided below for your reference.

11. Compile the COBOL program.
12. Update the configuration file named "NetEvents.ini" with path statements of the event DLL(s) created when you ran **NETDEFGEN**, or place the DLL(s) in the same directory as the ACUCOBOL-GT runtime, "wrun32.exe". The runtime must be able to access these event DLLs as well as the .NET assembly itself.
13. Run your COBOL program via "wrun32.exe MyProgram.acu".

At run time, your COBOL program transparently communicates with an interface called "AcuToNet.dll". This interface starts the .NET CLR, loads the requisite .NET assemblies and event handlers, executes .NET assembly methods, and returns the results to the COBOL program. Refer to 'Optimizing the "AcuToNet.dll" interface' for information on optimizing the performance of "AcuToNet.dll".

5.5.1.1 CoCreate Instance Failed Error

When attempting to run your program, there is a special case that will cause you to receive a runtime error message of "CoCreate Instance failed". This case involves running on a Microsoft Vista or Windows 2008 machine that previously (or currently) had ACUCOBOL-GT Version 8.0 installed on that machine.

This error is due to a combination of versions 8.0 and 8.1 using different CLSIDs (a unique identifier) for AcuToNet.dll, and Microsoft's more restrictive version control methods for side by side DLLs.

To correct this situation you need to use the ACUCOBOL-GT utility **RegAsm20Acu.exe** to register the version 8.1 AcuToNet.dll. Do this by performing the following steps:

1. From a command prompt, navigate to the bin directory of AcuGT (Program Files\Acucorp\Acucbl810\AcuGT)

2. Entering the following command:

```
regasm20acu /register /codebase AcuToNet.dll
```

Note: When distributing version 8.1 ACUCOBOL-GT applications with .NET controls to Vista or Windows 2008 machines that previously had version 8.0 installed, you must perform the registration step on the target machine. In this case you would go to the directory containing the AcuToNet dll, and issue the RegAsm20Acu command as described in step 2 above. If the machine you are distributing to never had version 8.0 installed than it is not necessary to perform this step.

5.5.1.2 Sample program

The following example shows how to use the CREATE, DISPLAY, MODIFY, and INQUIRE statements to create and interact with the .NET control once its COPY file has been included in your COBOL program. First, you'll find a sample COPY file. Then you'll find a sample COBOL program, with comments, in bold.

NETDEFGEN COPY file

```
-----  Generated by NetDefGen  -----
OBJECT @ASSEMBLY
NAME "@My.Assembly"
VERSION "1.0.0.0"
CULTURE "neutral"
STRONG "3f6e8fa90dc2951b"

NAMESPACE "My.Test.Namespace"
CLASS "MyClass"

CONSTRUCTOR,  0, @CONSTRUCTOR1

* printIteration
                PROPERTY_GET, 0, @printIteration
                        RETURNING, "int", TYPE 3
* printIteration
                PROPERTY_PUT, 0, @printIteration
                       "int (Property Value)", TYPE 3

* Int32 ToLog(System.String, Int32, System.String)
                METHOD,  0, "@ToLog"
                         "BSTR" @StringIn, TYPE 8
                         "int" @someNumber, TYPE 3
```

```
                        "BSTR" @anotherString, TYPE 8
                        RETURNING "int", TYPE 3
*  Public fields
FIELD,   0, @lastName
                        RETURNING, "BSTR", TYPE 8

NAMESPACE "My.Test.Namespace"
CLASS "MyGUIClass"
VISUAL

CONSTRUCTOR,  0, @CONSTRUCTOR1

* LogRecordRead  (Int32)
              EVENT, -709034780, @MyGUIClasss_LogRecordRead

* LogRecordWritten  (Int32, System.String)
              EVENT, -1411090252, @MyGUIClass_LogRecordWritten

NAMESPACE "My.Test.Namespace"
CLASS "UserControl1"
VISUAL

        CONSTRUCTOR,  0, @CONSTRUCTOR1

        CONSTRUCTOR,  0, @CONSTRUCTOR2
                        "BSTR" @userStuff, TYPE 8
                        "int" @intData, TYPE 3
                        "unsigned int" @uintData, TYPE 19
                        "single" @floatData, TYPE 4
                        "double" @doubleData, TYPE 5
                        "short" @shortintData, TYPE 2
                        "unsigned short" @ushortintData, TYPE 18

 -----  End Generated NetDefGen Code  -----
```

COBOL program

```
* Handles can be associated with a specific
* Assembly.NameSpace.Class
* Use this form when the COBOL statement, MODIFY - INQUIRE, etc.,
* uses the handle before a CREATE or DISPLAY statement occurs
* in the program.

77 MY-NONGUI-HANDLE USAGE IS HANDLE OF
"@My.Assembly.My.Test.Namespace.MyClass".

77  MY-GUI-HANDLE       USAGE IS HANDLE.
77  NUMBRPRINTS         USAGE IS SIGNED-INT VALUE 3.
77  QPRINTS             USAGE IS SIGNED-INT.
```

```
77  PARAM1              USAGE IS SIGNED-INT.
77  PARAM2              PIC x(128).
77  LAST-NAME           PIC x(32).
77  PARM1               pic x(12) VALUE "HELLO WORLD".
77  PARM2               USAGE IS SIGNED-INT VALUE 1111.
77  PARM3               USAGE IS UNSIGNED-INT VALUE 2222.
77  PARM4               USAGE IS FLOAT VALUE 0.3333.
77  PARM5               USAGE IS DOUBLE VALUE 123456.55.
77  PARM6               USAGE IS SIGNED-SHORT VALUE 4444.
77  PARM7               USAGE IS UNSIGNED-SHORT VALUE 5555.

*CREATE  - instantiate a NON-GUI CLASS.

CREATE "@My.Assembly"
NAMESPACE IS "My.Test.Namespace"
CLASS-NAME IS "MyClass"
EVENT PROCEDURE IS EVENT-PROC
HANDLE IS MY-NONGUI-HANDLE.

*DISPLAY - instantiate a GUI CLASS. GUI classes have a keyword
*VISUAL in the COPY file after the CLASS keyword.

DISPLAY "@My.Assembly"
NAMESPACE IS "My.Test.Namespace"
CLASS-NAME IS "MyGUIClass"
EVENT PROCEDURE IS MY-EVENT-PROCEDURE
HANDLE IS MY-GUI-HANDLE.

*INQUIRE - retrieve the value of a PROPERTY OR FIELD.

INQUIRE MY-NONGUI-HANDLE printIteration IN QPRINTS.
INQUIRE MY-NONGUI-HANDLE lastName IN LAST-NAME.

*MODIFY  - execute a method.  Methods are case sensitive. They
*must match the COPY file case and be enclosed in quotes.

MODIFY MY-NONGUI-HANDLE "ToLog"("Hello From COBOL", 99, "It's a Good
Thing").

*MODIFY  - set the value of a PROPERTY OR FIELD.

MODIFY MY-NONGUI-HANDLE printIteration = NUMBRPRINTS.
MODIFY MY-NONGUI-HANDLE lastName = LAST-NAME.

*Capture events and retrieve event data - Use EVENT-DATA-2 or the
*COPY file event name.

*EVENT-DATA-2 and the COPY file event name are an event ID.  The
*runtime tries to locate the last event thrown by matching the
```

```
*event ID.  If you use CONTROL-HANDLE or HANDLE IS from the CREATE
*and DISPLAY statements, the runtime tries to locate the last
*event thrown by matching the .NET Interface for the control.
*Using a CONTROL-HANDLE or COPY file event name/numeric ID makes
*a difference when event procedures cause another event before
*collecting the first event's DATA.  If the events are different,
*use the COPY file event name to retrieve the desired event data.
*If you use CONTROL-HANDLE, the most recent event, that is, the
*Last In First Out (LIFO), received by the runtime for a .NET
*interface is returned to a COBOL program possibly resulting in
*incorrect event data.

MY-EVENT-PROCEDURE.
EVALUATE EVENT-TYPE
    WHEN MSG-NET-EVENT
EVALUATE EVENT-DATA-2
    WHEN @MyGUIClass_LogRecordWritten
CALL "C$GETNETEVENTDATA" USING @MyGUIClass_LogRecordWritten PARAM1
PARAM2
    WHEN @MyGUIClass_LogRecordRead
CALL "C$GETNETEVENTDATA" USING EVENT-DATA-2 PARAM1
END-EVALUATE
END-EVALUATE.

*Screen section .NET Control with a Constructor

screen section.
01  screen-1.
    03  SOME-NETCONTROL, "@My.Assembly",
        LINE 1, COL 2,
        NAMESPACE IS "My.Test.Namespace",
        CLASS-NAME IS "UserControl1",
       CONSTRUCTOR IS CONSTRUCTOR2(PARM1, PARM2, PARM3, PARM4, PARM5,
        PARM6, PARM7),
        EVENT PROCEDURE IS USERCONTROL-EVENTS.

*Coding exceptions. NameSpace is optional in C# and VB NET and
*therefore may not appear in a COPY file. However, DISPLAY,
*CREATE, and Screen Section COBOL statements require a NameSpace
*entry. When a COPY file is missing a NameSpace keyword, use the
*class name as the NameSpace value on DISPLAY, CREATE, and Screen
*Section statements.
```

5.5.1.3 Limits and restrictions

ACUCOBOL-GT programs cannot create or retrieve .NET objects or pass .NET objects as parameters. If you want to use a Windows Forms control or .NET assembly that relies on a .NET object, you need to write a C# or Visual

Basic.NET intermediary program to handle it. (You can tell that a .NET method or field is using .NET objects if it contains the phrase "STORED-OBJECT".) The intermediary program can be written to interface with the assemblies that use objects, then you can use NETDEFGEN to generate a COBOL COPY file for the intermediary program.

The .NET interface cannot be used to instantiate .NET executables, either. The CREATE and DISPLAY statements associated with this interface support only .NET DLLs and controls. However, you can make a call to the C$SYSTEM library routine to spawn a .NET executable. This routine is described in Appendix I in ACUCOBOL-GT Appendices.

ACUCOBOL-GT supports methods that have integer, unsigned integer, byte, string, float single, and double precision data types, all known as blittable data types in the .NET world. Blittable data types are those that have a common representation in both managed (MSIL) and unmanaged (COBOL) code. Non-blittable types are ambiguous and not supported.

A special case generates a runtime error of "CoCreate Instance failed". See Section 5.5.1.1 for details on the case and how to overcome the error.

5.5.1.4 Optimizing the "AcuToNet.dll" interface

At run time, your COBOL program transparently communicates with an interface file called "AcuToNet.dll". This interface starts the .NET CLR, loads the requisite .NET assemblies and event handlers, executes .NET assembly methods, works with properties, and returns the results to the COBOL program.

The ACUCOBOL-GT to .NET bridging interface is delivered in compiled MSIL format. Any time a .NET assembly or "AcuToNet.dll" is loaded, it undergoes an additional compilation in the Microsoft Just-In-Time (JIT) compiler. This can raise questions about performance and whether it's possible to create native code "executables" in .NET.

Microsoft provides a tool in the .NET Framework directory called the Native Image Generator ("NGen.exe") that can be used for this purpose. This utility compiles an assembly into a "native image". You could run the NGen utility on "AcuToNet.dll", located in the ACUCOBOL-GT /bin directory. NGen will compile and place it in the GAC under the same name. To avoid

confusion you could rename the original file to “ORIG_AcuToNet.dll”. Then when the CLR loads the assembly, it will check the cache to see if a precompiled version exists, and if it does, it will load that.

Although you might think this will improve performance, there a number of drawbacks to consider (including maintenance issues), and you may not obtain the performance improvement you expect either. NGen creates a native image for a hypothetical machine architecture. The advantage is that it runs on any x86 processor, for example. However, when the JIT compiler executes, it takes the specific machine it’s running on into account and makes appropriate optimizations. The result is that assemblies dynamically compiled at run time often out perform precompiled assemblies.

Another drawback is that changes to a system’s hardware configuration or operating system (like service pack updates) often invalidate the precompiled assembly.

5.5.1.5 .NET control distribution and licensing

If you want to include a .NET assembly in your COBOL application, you must first acquire it and install it on the development machine. In some cases, a partner may provide the control. In others, you may download it off the Internet and license it for distribution.

Unlike ActiveX, .NET does not have a specific parameter for licensing but does have a class that performs this function. .NET uses a design-time licensing model that is verified again at run time, usually in the CONSTRUCTOR phase. Because ACUCOBOL-GT supports constructors, it also supports custom run-time licensing that requires parameters.

If you encounter an ActiveX control that was converted to .NET using Microsoft’s “Aximp.exe” utility and you experience licensing conflicts and violations, use the ActiveX version of the control instead of the converted .NET version.

To use ActiveX controls with ACUCOBOL-GT, use the AXDEFGEN utility rather than NETDEFGEN. It is documented in section of this guide.

5.5.1.6 Name clashes

Often a .NET assembly may have property, method, or event names that are the same as COBOL reserved words or ACUCOBOL-GT standard property or style names. This creates ambiguity for the compiler. In addition, because .NET class names are used in the USAGE HANDLE clause of data description entries, in Screen Section items, and in DISPLAY statements, they may also cause ambiguities with COBOL reserved words.

To avoid these ambiguities, NETDEFGEN prepends an “at” sign character (“@”) to every class, property, method, and event name in the generated COPY file.

In addition, ambiguity may occur with event names when two or more .NET assemblies define the same event name. To reduce this possibility, NETDEFGEN prepends the control name to each event name. For example, if a .NET control named “MyControl” has an event called “RightMouseButtonClick”, NETDEFGEN names the control “@MyControl” and the event “@MyControlRightMouseButtonClick”.

The “@” sign is not required unless ambiguities in the meaning exist in a certain context. However, to guard against unanticipated name conflicts and to ensure clarity in the reading and maintenance of the source code, we strongly recommend that you always use “@” when referring to a .NET property, style, or method in your source code. If you do not use an “@” sign and a clash occurs, a compiler error results.

5.5.2 NETDEFGEN Utility Reference

The NETDEFGEN utility is a dialog-based application designed to facilitate the consumption of .NET assemblies from Windows-based ACUCOBOL-GT programs. The role of NETDEFGEN is to locate the names of .NET assemblies on the user’s machine and generate a COBOL COPY file for the selected assembly.

The COPY file is used by the ACUCOBOL-GT compiler for syntax and parameter type checking as well as efficient code generation. For instructions on how to use NETDEFGEN and what to do with the COPY file once it is generated, refer to section 5.5.1.

Note: If you want to include a .NET assembly in your COBOL application, you must first acquire it and install it on the development machine. If you have downloaded the control off the Internet, you may need to license it for distribution. See ".NET control distribution and licensing" for more information.

The "netdefgen.exe" file is located in the \AcuGT\bin directory wherever you installed ACUCOBOL-GT (C:\Program Files\Acucorp\Acucbl800\AcuGT\bin by default). It is accessible from the Windows Start menu. When you run NETDEFGEN, a dialog box is displayed.

NETDEFGEN dialog box

Field or Button	Description
Assembly Location	Browse to the directory where the assembly from which you want to produce a COPY file resides. By default, this is the computer's path to the Global Assembly Cache (GAC). Click the **GAC** button to restore this field to the GAC path at any time.
Assemblies	If the Global Assembly Cache (GAC) is the currently selected Assembly Location, then this list box contains all the assemblies found in the GAC. If the directory selected is other than the GAC, then all of the DLL and EXE files in that directory are listed (both assemblies and non-assemblies). Select the assembly of interest from the list. All NameSpaces and classes found in that assembly display in the list box labeled Namespace Classes.
Namespace Classes	By default, when an assembly has been selected from the Assemblies list, all of the items found in the assembly are selected. Assemblies can contain many NameSpaces and classes, which can result in very large COPY files containing many references and declaration that are unused by your application. To reduce the size of your COPY file, select only those items that are needed by your application.
Copyfile To Create	Browse to the directory and filename of the COPY file to generate. If event handlers are generated, they will be located in the same directory with a filename prefix of the NameSpace and Class and a suffix of ".dll".

Field or Button	Description
Generate Copyfile	The **Generate Copyfile** button is enabled only when an assembly is selected, at least one NameSpace/class is selected, and a valid COPY file destination has been entered into the **Copyfile To Create** field. Click this button to generate the COPY file and any required event handlers. Success or failure messages appear in the status area at the bottom of the screen.
Help	Click the **Help** button to view the help file for the utility.
Exit	Select **Exit** from the menu bar to exit the utility.
Settings	Select **Settings** from the menu bar to change the default settings of the utility, including diagnostic settings. See section 5.5.2.1 for more information.

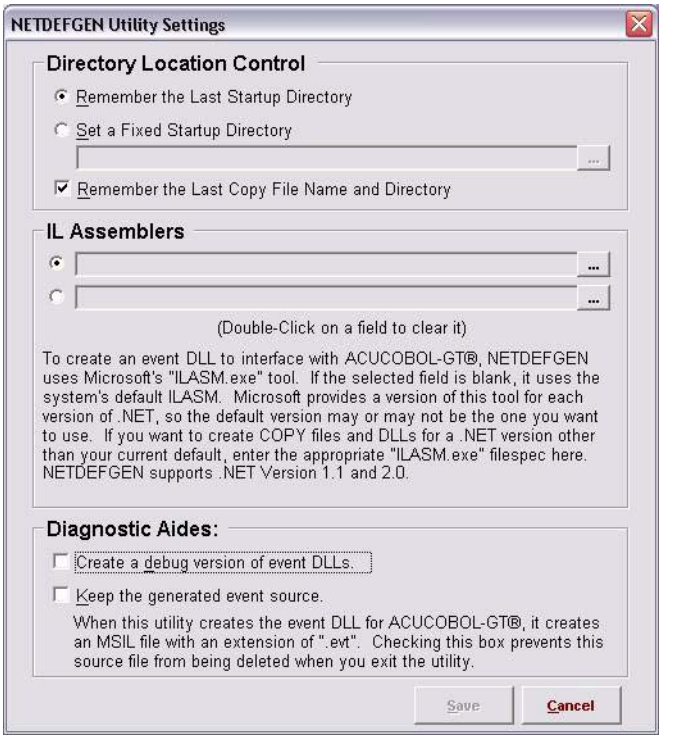
NETDEFGEN interrogates the assembly and produces an English translation for NameSpaces, classes, methods, enumerations, properties, and events that the ACUCOBOL-GT compiler understands. At run time, the COBOL program accesses the assembly using a NameSpace and class from the COPY file.

The resultant COPY file is included in a COBOL program for use by the compiler and programmer. In addition to the COPY file, NETDEFGEN generates event handlers for each class in the assembly. This is native MSIL code that listens for .NET assembly events, routes them to the ACUCOBOL-GT runtime, which in turn delivers events to the COBOL program. Event handlers are loaded by the runtime when the COBOL program executes a DISPLAY or CREATE statement or has a reference to an assembly in a Screen Section. The runtime locates event handlers from a configuration file called “NetEvents.ini”. This file contains a list of directory paths indicating where to search for event handlers. “NetEvents.ini” must be located in the same directory as the runtime. If “NetEvents.ini” is not present, the runtime attempts to load event handlers from the folder where the runtime is located.

Windows base types—such as integer, unsigned integer, float, double, byte, character, and character string—are supported. However, .NET objects as parameters are not supported. Any method or constructor requiring parameter object creation by the COBOL program fails.

5.5.2.1 Changing Default NETDEFGEN Settings

To enter settings that will change the default behavior of the NETDEFGEN utility, select **Settings** from the menu bar. The Settings dialog appears.



Settings dialog box

Field or Button	Description
Directory Location Control:	
Remember the Last Startup Directory	When this radio button is selected, as it is by default, the utility remembers the last Assembly Location selected and restores the selection on subsequent executions. Note that this is not affected by using the GAC button. You can select the GAC at any time and your previously selected directory is still retained.
Set a Fixed Startup Directory:	Use this selection if you want the NETDEFGEN utility to always start in the same Assembly Location directory. You can always Browse to a different directory if needed.
Remember the Last Copy File Name and Directory	When this radio button is selected, as it is by default, the utility remembers the last filename and directory selected under **Copy File to Create** and restores the selection on subsequent executions. Uncheck it if you want the utility to always assume that the startup location is the destination.
Intermediate Language (IL) Assemblers:	
This option allows you to use up to two different versions of .NET. Enter the paths of the versioned tools and select the radio button for the version you want the utility to use. For example, enter the path to Microsoft .NET Development Framework Version 2.0 and select that version if you want the utility to process .NET 2.0-compatible assemblies. By default, it processes the latest version that you have installed. Double-clicking the field clears it, causing the utility to use whichever version is the current default on the development machine.	
Diagnostic Aides:	
Create a debug version of event DLLs.	Causes the utility to include debug information in any event DLLs it creates.

Field or Button	Description
Keep the generated event source.	Keeps the intermediate language source code that the utility generates when creating an event DLL. This can be most useful when working with our Technical Support to resolve a problem.

5.5.2.2 NETDEFGEN COPY files

COPY files that have been generated by the NETDEFGEN utility can contain any of several .NET parameters. The parameters are described below. The same parameters have been added to the ACUCOBOL-GT DISPLAY, CREATE, and Screen Section statements.

To access a .NET assembly, your COBOL program must pass at least the ASSEMBLY-NAME, NAMESPACE, and CLASS-NAME parameters. The rest of the .NET parameters are optional. That is because the compiler extracts these parameters from the COPY file if they are not included in your program. The exception is FILE-PATH, which is never included in the COPY file. If you want to include a FILE-PATH, you must encode it as described in the table below.

You pass .NET parameters when you first instantiate a .NET control in your COBOL program, that is, in the CREATE, DISPLAY, or Screen Section statement.

Note that the returning value of any field in the parameter list can be updated and returned to your COBOL program if desired.

COPY File Parameter	Parameter to Pass	Required/ Optional	Description
NAME	ASSEMBLY-NAME	Required	Filename of assembly, without extension
NAMESPACE	NAMESPACE	Required	Assembly NameSpace
CLASS	CLASS-NAME	Required	Class name
VERSION	VERSION	Optional	Version number of assembly
CULTURE	CULTURE	Optional	Cultural information like language, country, region; default is neutral.

COPY File Parameter	**Parameter to Pass**	**Required/ Optional**	**Description**
STRONG	STRONG-NAME	Optional	PublicKeyToken; a cryptographic key that is generated by the Microsoft utility “sn.exe”. All assemblies that are loaded in the GAC must include a STRONG-NAME key. If an assembly has a key, the compiler automatically retrieves it from the COPY file when it encounters a NAMESPACE/CLASS parameter.
CONSTRUCTOR(n)	CONSTRUCTOR	Optional if a default constructor exists	All classes that result in an object have a constructor, which is a sort of method. When a class is instantiated with a CREATE, DISPLAY, or Screen Section statement in COBOL, this constructor is the first thing that is executed. It is executed once during the instantiation phase of an object. Constructors, like methods, can have parameters, although most do not. The default constructor usually has no parameters. But if a .NET programmer has written a control that has constructors with parameters, you will see the word “CONSTRUCTOR” in the COPY file, followed by a number and the parameters associated with the constructor. This is called overloading. It occurs when the same method name has different parameters. Following is an example of a constructor that has been generated by the **NETDEFGEN** utility:

COPY File Parameter	Parameter to Pass	Required/ Optional	Description
			OBJECT @ASSEMBLY NAME "@My.Assembly" VERSION "1.0.0.0" CULTURE "neutral" STRONG "3f6e8fa90dc2951b" NAMESPACE "My.Test.NameSpace" CLASS "MyClass" CONSTRUCTOR, 0, @CONSTRUCTOR1 CONSTRUCTOR, 0, @CONSTRUCTOR2 "BSTR" @value, TYPE 8 CONSTRUCTOR, 0, @CONSTRUCTOR3 "int" @overloadedconstruct, TYPE 3 Because CONSTRUCTOR1 has no parameters, you can use a CREATE, DISPLAY, or Screen Section entry without a CONSTRUCTOR parameter. If you want to pass values, then you would select CONSTRUCTOR2 or CONSTRUCTOR3. The following depicts a CONSTRUCTOR parameter in a CREATE statement. 77 MY-ASSEMBLY-HANDLE USAGE IS HANDLE. CREATE "My.Assembly" NAMESPACE IS "My.Test.NameSpace" CLASS-NAME IS "MyClass" CONSTRUCTOR IS CONSTRUCTOR3 123456) HANDLE IS MY-ASSEMBLY-HANDLE.
MODULE identifier	MODULE	Optional	Assemblies can have multiple modules within them; that is, assemblies constructed from other assemblies. MODULE identifies the file where this NameSpace class combination resides. It is used by the runtime to select a module within an assembly.

COPY File Parameter	**Parameter to Pass**	**Required/ Optional**	**Description**

n/a	FILE-PATH	Optional	By default, the runtime looks for .NET assemblies in the end user's GAC. If it can't find the requested assembly in the GAC, it looks in the same directory as the runtime, "wrun32.exe".
			
			Use FILE-PATH when the assembly that the program must access does not reside in the user's GAC or in the same directory as "wrun32.exe".
			
			To include a FILE-PATH, first create an XML file containing the full file path where the assembly is located. Then in your COBOL program, include the FILE-PATH parameter followed by the full path of an XML file. If the FILE-PATH contains a filename only, the file is loaded from the same directory as the runtime.
			
			Following is an example of a FILE-PATH XML file:
			
			`<?xml version="1.0" encoding="utf-8" ?>`
			`<FILESPEC>`
			`<Assembly>AmortControl</Assembly>`
			`<Module>amortcontrol.dll</Module>`
			`<StrongName />`
			`<Version>1.0.1242.11216</Version>`
			`<Culture>neutral</Culture>`
			`<FilePath>E:\AmortControl\bin\Debug\AmortControl.dll</FilePath>`
			`</FILESPEC>`

5.5.2.3 Passing data as parameters

The .NET CLR is type-sensitive and generates or throws an error if an exact match on a data type does not occur. At run time, ACUCOBOL-GT attempts to convert data types to the format expected by the CLR. For the best possible results, when passing data as parameters to .NET functions, use the table below to determine what parameter to pass.

For example, when the COPY file says “int”, pass “SIGNED-INT” for the parameter type in your COBOL program. The ACUCOBOL-GT runtime converts this to “System.Int32” before passing it to the .NET assembly. The “int”, “SIGNED-INT”, and “System32.Int32” parameters all represent a 4-byte field holding signed binary data. The runtime also converts PIC s999 to a SIGNED-INT field.

COPY File Parameter	Parameter to Pass	Converted to .NET
BSTR - TYPE 8	PIC X(n)	System.String
int - TYPE 3	SIGNED-INT	System.Int32
unsigned int - TYPE 19	UNSIGNED-INT	System.UInt32
single - TYPE 4	FLOAT	System.Single
double - TYPE 5	DOUBLE	System.Double
short - TYPE 2	SIGNED-SHORT	System.Int16
unsigned short - TYPE 18	UNSIGNED-SHORT	System.UInt16
unsigned char - TYPE 17	PIC X	System.Byte

Be aware that the compiler does not always know which method declaration to use when methods are overloaded. In your COBOL program, be sure to use COBOL types that match the COPY file method declaration.

5.5.2.4 NETDEFGEN methods

The NETDEFGEN COPY file contains methods for classes within NameSpaces. These are the program functions for that .NET object. For example:

```
METHOD, 0, "@TypesTest"
           "unsigned int" @someUint, TYPE 19
           "single" @someFloat, TYPE 4
```

```
"unsigned char" @someByte, TYPE 17
"double" @someDouble, TYPE 5
"boolean" @someBool, TYPE 11
 RETURNING "int", TYPE 3
```

Use the COBOL MODIFY statement to execute a method. The RETURNING value of any field in the parameter list may be updated and returned to the COBOL program. Update fields, IN/OUT, can be identified by types in the 16000 range. For example:

```
"int" @somefield TYPE 16387
```

Be aware that the compiler does not always know which method declaration to use when methods are overloaded. In your COBOL program, be sure to use COBOL types that match the COPY file method declaration.

Listed below are some guidelines on passing parameters to methods. Following these guidelines should help you ensure that the correct overloaded definition is compiled.

- You can pass character strings as a PIC X…X defined variable or a quoted string literal. For example “This is a quoted string”.
- You can pass an unsigned integer as a USAGE UNSIGNED-INT defined variable or as a constant. For example, “0”, “1”, “356”, and so on.
- You can pass a signed integer as a USAGE SIGNED-INT defined variable or as a constant. For example, “+0”, “+1”, “+356”, “-34”, and so on.
- You can pass a byte data type as a PIC X COMP-X defined variable.
- You can pass a boolean data type as a PIC 9 defined variable that is casted as a boolean type:

  ```
  BOOL-PARAM    PIC 9.
  "@methodname" (BOOL-PARAM AS VT_BOOL)
  ```

  or as a constant. For example, 00 = false, 01 = true.

  ```
  "@methodname" (01)
  ```

Note: When casting data types is required, you must refer to the ACUCOBOL-GT provided COPY file “activex.def” in the Working-Storage section of your program.

- You can pass a signed byte data type as a USAGE SIGNED-INT defined variable and then casted as a signed byte type:

```
SBYTE-PARAM    USAGE SIGNED-INT.
"@methodname" (SBYTE-PARAM AS VT-I1)
```

- You can pass a Microsoft date data type as a USAGE DOUBLE defined variable and then casted as a date type:

```
DATE-PARAM    USAGE DOUBLE.
"@methodname" (DATE-PARAM AS VT-DATE)
```

For more examples of these and other forms of .NET method calls, see the sample application “MethodTests” in your Acucorp \sample\dotnet directory.

5.5.2.5 NETDEFGEN properties

NETDEFGEN translates all properties in the NameSpace class and includes them in the COPY file. For example:

```
* myPet
       PROPERTY_GET, 0, @myPet
          RETURNING, "int", TYPE 3
* myPet
       PROPERTY_PUT, 0, @myPet
          "int (Property Value)", TYPE 3
```

To retrieve a property value, use the INQUIRE statement. To change a property value, use the MODIFY statement.

5.5.2.6 NETDEFGEN events

NETDEFGEN generates event handlers for each class in the assembly. This MSIL code listens for .NET assembly events and routes them to the ACUCOBOL-GT runtime, which in turn delivers events to the COBOL program. If you see EVENT in the NETDEFGEN COPY file, as shown below, you know that event handlers were generated for this particular class.

```
* ooHHello  (Int32, System.String)
       EVENT, 1984084948, @MyClass_ooHHello
```

To capture events and data, write an event procedure using the COBOL EVENT PROCEDURE IS phrase on the DISPLAY, CREATE, and Screen Section statements. Call the C$GETNETEVENTDATA library routine described in Appendix I in ACUCOBOL-GT Appendices. For example:

```
EVALUATE EVENT-DATA-2
  WHEN @MyClasss_ooHHello
  CALL "C$GETNETEVENTDATA" USING EVENT-DATA-2 PARAM1
```

5.5.2.7 NETDEFGEN enumerators

If the .NET assembly has enumerators, the NETDEFGEN COPY file lists them all and makes them accessible to the COBOL program. For example, the COPY file might list:

```
* "int animals_horses"
          ENUMERATOR, @animals_horses,  1
```

If the COPY file lists enumerators, you can use the enumerators in your COBOL program to set properties or invoke methods. This allows you to use a text name in place of a value, enhancing the readability of your program. The compiler checks the COPY file and substitutes the value associated with the name. For example, to set a property to the enumerator shown above, you could code:

```
MODIFY MyHandle  myPet = @animals_horses.
```

rather than

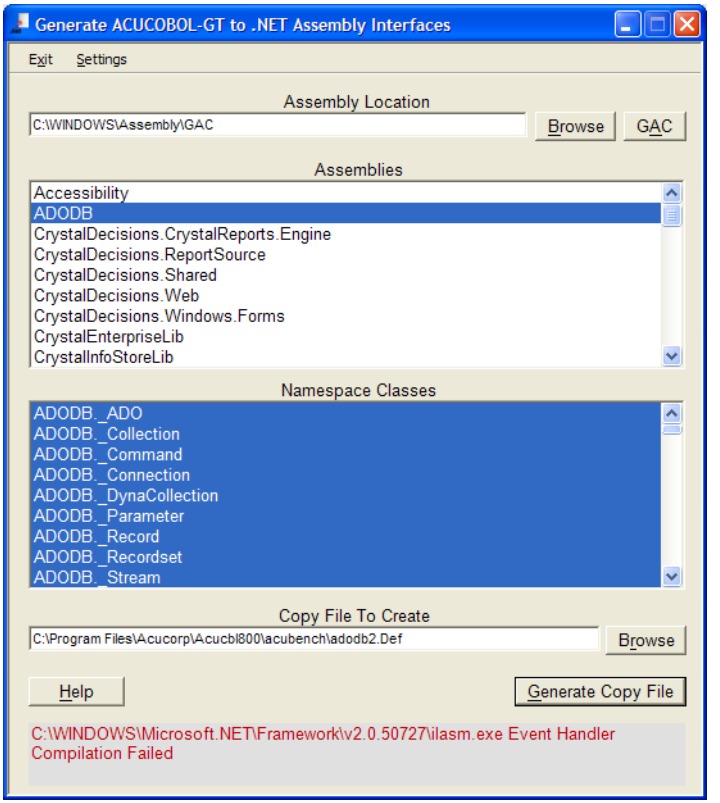
```
MODIFY MyHandle  myPet = 1.
```

5.5.2.8 NETDEFGEN errors

Occasionally when using NETDEFGEN, you may encounter the error message:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ilasm.exe Event
Handler Compilation Failed
```

as shown on the bottom of the screen below.



This is a Microsoft message indicating that “ilasm.exe,” a Microsoft executable, could not generate an event handler, probably because the assembly did not specify events. Our NETDEFGEN did not generate this message. It generates the COPY file normally despite the Microsoft error.

5.5.2.9 Sample COPY file

Following is an example of a COBOL COPY file generated by the NETDEFGEN utility:

```
* .NET Copy Book - Generated On 1/28/2004 12:25:20 PM

     OBJECT @ASSEMBLY
     NAME "@AmortControl"
            VERSION "1.0.1266.13363"
            CULTURE "neutral"
            STRONG "null"

* FULLY-QUALIFIED-NAME AmortControl.AmortCalc, AmortControl,
Version=1.0.1266.13363, Culture=neutral, PublicKeyToken=null

* AmortControl.AmortCalc
     NAMESPACE "AmortControl"
     CLASS "AmortCalc"
     MODULE "amortcontrol.dll"

          CONSTRUCTOR,  0, @CONSTRUCTOR1

* Void ProcData(Double, UInt32, Double, Double)
          METHOD,  0, "@ProcData"
                   "double" @AmortizAmount, TYPE 5
                   "unsigned int" @Months, TYPE 19
                   "double" @InterestRate, TYPE 5
                   "double" @WhatIfMonthlyPayment, TYPE 5

* Public - fields
          FIELD,   0, @MonthlyPayment
                   RETURNING, "double", TYPE 5
          FIELD,   0, @TotalInterest
                   RETURNING, "double", TYPE 5
          FIELD,   0, @TotalPayment
                   RETURNING, "double", TYPE 5
          FIELD,   0, @WhatIfTotalInterest
                   RETURNING, "double", TYPE 5
          FIELD,   0, @WhatIfTotalPayment
                   RETURNING, "double", TYPE 5
          FIELD,   0, @WhatIfMonths
                   RETURNING, "unsigned int", TYPE 19
          FIELD,   0, @Yearly_Interest
                   RETURNING, "STORED_OBJECT", TYPE 69
          FIELD,   0, @Yearly_Principal
                   RETURNING, "STORED_OBJECT", TYPE 69
          FIELD,   0, @Life_Interest
                   RETURNING, "STORED_OBJECT", TYPE 69
```

```
          FIELD,   0, @Life_Principal
                   RETURNING, "STORED_OBJECT", TYPE 69
          FIELD,   0, @Month_Interest
                   RETURNING, "STORED_OBJECT", TYPE 69
          FIELD,   0, @Month_Principal
                   RETURNING, "STORED_OBJECT", TYPE 69

* FULLY-QUALIFIED-NAME AmortControl.CalcFired, AmortControl,
Version=1.0.1266.13363, Culture=neutral, PublicKeyToken=null

* AmortControl.CalcFired
     NAMESPACE "AmortControl"
     CLASS "CalcFired"
     MODULE "amortcontrol.dll"

         CONSTRUCTOR,  0, @CONSTRUCTOR1
                  "STORED_OBJECT" @object, TYPE 69
                  "STORED_OBJECT" @method, TYPE 69

* Void EndInvoke(System.IAsyncResult)
          METHOD,  0, "@EndInvoke"
                   "STORED_OBJECT" @result, TYPE 69

* System.IAsyncResult BeginInvoke(System.AsyncCallback,
System.Object)
          METHOD,  0, "@BeginInvoke"
                   "STORED_OBJECT" @callback, TYPE 69
                   "STORED_OBJECT" @object, TYPE 69
                     RETURNING "STORED_OBJECT", TYPE 69

* Void Invoke()
          METHOD,  0, "@Invoke"

* Public - fields

* FULLY-QUALIFIED-NAME AmortControl.UserControl1, AmortControl,
Version=1.0.1266.13363, Culture=neutral, PublicKeyToken=null

* AmortControl.UserControl1
     NAMESPACE "AmortControl"
     CLASS "UserControl1"
     MODULE "amortcontrol.dll"
         VISUAL

          CONSTRUCTOR,  0, @CONSTRUCTOR1

* Void Dispose(Boolean)
          METHOD,  0, "@Dispose"
                   "boolean" @disposing, TYPE 11
```

```
* Void add_FireCalc(AmortControl.CalcFired)
         METHOD,  0, "@add_FireCalc"
                  "STORED_OBJECT" @value, TYPE 69

* Void remove_FireCalc(AmortControl.CalcFired)
         METHOD,  0, "@remove_FireCalc"
                  "STORED_OBJECT" @value, TYPE 69

* System.String get_TotalInterest()
         METHOD,  0, "@get_TotalInterest"
                    RETURNING "BSTR", TYPE 8

* System.String get_TotalPayment()
         METHOD,  0, "@get_TotalPayment"
                    RETURNING "BSTR", TYPE 8

* System.String get_MonthPayment()
         METHOD,  0, "@get_MonthPayment"
                    RETURNING "BSTR", TYPE 8

* System.String get_WhatIfTotalInterest()
         METHOD,  0, "@get_WhatIfTotalInterest"
                    RETURNING "BSTR", TYPE 8

* System.String get_WhatIfTotalPayment()
         METHOD,  0, "@get_WhatIfTotalPayment"
                    RETURNING "BSTR", TYPE 8

* System.String get_WhatIfMonths()
         METHOD,  0, "@get_WhatIfMonths"
                    RETURNING "BSTR", TYPE 8

* Void InitializeComponent()
         METHOD,  0, "@InitializeComponent"

* Void calcBtn_Click(System.Object, System.EventArgs)
         METHOD,  0, "@calcBtn_Click"
                  "STORED_OBJECT" @sender, TYPE 69
                  "STORED_OBJECT" @e, TYPE 69

* Public - fields
         FIELD,   0, @columnHeader1
                  RETURNING, "STORED_OBJECT", TYPE 69

* TotalInterest
         PROPERTY_GET, 0, @TotalInterest
                  RETURNING, "BSTR", TYPE 8
```

```
* TotalPayment
          PROPERTY_GET, 0, @TotalPayment
                    RETURNING, "BSTR", TYPE 8

* MonthPayment
          PROPERTY_GET, 0, @MonthPayment
                    RETURNING, "BSTR", TYPE 8

* WhatIfTotalInterest
          PROPERTY_GET, 0, @WhatIfTotalInterest
                    RETURNING, "BSTR", TYPE 8

* WhatIfTotalPayment
          PROPERTY_GET, 0, @WhatIfTotalPayment
                    RETURNING, "BSTR", TYPE 8

* WhatIfMonths
          PROPERTY_GET, 0, @WhatIfMonths
                    RETURNING, "BSTR", TYPE 8

* FireCalc  ()
         EVENT, 520214344, @UserControl1_FireCalc
```

5.5.2.10 Sample controls

Sample .NET controls have been included on the ACUCOBOL-GT distribution media in the /AcuGT/sample directory. Refer to "ReadMeSetup.txt" in each sample directory for an overview of each control and setup instructions. The samples include:

- ACUNET_WEB_SERVICE\WEB_SERVICE, Web Service using remoting objects
- ACUNET_WEB_SERVICE\WebService2, Web Service using ASP.NET
- AmortControl, Composite control
- CompositeControl, Composite control returning event data
- netdb, .NET database sample
- NetToAcuCobol, .NET program calling and retrieving data from an ACUCOBOL-GT program

The amortization control sample consists of an ACUCOBOL-GT program displaying edit boxes, an exit button, and a .NET amortization user control. "AmortControl.dll" fires an event when the control's **Calculate** button is clicked. The COBOL file "AmortControl.acu" receives the event, retrieves updated properties, and displays the results in edit controls. To view the contents of the AmortControl COPY file, the COBOL program, and the GUI screen, refer to the /AcuGT/sample directory.

5.6 Interacting with .NET Web Services

ACUCOBOL-GT supports Web services via .NET client-side control interfaces. This means that an ACUCOBOL-GT program can consume or invoke a Web service via a .NET control. Also, a server-side Web service can consume or invoke an ACUCOBOL-GT server-side program.

Invoking .NET Web Services from COBOL

As a rule, .NET Web services include a C# proxy. This C# proxy contains program code that resolves function calls and connections to the server -side .NET Web service. The proxy is compiled into the client application which uses the proxy to connect to and retrieve information from the server-side .NET Web service.

The .NET samples provided on your ACUCOBOL-GT distribution media contain two Web services projects: one for ASP.NET services and the other for remoting objects. They both utilize HTTP and SOAP.

The ASP.NET example generates a C# proxy that is compiled with the client-side control with which the ACUCOBOL-GT program converses. Refer to the "ReadMeSetup.txt" in each sample directory for an overview and setup of the Web services samples.

1. A .NET programmer writes a .NET Web service using ASP.NET, C#, or VB.NET.

2. This programmer generates a C# proxy from the Web service using the Microsoft WSDL utility that comes with .NET.

The client needs this proxy so that it can connect to the service over the Internet and call methods that reside in the service.

3. The programmer writes a client-side .NET control in C# or VB.NET that accesses the Web service (via the proxy).

4. A COBOL programmer runs the **NETDEFGEN** utility on the control as he would any .NET control. Refer to section 5.5.1 for instructions.

5. The COBOL programmer copies the resulting COPY file into the ACUCOBOL-GT program, then uses the CREATE, DISPLAY, INQUIRE, and MODIFY statements to access the .NET control methods and events.

Invoking COBOL Services from .NET

There are two ways to invoke COBOL-based services from .NET:

1. Via the ACUCOBOL-GT COM Server as described in section 5.4 of this guide.

2. Via the .NET API, “wrunnet.dll” described in section 5.4.2 of this guide.

6 Working with C and C++ Programs

Key Topics

6.1 COBOL and C/C++

In many applications, enterprise systems, and operating systems, C and C++ provide core capabilities and support functions. To help you build integrated applications that make the best use of COBOL programs, ACUCOBOL-GT® provides several robust methods for interoperating with C programs. These technologies work equally well with C++ programs that conform to C calling conventions.

ACUCOBOL-GT provides methods for both calling C from COBOL and calling COBOL from C. Methods for calling C from COBOL include:

- Direct calls to C programs located in Windows dynamic link libraries (DLLs) and UNIX/Linux shared object libraries.
- Direct calls to C programs linked into the ACUCOBOL-GT runtime.
- Interface calls, through a special interface, to C programs linked into the ACUCOBOL-GT runtime.

To support the ability of C programs to call COBOL, ACUCOBOL-GT includes an extensive C Application Programming Interface (API).

All of these facilities are described in detail in this chapter.

Note: ACUCOBOL-GT does not support the Object COBOL elements of the 2002 ISO/IEC COBOL Standard. Neither does ACUCOBOL-GT provide direct object-level interoperability with OO elements and structures.

Interoperating with C to create more powerful applications

Organizations may want to integrate C programs and C-based technologies with COBOL applications for a variety of reasons:

- To use a C application as the *driver* program in a deployment that includes COBOL programs in a transaction server or application server environment
- To boost system-level performance or flexibility of specialized routines

- To take advantage of C-based, C++-based, Visual Basic (VB), or Delphi front ends
- As an alternative to ACUCOBOL-GT methods for implementing and supporting Internet portals, mobile devices, etc.
- For any one of hundreds of other special purposes that enterprise systems architects and ACUCOBOL-GT developers invent

With our C interface technologies, businesses can more easily retain and include valuable, time-proven COBOL programs in their 21st century, multi-technology enterprise systems. ACUCOBOL-GT technologies eliminate the need to re-engineer proven COBOL applications, when they can be easily integrated into next-generation enterprise systems.

6.2 Matching C Data Items

To successfully interoperate with C programs, you need to understand and properly implement data type matching and data exchange between COBOL and C programs.

Note: In addition to the information given here, extensive reference-level information is given in section 5.7.1.8, “USAGE Clause,” in *ACUCOBOL-GT Reference Manual*. If you are working in a Windows environment, see also section , “Data mapping,” in this manual.

Matching simple data items

To match simple C external variables or passed data items, you must choose the appropriate USAGE type. For example, to share an integer variable with a C routine, declare it in COBOL as:

```
77  MY-SHARED-INT  SIGNED-INT, EXTERNAL.
```

In C, this item is then declared as:

```
int my_shared_int;
```

Note: You must also name my_shared_int in the table of external identifiers, as described in the “direct.c” file.

Here is an example of passing an “int” value in a portable fashion to a C routine:

```
77  MY-INT   SIGNED-INT.
MOVE 123 TO MY-INT.
CALL "C-ROUTINE" USING, BY VALUE, MY-INT.
```

The ANSI C routine could then read:

```
void c_routine( int param1 )
{
 printf( "This should be '123': %d\n", param1 );
}
```

Matching complex data items

Matching complex data items such as C structure (struct) items or arrays is more involved. The challenge arises in trying to match the FILLER that may follow the COBOL data items. For example, consider the following group item:

```
01  GROUP-1.
    03  DATA-1  PIC X(5).
    03  INT-1   SIGNED-INT.
```

Assuming that each item is allocated 4 bytes, this would seem to match the following C structure:

```
struct {
    char  data_1[5];
    int   int_1;
} group_1;
```

However, it most likely won’t match up with the default structure packing due to alignment. If you must match complex C data types, you can take one of three approaches:

1. You can use fixed-size COMP-5 COBOL data types that match your C structures. You will then have to change your COBOL code and recompile when you move to a different target environment.

2. You can use the variable-size COBOL data types described in this section and adjust your C structures accordingly. This approach requires a change to your C code when you move to a new environment.

3. You can use the variable-size COBOL data types described in this section and select different target architectures with the "-Dw" compile option. In this scenario, you do not have to change code to go to a new environment; you just have to recompile with a different "-Dw" setting. For example, you could set up two different directories on the development machine, one for "-Dw32" objects, and one for "-Dw64" objects. This approach would provide you with COBOL objects for all currently supported machines.

Note: Most C compilers align structure elements according to their own needs. The automatic synchronization that occurs with variable-size data items matches the most natural alignment boundaries. But the automatically synchronized data items may not match with the alignment rules used by a particular C compiler. As a result, you may find yourself forced to make some code adjustments for a particular machine.

6.3 Calling C Programs From COBOL

ACUCOBOL-GT supports three methods of calling C programs from COBOL:

- Loading C routines packaged in a DLL or UNIX shared object library and then calling the programs directly
- Via ACUCOBOL-GT's *direct method* (without interfacing routines), when the routine is linked into the runtime
- Via ACUCOBOL-GT's *interface method* (using "sub" or "sub85"), when the routine is linked into the runtime

Wherever possible, we recommend that you package routines in a DLL or shared object library. This allows you to call the routines without having to relink the runtime. Note, however, that the method for packaging routines is platform specific.

Alternatively, the *direct method* and the interface method require that you link the external routines into the runtime. The relinking process is described in section 6.3.6, "Relinking the Runtime System." However, on Windows platforms, the interface method can be used with a DLL, in which case the runtime does *not* have to be relinked. See section 6.3.3.1, "The "sub" interface."

All of these methods may be used in combination in the same program.

Note: On UNIX and Linux systems, the runtime also attempts to resolve calls that are in its global symbol space. It does this by calling dlopen(NULL, ...) and adding the return value to the list of shared libraries to search. While not all functions are available, many are, including all of the standard C library functions (those in "libc.a" or "libc.so").

Caution: To successfully interface to C routines, you need to analyze and implement appropriate data type matching and data exchange between COBOL and C programs. For an introduction to C data type matching, see section 6.2, "Matching C Data Items."

6.3.1 Calling C Programs in DLLs or Shared Object Libraries

The simplest way to call a C program from ACUCOBOL-GT is to:

1. Package the routine into a Windows DLL or UNIX/Linux shared object library.
2. Load the DLL or shared object at startup by using one of the following options:
   - Use the "-y" runtime option
   - Use the SHARED_LIBRARY_LIST runtime configuration variable
   - Load the DLL or shared object during program execution with a CALL statement or via the SHARED_LIBRARY_LIST variable and a SET ENVIRONMENT statement.

3. Call the routine with a CALL statement.

For a detailed description of using this method with DLLs, see section 3.3, "Calling DLLs from COBOL."

This section contains detailed information on calling routines in UNIX/Linux shared objects. An important advantage of this method is that it is not necessary to relink the runtime.

In this section, we describe the following methods of loading and calling programs in shared object libraries:

- Loading libraries with the "-y" runtime option
- Loading libraries with the SHARED_LIBRARY_LIST configuration variable
- Loading libraries with the CALL statement
- Calling routines with the CALL statement

Once loaded, any exported function can be called with a COBOL CALL statement.

6.3.1.1 Loading shared libraries with the "-y" runtime option

You can use the "-y" runtime command-line option to specify the name of a shared object library to load when the runtime starts. When the program calls a routine in the library, the runtime applies its standard logic to resolve the CALL (see section 2.9, "Calling Subprograms," in *ACUCOBOL-GT User's Guide*).

For example, to call any function exported by "mylibc.so", start the runtime with the following command:

```
runcbl -y mylibc.so mycobol
```

You must specify a new "-y" for each shared library. The default filename extension for shared libraries is ".so". You can change the default by specifying another extension in the SHARED_LIBRARY_EXTENSION configuration variable. See its entry in Appendix H in *ACUCOBOL-GT Appendices*.

You can use the SHARED_LIBRARY_PREFIX configuration variable to specify a list of directories to search when the runtime attempts to load a shared library. The values are applied if the shared library is specified without path information and the runtime fails to find the object in the default directories. For more information, see the SHARED_LIBRARY_PREFIX entry in Appendix H in *ACUCOBOL-GT Appendices*.

6.3.1.2 Loading shared libraries with the SHARED_LIBRARY_LIST configuration variable

You can use the SHARED_LIBRARY_LIST configuration variable to specify a list of shared libraries to load at program startup, or as the result of a SET ENVIRONMENT statement during program execution. You can set SHARED_LIBRARY_LIST in three ways:

- In the environment
- In the runtime configuration file
- Programmatically with the SET ENVIRONMENT statement

Library names in SHARED_LIBRARY_LIST are delimited by spaces or colons. You can set SHARED_LIBRARY_LIST with the SET ENVIRONMENT statement any number of times during program execution. Each time, the runtime loads the libraries listed. Previously loaded libraries remain loaded.

On some systems, 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, on AIX:

```
SHARED_LIBRARY_LIST=/usr/opt/program/lib/archive.a(lib.o)
```

SHARED_LIBRARY_LIST is similar to the runtime “-y” command-line option except that:

- You do not need to set SHARED_LIBRARY_EXTENSION when you use SHARED_LIBRARY_LIST.
- With SHARED_LIBRARY_LIST, you can mix “.a” and “.so” libraries.

You can use the SHARED_LIBRARY_PREFIX configuration variable to specify a list of directories to search when the runtime attempts to load a shared library. The values are applied if the shared library is specified without path information and the runtime fails to find the object in the default directories. For more information, see the SHARED_LIBRARY_PREFIX entry in Appendix H in *ACUCOBOL-GT Appendices*.

6.3.1.3 Loading shared libraries with the CALL statement

You can load a shared object library programatically with the CALL statement. Once the shared library is loaded, you can call any function exported by the library with a CALL statement (for syntax and rules, see the entry for the CALL verb in section 6.6 in *ACUCOBOL-GT Reference Manual*). For example, the following statement loads "sharedlibrary.so":

```
CALL "sharedlibrary.so"
```

As mentioned earlier, the default filename extension of shared libraries is ".so". You can specify a different extension in the CALL statement, or the default extension can be redefined with the SHARED_LIBRARY_EXTENSION runtime configuration variable. See Appendix H in *ACUCOBOL-GT Appendices*.

If no relative or absolute path is specified, the runtime looks for the library in the current working directory, the locations specified in the operating system's environment variable for shared libraries (e.g., LIBPATH, LD_LIBRARY_PATH, or SHLIB_PATH), and then in the locations specified in the SHARED_LIBRARY_PREFIX configuration variable (see Appendix H in *ACUCOBOL-GT Appendices*).

Like other programs that are loaded with a CALL statement, you can unload a CALLed shared library with a CANCEL statement. When you CANCEL a shared library, you may no longer call its exported functions. Unless the *logical cancels* feature is enabled, all memory used by the program is released. For information about runtime memory management and the logical cancels feature, see section 6.3, "Memory Management," in the *ACUCOBOL-GT User's Guide*.

Note: While most UNIX systems use a binary format that supports calling subroutines in shared libraries exactly as described in this section, SCO UNIX (and possibly some other UNIX systems) uses different binary formats, namely COFF and ELF. On those systems, in order to support calling shared libraries, the objects and executables must be in the ELF format. See the UNIX "man" page for your C compiler on how to create ELF format objects (on some systems, for example, you specify the "-b elf" C compiler option).

6.3.1.4 Calling routines in shared libraries with the CALL statement

After a shared library has been loaded, any of its exported routines can be called. For example:

```
CALL "sharedfunction"
```

Note: By default, the runtime first attempts to find a COBOL program with a matching name. To do this, the runtime can apply several extensions and look in several locations. Only if all of those attempts fail does the runtime attempt to locate the function in the current process or in one of the loaded shared libraries. However, you can use the DYNAMIC_FUNCTION_CALLS configuration variable to specify a list of functions or function name prefixes that the runtime will treat as dynamic functions, which it searches before searching the disk for COBOL programs. See the entry for DYNAMIC_FUNCTION_CALLS in Appendix H in *ACUCOBOL-GT Appendices*.

6.3.2 Calling C Programs via the Direct Method

The direct method allows you to pass arguments to C functions without writing special interfacing routines. Parameters are passed directly to the C function according to the CALL statement that invoked the function, using the standard C calling conventions. This direct method simulates the actions of a native code compiler such as Micro Focus or VAX/COBOL.

Note: The runtime has an internal limit of 30 parameters that it can pass to a C routine called through the direct method; it aborts if more than 30 parameters are passed and suggests ways to work around the abort. You can change this limit by modifying the file "lib/callc.c" and relinking the runtime.

Use the BY VALUE phrase of the CALL statement to pass numeric parameters in a way that is compatible with C calling conventions. Use the BY REFERENCE phrase to pass address parameters.

With the direct method, using BY VALUE causes the actual value to be passed to the routine (as expected). Using NULL causes binary zeros to be passed to the routine (matching the NULL concept in C). With the direct method, passing a zero BY VALUE is much the same as specifying NULL (contrast this with the corresponding note under the interface method described in section 6.3.3, "Calling C Programs via the Interface Method").

When a C function is called by the direct method, its return value is placed in the special register RETURN-CODE.

You should note that the direct method makes it easy to generate memory access violations. You may omit a BY REFERENCE or BY VALUE phrase or forget to terminate strings properly with a NULL value (as required by C).

To use the direct method, add the name of the C function to be called to DIRECTTABLE in the file "direct.c". The table has three columns:

- In the first column, place the name you want to use in the COBOL CALL statement. Use all uppercase characters, and place the name in quotation marks.
- In the second column, place the address of the routine to be called. (You can accomplish this by specifying "FUNC" followed by the exact name of the routine as declared in C. "FUNC" is a macro that generates the appropriate cast of the routine name.)
- In the third column, place the function's return type, which may be one of these seven types:

```
C_int
C_long
C_unsigned
C_pointer
C_short
C_void
C_char
```

You may need to prototype the function if it is not prototyped in an included header file. For example, to call the C function “open” directly, you would include the following code in the “direct.c” file:

```
extern int open();
struct  DIRECTTABLE LIBDIRECT[] = {
    { "OPEN",  FUNC open,  C_int },
    {NULL,     NULL,       0 }
    };
```

After you make the change to “direct.c”, be sure to relink the runtime system.

To use the “open” function in COBOL, you might do something like this:

```
77   FILE-NAME      PIC X(20)
77   FILE-HANDLE    SIGNED-INT

MOVE "myfile" TO FILE-NAME.
INSPECT FILE-NAME REPLACING TRAILING SPACES
     BY LOW-VALUES.
CALL "OPEN" USING BY REFERENCE FILE-NAME BY VALUE 0.
MOVE RETURN-CODE TO FILE-HANDLE.
```

Note: Strings passed to C routines should have LOW-VALUE terminators. Variables that are not passed by address should have the BY VALUE qualifier in COBOL and should be COMP-5 or one of the C data types.

In the example above, FILE-NAME cannot be more than 19 characters, because the 20th, or last, character must be the string terminator.

Up to 20 parameters may be passed via the direct method. If you need to pass more than 20, call our Technical Support and request the routine “callc.c”. The comments within the code explain how to use the “callc.c” routine.

External C variables and system functions can also be linked with COBOL EXTERNAL data items. One function in particular called “ERRNO” can be used to obtain error information from those system functions. The runtime exports the errno variable so that COBOL programs can reference it easily and without requiring a relink.

ERRNO should be shown as defined as:

```
77  ERRNO EXTERNAL SIGNED-INT.
```

Consult your preferred C manual for information on using ERRNO.

6.3.3 Calling C Programs via the Interface Method

The interface method uses special routines that are passed the name of the called function. Use this method when you want to simulate the actions of a non-native system. ACUCOBOL-GT supports two types of interfaces that simulate interfaces available in two versions of RM/COBOL:

- the “sub85” interface, which is source-compatible with C routines written for RM/COBOL-85.
- the “sub” interface, which is compatible with RM/COBOL version 2.

You may use either or both of these interfaces. With the interface method, parameters are passed to the interface routines in a standardized format. The parameters must typically be converted to a format that is usable in C.

Every ACUCOBOL-GT compiler comes with a sample C subroutine interface. The “sub.c” file implements the “sub” interface. The “sub85.c” file implements the “sub85” interface. The “sub.h” file contains some useful definitions, particularly if you are using the “sub85” interface. These files contain extensive comments describing how they are used. They also contain the source to the SYSTEM library routine. You can use these files as the starting point for your code. The two interfaces are described in the following sections.

Note: At run time, the "sub" interface performs a linear search for a called routine. This process can be inefficient when a very large number of "sub" routines are present. If your program calls a large number of C routines, we recommend that you use the "sub85" or "direct" interface.

Also, in the "sub85" and "sub" interface methods, parameters are not passed directly to the C routine. Instead, an array of pointers is passed, and each pointer points to the corresponding parameter (or in the case of "sub85," a description of the parameter). In this case, the notion of BY VALUE has no reasonable definition, because there is no place to put the value. Because of this, the runtime ignores the BY VALUE phrase and passes an address to a copy of the value (essentially treating BY VALUE as BY CONTENT). It must do this because there is no C variable available in which to pass the value. However, specifying NULL does have a reasonable definition: The pointer corresponding to that parameter is set to binary zeros. Therefore, with the interface method, NULL and BY VALUE ZERO have different meanings (contrast this with the corresponding note under the direct method described in section 6.3.2, "Calling C Programs via the Direct Method").

6.3.3.1 The "sub" interface

Every time a CALL statement executes, it calls a C routine called "sub". This routine is passed the name of the called program and its USING parameters. You may modify this routine to recognize the call name that you want to assign to a C subprogram and perform the appropriate code. This routine is contained in the "sub.c" file.

The "sub" routine is passed two arguments: argc and argv. The argv parameter is an array of character pointers. The argc parameter is an integer count of the number of elements in the argv array. The first element in argv points to the call name exactly as it appears in the COBOL CALL statement. This name is terminated with a NULL character. The remaining elements of argv point to each of the USING arguments.

The "sub" function should check to see if the called name is one that should be handled in C. It can do this by comparing "argv[0]" with the desired routine name using the "strcmp" C library routine. If the routine is one that is not handled by a C subroutine, then "sub" should return a negative value. This indicates to **runcbl** that the CALL statement has not been fulfilled and

that it should try to find a COBOL subprogram by that name. If the routine is handled by the "sub" function, then a zero should be returned. In this case, **runcbl** assumes that the CALL statement has been completed and it continues with the next statement. Finally, if "sub" returns a positive value, then **runcbl** executes a STOP RUN, returning the value to the operating system as **runcbl**'s exit value. See "sub.c" for an example of this interface.

When processing a USING parameter, note that the C subroutine must know what internal format the parameter uses. Also note, that in COBOL, literal values are *not* terminated by a NULL character. Thus, you should not treat a passed value as a C string unless the calling program ensures that the passed value is NULL terminated. This can be accomplished in the following fashion:

```
STRING   "literal", LOW-VALUES, DELIMITED BY SIZE
         INTO ITEM-1
CALL NAME-1 USING ITEM-1.
```

Note: The "sub" interface provides compatibility with the RM/COBOL-85 interface. At run time "sub" performs a linear search for a called routine. This can be inefficient if your program calls a large number of C routines. We recommend that you use the "sub85" or "direct" interface.

Placing the "sub" routine in a DLL

In addition to linking the "sub" function directly into the runtime, Windows users may place the "sub" routine into one or more DLL files.

You must specify which routine to use as the "sub" interface routine by setting the DLL_SUB_INTERFACE configuration variable. Then you call the DLL from your COBOL program. The runtime loads the DLL, then checks DLL_SUB_INTERFACE for the name of the routine to use as the "sub" interface routine. For example, the following C program (subdll.c) is the source for the DLL that contains the "sub" interface, called AcuSub in this example:

```
#include <stdio.h>
#include <windows.h>
#include "sub.h"

#define   DllExport             __declspec(dllexport)
#define   CCallingConvention    __cdecl
```

```
DllExport int CcallingConvention
AcuSub( int argc, char *argv[] )
{

    if ( strcmp( argv[0], "MSGBOX" ) == 0 ) {
             MessageBox( NULL, argv[1], NULL, MB_OK );
             return Okay;
    }

    return NotFound;

}   /* AcuSub */

/* end of subdll.c */
```

The following COBOL program (“callsub.cbl”) shows how the DLL is loaded and called:

```
program-id.  test.
working-storage section.

77  message-text          pic x(80).

procedure division.
main-logic.
    display standard window.

* Load DLL and establish "sub" interface

    set environment "dll-sub-interface" to "AcuSub".
    call "subdll".

* Call "MSGBOX", one of the routines handled by "AcuSub"

    move "This is a test message" to message-text.
    inspect message-text replacing trailing spaces by
        low-values.
    call "msgbox" using message-text.

    accept omitted.
    stop run.
```

If DLL_SUB_INTERFACE is blank when the DLL is loaded, no “sub” routine is used in that DLL.

When a CALL statement executes, the “sub” interface routine in each loaded DLL is called, in the order that they were loaded, until one of them returns that it has executed the subroutine. If none of the DLLs returns success, the normal, or linked-in, “sub” routine is called. If that does not return success, then the standard calling sequence resumes. As soon as any routine returns success, the CALL is considered satisfied and no further processing of the CALL is done.

If you CANCEL a DLL with an active “sub” interface, that interface is removed from the list of available interfaces and the DLL is unloaded.

6.3.3.2 The “sub85” interface

The “sub85” interface can be more useful than the “sub” interface because more information about each USING parameter is passed to the C subroutines. However, using the “sub85” interface also involves more programming.

The “sub85.c” file contains a table called LIBTABLE. This table consists of a variable number of entries. Each entry contains a routine name and the name of the corresponding C subroutine. The last entry in this table must be NULL to mark the end of the table. When a CALL statement executes, this table is searched for a matching routine name. If a match is found, the corresponding routine is executed.

When the routine is called, it is passed four parameters: name, num_args, args, and initial (in that order). The name argument is a character pointer to the name that the CALL statement used to access the subroutine. The num_args parameter is an integer that contains the number of USING arguments specified by the CALL statement.

The args parameter is defined as an array of type Argument. This type (a structure) is declared in “sub.h”. For each USING parameter, the corresponding array element contains the following series of fields that describes that parameter:

a_address – a character pointer that points to the first byte of the USING parameter

a_length – a long integer that contains the number of bytes contained in the USING parameter

a_type – a short integer that contains one of the following values depending on the passed data type:

0	Numeric edited
1	Unsigned numeric (DISPLAY)
2	Signed numeric (DISPLAY, trailing separate)
3	Signed numeric (DISPLAY, trailing combined)
4	Signed numeric (DISPLAY, leading separate)
5	Signed numeric (DISPLAY, leading combined)
6	Signed COMP-2
7	Unsigned COMP-2
8	Unsigned COMP-3
9	Signed COMP-3
10	COMP-6
11	Signed binary (COMP-1, COMP-4, COMP-X)
12	Unsigned binary (COMP-1, COMP-4, COMP-X)
13	Signed native (COMP-5, COMP-N, SIGNED-SHORT, SIGNED-INT, SIGNED-LONG)
14	Unsigned native (COMP-5, COMP-N, UNSIGNED-SHORT, UNSIGNED-INT, UNSIGNED-LONG)
15	Floating point (FLOAT, DOUBLE)
16	Alphanumeric
17	Alphanumeric (justified)
18	Alphabetic
19	Alphabetic (justified)
20	Alphanumeric edited
21	Not used
22	Group

Note that setting the u.pass_type member of the Argument structure controls how data is passed back and forth between a program running on the client and a remote COBOL object. Set u.pass_type to “0” to pass data by

reference. Use this when the COBOL application is to read and write data to the variable. Use BY REFERENCE for string literals. Set u.pass_type to "1" to pass data by content. Use this when you want to allow read access to the data item, but the COBOL application should not write to the address. Use BY CONTENT for constants. Set u.pass_type to "2" to pass data by value. Use BY VALUE for numbers.

a_digits – a char that contains the total number of digits in a numeric data type. For non-numeric data types, this value is always zero.

a_scale – a char that contains the power of 10 to multiply the number by. This indicates the location of the decimal point. For example, a value of "-1" indicates that there is one digit to the right of the decimal point. Note that since this is defined as a "char", you may not be able to treat this as a signed value on all machines. The macro "Scale" in "sub.h" converts one of these fields to a signed integer in a machine independent manner.

The final argument to the C subroutine, initial, is set to a non-zero value if the routine is being called for the first time. It is also set to a non-zero value for the first call after a CANCEL statement has been executed for the routine. On any other call, initial is set to zero.

The external variable return-code can be set to any value. The COBOL variable RETURN-CODE will have that value when control returns to the COBOL program. By convention, when a C routine finishes it should return zero if everything was okay. Any other positive return value causes a STOP RUN to be executed, and that value is returned to the operating system as the RETURN-CODE for the run. The C routine should never return a negative value.

6.3.4 Cancelling a CALLed C Program

Shared objects loaded with a CALL statement can be unloaded with a CANCEL statement. For information about the CANCEL statement, see its entry in section 6.6 in *ACUCOBOL-GT Reference Manual*, and section 2.9.2 in *ACUCOBOL-GT User's Guide*.

Shared objects loaded with the "-y" runtime option or via the SHARED_LIBRARY_LIST configuration variable cannot be unloaded.

Note: The CANCEL_ALL_DLLS configuration variable can be used to control whether a CANCEL ALL statement frees shared object libraries and DLLs. See Appendix H in *ACUCOBOL-GT Appendices* for more details.

6.3.5 Managing the Terminal

If a called C routine accepts input from the user or performs screen I/O, those routines need to manage the terminal. This is because ACUCOBOL-GT programs and C programs expect the terminal to be in different initial states.

When the ACUCOBOL-GT runtime initializes the terminal manager on a character-based system, it puts the terminal into a "half-cooked" state. This allows the runtime to optimize certain screen I/O and provides support for auto-terminated fields and other screen niceties. C programs, on the other hand, expect the terminal to be in its default state. Therefore, C routines called from COBOL that interact with the user or display information to the screen must manage the state of the display device. Specifically, the C routine must set the terminal to its default mode before performing screen I/O, and restore the terminal to the "half-cooked" state before returning to COBOL. This is accomplished with the **w_reset_term()** and **w_set_term()** functions. These functions are built into the runtime.

To prepare the terminal for C screen I/O, at the top of your C routine call **w_reset_term()** to place the terminal in its default state.

To prepare the terminal for COBOL screen I/O, before your C routine returns call **w_set_term()** to place the terminal back into the "COBOL" state.

6.3.6 Relinking the Runtime System

You must relink the ACUCOBOL-GT runtime when you:

- Modify the ".c" files located in $ACUCOBOL/lib, including "config85.c", "filetbl.c", and others.

- Call C routines that are not available in DLLs or shared object libraries. This means that you are using the direct method or interface method to call C routines (see section 6.3.2 and section 6.3.3).

The exact procedure for relinking depends on the host system. On most UNIX and Linux systems, you have the option of linking with either the **makerun** script or the **make** utility.

Relinking the runtime requires that you have the appropriate C development system for your host. Required software is specified in the host-specific subsections that follow.

6.3.6.1 Linking on Windows systems

On Windows systems, routines are linked into the ACUCOBOL-GT runtime DLL ("wrun32.dll"), not the **runcbl** program ("wrun32.exe"). Note that both the standard runtime and the Web runtime use the same DLL.

Required software

Microsoft Visual Studio 2005

Linking the standard runtime

To make a new standard runtime for Windows, load the "wrun32.sln" solution file into Visual Studio 2005 and build. By default, "wrun32.sln" is located in the "lib" subfolder of "acugt".

Linking the Alternate Terminal Manager runtime

To make a new Alternate Terminal Manager runtime for Windows, load the "run32.sln" solution file into Visual Studio 2005 and build. By default, "run32.sln" is located in the "lib" subfolder of "acugt".

Linking the console runtime

To make a new console runtime for Windows, load the "crun32.sln" solution file into Visual Studio 2005 and build. By default, "crun32.sln" is located in the "lib" subfolder of "acugt".

6.3.6.2 Linking on UNIX and Linux systems

On most UNIX and Linux systems, ACUCOBOL-GT provides two ways to relink the runtime. The first is the traditional **make** facility for which ACUCOBOL-GT includes a Makefile. The second is with a script named **makerun**. **makerun** allows you to specify the names of libraries and objects on the command line. **make** is immediately familiar to most UNIX and Linux developers. Both facilities are described below.

On UNIX and Linux systems where ***extend***® products are delivered as shared object libraries, relinking the runtime requires relinking the appropriate shared object. In the case of the runtime, this is "libruncbl.so" or "libruncbl.sl". On systems where ***extend*** products are delivered as statically linked executables, the runtime executable is named **runcbl** by default.

Note: After relinking, be sure to move or copy the new object to the required directory. The default location is $ACUCOBOL/lib for shared object libraries, and $ACUCOBOL/bin for static libraries.

Required software

An ANSI 89-compliant C compiler supplied by the vendor of your UNIX system, either included as part of your UNIX system or as an add-on option. Check with your operating system vendor to see if the C compiler is capable of building shared libraries. If it is not, you will need a full version of the compiler to relink the ACUCOBOL library.

Using the **make** utility

On all supported UNIX and Linux platforms, the ACUCOBOL-GT runtime can be relinked with **make** using the Makefile included with the ***extend*** distribution.

For your convenience, two environment variables can be used to specify object files and object libraries to be added to the link line of the Makefile. Set EXTLIBS to a list of libraries, and set EXTOBJS to a list of objects. For example, the following command links the C routines in the object file "myroutines.o" into the ACUCOBOL-GT runtime:

```
EXTOBJS="myroutines.o" make
```

Linking with **make**

To create a new runtime, **cd** to the lib directory and type:

```
"make"
```

Using the **makerun** script

On most UNIX and Linux systems, ACUCOBOL-GT comes with a **makerun** script that can be used to relink the runtime. **makerun** is a UNIX Korn shell script that relinks the runtime executable (**runcbl**) or shared object ("libruncbl.so" or "libruncbl.sl").

The **makerun** command format is:

```
makerun [ldflags] [objects] [libraries]
```

where ldflags, objects, and libraries are optional arguments passed to the "ld" command as described below.

ldflags	A list of any valid "ld" options
objects	A list of additional objects to be linked into the runtime.
libraries	A list of additional libraries to be linked into the runtime. Instead of specifying absolute paths to library files, you can also use "-L" and "-l" to specify path and library names. For example, "/home/joe/lib/libacme.so" can be specified as "-L /home/joe/lib –lacme".

For example, the following command links C routines contained in the object file "myroutines.o" into the ACUCOBOL-GT runtime:

```
makerun myroutines.o
```

You can also execute **makerun** from your own script. For example, the following script links three object files and three libraries into the ACUCOBOL-GT runtime:

```
#!/bin/ksh
OBJDIR=/home/appuser/appobj
OBJECTS="$OBJDIR/a.o \
   $OBJDIR/b.o \
   $OBJDIR/c.o"
LIBDIR=/home/appuser/applib
```

```
LIBRARIES="$LIBDIR/lib1.a \
   $LIBDIR/lib2.a \
   $LIBDIR/lib3.a"
cd /usr/acu/lib
makerun $OBJECTS $LIBRARIES
```

The **makerun** script passes the additional linker options in the EXTLIBS environment variable. Everything specified on the **makerun** command line is passed to the C linker from the Makefile.

Linking with **makerun**

To use the **makerun** script:

1. Copy the files from the lib subdirectory of your ACUCOBOL-GT installation into the directory of your choice and **cd** into that directory.
2. Type the **makerun** command line. For example:

   ```
   makerun myroutines.o
   ```

6.3.6.3 Linking on VMS systems

On VMS systems, a command file is included with ACUCOBOL-GT to relink the runtime system. To invoke it, simply type:

```
"@ACU_LINK"
```

Required software

Digital Equipment Corporation's VAX C or DEC C compiler for VMS

6.3.6.4 Linking on MPE/iX systems

Instructions for linking the runtime under HP MPE/iX are included in Chapter 4 of the book titled *Transitioning to ACUCOBOL-GT.*

Required software

Hewlett Packard's C POSIX Developer's Kit

6.4 Calling COBOL from C

If you are working in an environment that includes programs written in both COBOL and C or C++, refer to information in this section for details on using the ACUCOBOL-GT C API. You will find a reference for all the functions in the C API and instructions for calling COBOL from C.

Note that you can call COBOL from C locally or remotely. You can even have the runtime execute remotely without a COBOL object executing on the client. All you need on the client is a C or C++ program and a runtime. For this to work, you set CODE_PREFIX in the configuration file that you provide with the runtime initialization to point to a remote server hosting your COBOL application. The remote server must also be running AcuConnect. AcuConnect is able to execute a COBOL object remotely and share data with the local runtime. For more information on executing remote COBOL programs with AcuConnect, please refer to the *AcuConnect User's Guide.*

Unless noted otherwise, references to C also apply to C++ code that conforms to the C calling conventions.

6.4.1 Include Files

The declarations, constants, and return values of the C API are defined in two files:

- lib/sub.h
- lib/acusetjmp.h (for the **acusavenv()** and **aculongjmp()** functions)

The "sub.h" file includes a declaration of the routines for all platforms to ensure that you use the correct calling convention with these routines.

6.4.2 Using the C API

The most efficient way to call COBOL from C is to use the C application programming interface (API). This interface can be used by C developers to call COBOL functionality (programs, entry points, etc.) from their C program.

To use the C API, you need to do the following:

1. Install and configure ACUCOBOL-GT runtime. On Windows systems, optionally install AcuBench® if the system is also used for COBOL development.
2. Include “sub.h” and “acusetjmp.h” in any source files that use API calls.
3. Install and configure your C compiler.

6.4.2.1 Using the C API in Windows

You can use the C API for calling ACUCOBOL-GT programs on Windows, as well as on UNIX/Linux systems. Call the ACUCOBOL-GT runtime DLL from other languages in one of two ways.

1. Use the functions **acu_initv()**, **acu_cobol()**, **acu_shutdown()**, and others fully described in section 6.4.3, “Function Reference.”

   These functions offer more control and are designed to be called from C. They are portable between Windows and UNIX; the same C source code calling ACUCOBOL-GT on Windows compiles and runs on UNIX.

2. Use the corresponding functions, **AcuInitialize()**, **AcuCall()**, **AcuShutdown()** and so on, that are described in section 3.2.2, “Calling the Runtime DLL.”

   These functions are designed for use with programming languages like VisualBasic that use Variant type arguments. For example:

   ```
   extern void __stdcall AcuRunMain(char *command_line);
   ```

   In Visual Basic, for example, the declaration is:

   ```
   Declare Function AcuRunMain Lib "wrun32.dll" _
      (ByVal cmdLine As String)
   ```

This has the same functionality and use as **acu_runmain()**. It is optional to call **AcuInitialize** before **AcuRunMain**. For example:

```
if ( AcuInitialize(lpCmdLine) >= 0 )
   AcuRunMain(lpCmdLine);
```

6.4.3 Function Reference

Functions in the C API include the following:

Function	Description
acu_abend()	Performs the ACUCOBOL-GT signal handling logic
acu_cancel()	Simulates the COBOL CANCEL verb from C
acu_cancel_all()	Simulates the COBOL CANCEL ALL verb from C
acu_cobol()	Allows you to call COBOL programs and control how they are executed
acu_initv()	Initializes the runtime with command-line options
aculongjmp()	Cuts the C and COBOL call stacks to the point recorded in the buffer
acu_register_sub()	Registers a COBOL subroutine that calls a C routine, so you can locate the routines in the calling executable file
acu_runmain()	Starts up a COBOL program that will be treated like the main program of the run unit
acusavenv()	Records information about the state of the C and COBOL call stack
acu_shutdown()	Halts the ACUCOBOL-GT runtime
acu_unload()	Removes a cached program from memory
acu_unload_all()	Removes all cached programs from memory

acu_abend()

The **acu_abend()** function allows you to perform the ACUCOBOL-GT signal handling logic in selected programs when the runtime is initialized with the "--no-signal-handlers" option. If you run with "--no-signal-handlers", error conditions that raise signals are not detected automatically by the runtime. **acu_abend()** causes the runtime to output error messages and resets the state of the runtime. This function is designed for use in transaction processing environments that call the ACUCOBOL-GT runtime from a C main program.

Refer to the *ACUCOBOL-GT User's Guide* for more information on "--no-signal-handlers".

Usage

C programs that specify the "--no-signal-handlers" option in the call to **acu_initv()** may call the **acu_abend()** function from their own signal-handling code as follows:

```
void acu_abend( int signal_number )
```

where signal_number is the signal number or "-1" if no signal information is available.

Although you can call **acu_abend()** at any time, we recommend that you call it only if the application was executing COBOL code when the signal occurred; otherwise, the error messages reported by the runtime may be misleading. For example, you do not need to call **acu_abend()** if:

1. Your application calls **acu_cobol()** to execute a COBOL program.
2. The COBOL program exists.
3. After **acu_cobol()** returns, your application executes other non-COBOL processing.

Note: **acu_abend()** does not shut down the runtime. It reports the current COBOL program and execution address. It also reports the COBOL source file and line number if you have compiled with "-Gl" or "-Ga". It then resets the runtime to its initial state. Therefore, even if you have called acu_abend(), you still need to call **acu_shutdown()** to actually shut down the runtime.

acu_cancel()

The **acu_cancel()** function marks a program as "cached" and resets Working Storage to its initial state for the next call. This function is equivalent to the CANCEL verb.

Usage

```
void ASTDCALL acu_cancel(char *name);
```

If the user has specified that the program is not to be cached, then **acu_cancel()** also unloads the program from memory so that a new copy is loaded from disk the next time the program is called.

For more information on using this function, refer to section 6.5.5, "Unloading Programs from Memory."

acu_cancel_all()

The **acu_cancel_all()** function performs an **acu_cancel()** on all "Loaded but inactive" programs. This function is equivalent to the CANCEL ALL verb.

Usage

```
void ASTDCALL acu_cancel_all(void);
```

Application servers such as CICS should call **acu_cancel_all()** after every **acu_cobol()** call returns. Subsequent calls to the same program and its called subprograms will then have Working-Storage items in their initial state.

For more information on using this function, refer to section 6.5.5, "Unloading Programs from Memory."

acu_cobol()

The **acu_cobol()** function allows you to call COBOL programs and control how they are executed. This function is required in order to build and invoke the ACUCOBOL-GT runtime with a transaction processing environment like CICS.

Note: This function deprecates the **cobol()** and **cobol_no_stop()** functions.

To call COBOL subroutines directly from C, you can use the **acu_cobol()** routine anytime *after* **acu_initv()** has been called and *before* **acu_shutdown()** has been called. For more information, see section 6.5.2, "Calling the Runtime From a C Main Program."

Usage

The **acu_cobol()** function has the following prototype:

```
int ASTDCALL acu_cobol(struct ACUCOBOLINFO *data);
```

The struct a_cobol_info structure is defined in lib/sub.h:

```
struct  a_cobol_info
{
    size_t       a_cobol_info_size;
    char         *pgm_name;
    int          num_params;
    Argument     *params;
    int          exit_code;
    const char   *exit_msg;
    int          signal_number;
    int          call_error;
    long         cobol_return_code;
    unsigned     no_stop:1;
    unsigned     cache:1;
    ADM_t        debug_method;
    char         *debug_method_string;
};
typedef struct a_cobol_info ACUCOBOLINFO;
```

The ADM_t type is described in lib/sub.h as an enumeration.

```
typedef enum tag_ACUCOBOL_DEBUG_METHODS
{
    ADM_NONE,
    ADM_XTERM,
    ADM_TERMINAL,
    ADM_THINCLIENT,
} ACUCOBOL_DEBUG_METHOD, ADM_t;
```

The enumeration values describe the debugging method to be used for the program. See the Parameters section below for more information.

Parameters

The a_cobol_info structure has the following parameters. Input variables are set before the call to **acu_cobol()** and output variables set after **acu_cobol()** returns:

a_cobol_info_size	A size_t initialized to sizeof(a_cobol_info) to allow for future expansion *(input)*
pgm_name	A char* that contains the name of the COBOL program to call *(input)*
num_params	An int that contains the number of elements in the params array *(input)*
params	An Argument* that contains an array of arguments sent to the COBOL program *(input)*
exit_code	An int that contains the exit code. Refer to the exit_code values in the Return Values section below *(output)*
exit_msg	A const char* that contains the exit message. Its contents should not be modified. This pointer may be NULL. (o*utput)*
signal_number	An int value of a signal that caused the COBOL_SIGNAL or COBOL_FATAL_ERROR. If the value is non-zero, the error was caused by this signal. *(output)*
call_error	An int value that contains the call error. Refer to the call_error values listed in the Return Values section below *(output)*

cobol_return_code	A long value returned in “exit program.nnn” by a COBOL program called from **acu_cobol()**. The COBOL program can set this value by either setting the return-code or exiting the program. *(output)*
no_stop	An unsigned value that, when set to “1”, causes STOP RUN to be ignored *(input)*
cache	An unsigned value that determines whether the runtime should maintain the program in a memory cache after it has been canceled. This parameter is useful for application servers like CICS that allow each program to be configured as resident or nonresident. *(input)* If cache is FALSE (“0”), **acu_cancel()** removes the program from memory and sets the Working-Storage to its initial state on subsequent calls. If cache is TRUE (“1”), **acu_cancel()** marks the program as “cached” and resets Working-Storage for the next call; the program remains in memory according to the caching rules. For information on managing logical and physical cancels that may affect the behavior of cache, refer also to the LOGICAL_CANCELS configuration variable in Appendix H in *ACUCOBOL-GT Appendices*.
debug_method	An ADM_t type that defines the debugger method to use for the program when **acu_cobol()** is called. The debug_method is deinitialized when the COBOL program returns. *(input)* To enable background debugging, specify one of the following types: • ADM_NONE — For no debugging • ADM_XTERM — Debug using a new xterm • ADM_TERMINAL — Debug using an existing terminal through **runcbl** • ADM_THINCLIENT — Debug using a waiting thin client
Based on the debug_method selected, you may need to also specify debug_method_string:	

debug_method_string	A char* that sets the display setting for the debug_method *(input)* • For ADM_XTERM, set to the *Xservername:displaynumber* of the xterm or set to NULL to allow the xterm to use the default display given by the DISPLAY environment variable. • For ADM_TERMINAL, set the string to the *tty* device on which you will execute **runcbl**. • For ADM_THINCLIENT, set to *client:port* where the *client* is the host on which **acuthin** is executing and *port* is the port on which it is listening. **Note:** The value of debug_method_string overrides the value, if any, in the DISPLAY configuration variable for the xterm.

See section 9.8, "Background Debugging Options," for more information on background debugging.

Return values

The **acu_cobol()** function returns "0" if the call is successful and "-1" if the call failed.

The exit_code returns one of the following values defined in "sub.h":

Value		Description
COBOL_EXIT_PROGRAM	1	The called program finished via an EXIT PROGRAM statement (or equivalent, such as GOBACK).
COBOL_REMOTE_CALL	2	The called program is a remote program being run by AcuConnect®. In this case, the exact reason why the remote program finished is not available.

Value		Description
COBOL_STOP_RUN	3	The run unit halted due to a STOP RUN statement, and the runtime has been configured to return to the caller instead of exiting to the system.
COBOL_CALL_ERROR	4	The called program could not be run and the **acu_cobol()** function has returned "-1". This applies only to programs called directly by **acu_cobol()**. On an error loading a subroutine, **acu_cobol()** returns COBOL_FATAL_ERROR.
COBOL_SIGNAL*	5	The runtime caught a system signal that would normally shut down the runtime, but the runtime has been configured to return to the caller instead. Any error message associated with this signal is returned in exit_msg.
COBOL_FATAL_ERROR*	6	A fatal error has occurred that would normally shut down the runtime, but the runtime has been configured to return to the caller instead. Any message associated with the error is returned in exit_msg.
COBOL_NONFATAL_ERROR	7	An error has occurred that causes the **acu_cobol()** function to return and prevents the runtime from continuing to run the current program only. The runtime remains in a stable state and it is safe to make subsequent calls to **acu_cobol()**.
COBOL_DEBUGGER	8	The user has quit the ACUCOBOL-GT debugger. It is safe to make subsequent calls to **acu_cobol()** after this error occurs.

* After a COBOL_SIGNAL or COBOL_FATAL_ERROR error, the process should not make any further calls to acu_cobol(). If another call is necessary, you must first unload and reload the

runtime. If you are dynamically loading the runtime shared library on UNIX/Linux or runtime DLL on Windows, you can do this using dlclose()/dlopen() or FreeLibrary()/LoadLibrary() respectively. If you are linking directly to the runtime libraries, you will need to exit and restart the current process. Consider wrapping your executable in a shell script or another main() program that checks for a particular exit code and then loops.

The call_error can be one of the following values:

Value		**Description**
CS_SUCCESSFUL	0	Call was successful.
CS_MISSING	1	Program file is missing or inaccessible.
CS_NOT_COBOL	2	Called file is not a COBOL program.
CS_INTERNAL	3	Corrupted program file
CS_MEMORY	4	Inadequate memory is available to load program.
CS_VERSION	5	Unsupported object code version number
CS_RECURSIVE	6	Recursive CALL of a program; program already in use.
CS_EXTERNAL	7	Too many external segments
CS_LARGE_MODEL	8	Large-model program is not supported.
CS_JAPANESE	14	Japanese extensions are not supported.
CS_MULTITHREADED	22	Multithreaded CALL RUN is illegal.
CS_AUTHORIZATION	23	Access denied.
CS_CONNECT_REFUSED	25	Connection refused; user count is exceeded on remote server.
CS_MISMATCHED_CPU	27	Program contains object code for a different processor.

Value		Description
CS_SERIAL_NUMBER	28	Incorrect serial number
CS_USER_COUNT_EXCEEDED	29	Connection refused; user count is exceeded on remote server.
CS_LICENSE	30	License error

acu_initv()

The **acu_initv()** function performs all of the initialization needed to run a COBOL program. This includes loading the COBOL configuration file, initializing the user's station, and initializing each available file system.

Usage

```
int ASTDCALL acu_initv(int argc, char **argv);
```

The **acu_initv()** function also registers various signal handlers so that the runtime can catch certain signals and perform an orderly shutdown or some other function. Using **acu_abend()** with the "--no-signal-handlers" runtime option allows you to control signal handlers based on the signal type.

For more information on **acu_initv()**, see also section 6.5.2.2, "Initializing the runtime."

Parameters

argc – the number of arguments passed in argv, at least "1"

argv – an array of arguments. It must contain at least one argument in argv[0]. This should be a pointer to the name of the executable file, that is, the same as the argv[0] passed to your own main routine.

Note that in some situations, this array of character pointers is needed after **acu_initv()** returns. Therefore, the variable passed in this parameter must remain in scope until the runtime shuts down. You can accomplish this in one of several ways:

1. Make argv a static variable in the C function where it is defined.
2. Make argv a global variable to the C module.

3. Call the functions, **acu_initv()**, **acu_cobol()**, and **acu_shutdown()** from within a single function of your application (or in subfunctions of the C function that calls **acu_initv()**).

Caution: If you do not keep argv in scope for the duration of the runtime's life, you may experience unwanted results.

No other arguments are required. To pass additional arguments, format them in the same manner as a typical **runcbl** command line, where arguments to the runtime are followed by the COBOL program name, and then by any arguments to that COBOL program.

Return values

This function returns the argument number of the COBOL program name in argv (or the value argc if there isn't one). The runtime's own main routine uses this to easily locate the name of the program to load.

aculongjmp()

This **aculongjmp()** routine cuts the C and COBOL call stacks to the point recorded in passed "acujmp_buf". This point must have been recorded by a prior call to both **acusavenv()** and **setjmp()**. Once a state has been recorded, you can only jump to it one time using **aculongjmp()**. After that, it must be recorded again.

Usage

```
void
aculongjmp(acujmp_buf *buffer)
```

Note: Like **acusavenv()**, **aculongjmp()** is intended for use with batch programs or programs written for a transaction processing system and therefore is not recommended for programs that use the ACUCOBOL-GT graphical user interface.

Return values

This routine does not return; instead, it internally calls the C library **longjmp()** routine which transfers control to the point of the prior **setjmp()** call and causes **setjmp()** to return a value of “1”.

Example

```
#include "acusetjmp.h"
acujmp_buf  mark1;

/* A COBOL or 'C' subroutine can CALL "myexception" to transfer
/* control back to the point where "mark1" was recorded.  */

void myexception()
{
    aculongjmp( &mark1 );
}

/* Prototypical 'C' service routine to run a COBOL program with
/* the ability to exit out of the COBOL code via call to
/* 'myexception' */

void myservice()

{
    /* Record our position in the 'C' and COBOL call stacks.
    /* 'Setjmp' will return "0" when executed first. We
    /* call it in a "while" loop so that the location
    /* recording re-executes in case we end
    /* up jumping here via 'aculongjmp' */

    while( setjmp( *acusavenv(&mark1) ) )
    {

        /* If we get here, it’s because 'aculongjmp' jumped
        /* here. Put recovery/cleanup code here.
        /* If you do not want to re-execute the COBOL routine,
        /* add a "return" here.  Otherwise, just fall out of the
        /* while loop to re-execute the COBOL routine */
    }

    /* Setup to call COBOL here and call a COBOL routine */
    /* The COBOL program can jump back to the top of this
    /* routine by calling "myexception" at some point.  This
```

```
    /* will transfer control to the point of the 'setjmp'
    /* call above and cause 'setjmp' to return "1".  */

    struct a_cobol_info cblinfo;
    memset(&cblinfo, 0, sizeof(cblinfo));
    cblinfo.a_cobol_info_size = sizeof(cblinfo);
    cblinfo.pgm_name = "cblprog";
    cblinfo.num_params = 0;
    cblinfo.params = NULL;
    acu_cobol(&cblinfo);
}
```

Restrictions

- As with **setjmp()**, the function that calls **acusavenv()** must not exit before any calls to **aculongjmp()** that use the same buffer. Programs that do this will have undefined (and usually fatal) results.

- **aculongjmp()** works by simulating EXIT PROGRAM statements for each of the COBOL programs being exited. This EXIT PROGRAM has all its normal effects. If a program cannot exit for some reason, the call to **aculongjmp()** fails and the runtime shuts down with a fatal error. Reasons that EXIT PROGRAM can fail include:

    - The program is the main program.

    - The program is currently running an event procedure.

    - The program is the root program of a thread.

    - The program is currently in a Declarative called directly by the BULK-ADDITION feature of Vision.

- Because the caller of **aculongjmp()** must be in the same run unit as the caller of the corresponding **acusavenv()**, avoid using the CHAIN verb.

- The caller of **aculongjmp()** must be in the same COBOL thread as the caller of the corresponding **acusavenv()**. In addition, the effect of **aculongjmp()** on other COBOL threads is undefined. We recommend that you stop any other threads prior to calling **aculongjmp()**.

- After **aculongjmp()** uses a buffer, that buffer is no longer valid and must be re-recorded by a new call to **acusavenv()** and **setjmp()**. This behavior is different from the normal **setjmp()/longjmp()** convention. If converting existing **setjmp()** logic, you can usually accomplish this by replacing the "if" statement that contains the call to **setjmp()** with a "while" statement. For example:

```
if ( setjmp(mybuf) == 1 )
{
    /* longjmp lands here */
}
```

becomes:

```
while ( setjmp(*acusavenv(&myacubuf)) == 1 )
{
    /* aculongjmp lands here */
}
```

acu_register_sub()

The **acu_register_sub()** function allows you to register a subroutine in your C main program.

Usage

```
typedef int (*ACU_SUB_FUNC) (int argc, char **argv);
ACU_SUB_FUNC *acu_register_sub( ACU_SUB_FUNC *pFunc );
```

See section 6.5.3.2, "Calling COBOL subroutines that call C routines," for more information on using this function.

Return values

This function returns the previously registered function, or NULL if no function was registered.

acu_runmain()

The **acu_runmain()** function can be used to start up a COBOL program which is treated like the main program of the run unit.

Usage

```
void ASTDCALL acu_runmain(int argc, char **argv, int pgm_arg);
```

Parameters

argc and **argv** are main-style arguments. Typically, they are the same as the values passed to **acu_initv()**.

program_arg is the argument number (in argv) of the name of the COBOL program.

Example

For an example of using **acu_runmain()**, see section 6.5.3.1, "Starting a COBOL main program."

Return values

When the COBOL program stops, the **acu_runmain()** function halts but does not return.

acusavenv()

The **acusavenv()** routine records information about the state of the COBOL call stack and returns a pointer to a jmp_buf structure that can be passed to **setjmp**. You pass this routine an acujmp_buf structure, which carries the combined C and COBOL stack states.

Usage

```
jmp_buf *
acusavenv(acujmp_buf *buffer)
```

Refer to the entry on **aculongjmp()** for more information on using **acusavenv()**. See your C documentation for information on the **setjmp** and **longjmp** routines, as you need a working understanding of these routines to use **acusavenv()** and **aculongjmp()**.

Note: The **acusavenv()** function is intended for use with batch programs or programs written for a transaction processing system, allowing such systems to restart or recover from a failed service component. Therefore, they are not recommended for programs that use the ACUCOBOL-GT graphical user interface, which are typically event-driven and multithreaded.

acu_shutdown()

This routine performs all the necessary exit handling needed by the runtime. It also ensures that all COBOL files are closed and that the screen is returned to its normal operating state.

Usage

```
void ASTDCALL acu_shutdown(int place_cursor);
```

Note: After **acu_shutdown()** returns, do not call any COBOL subroutines again during this run. In particular, do not try to re-initialize the runtime with another call to **acu_initv()**.

Parameters

If you pass a non-zero value in place_cursor, the runtime places the screen's cursor on the last line of the screen and scrolls the screen one line. This is the normal behavior that you see when you execute a STOP RUN. If you pass a zero in place_cursor, the cursor is not moved and the screen is not changed.

The value of place_cursor overrides the EXIT_CURSOR configuration variable, which also controls the handling of the cursor at shutdown.

Example

For an example of using **acu_shutdown()**, see section 6.5.2.3, "Shutting down the runtime."

acu_unload()

The **acu_unload()** function removes a "cached" program from memory. If "called_programs = 1", **acu_unload()** also unloads called subprograms.

Usage

```
void ASTDCALL acu_unload(char *name, int called_programs);
```

Note: The DYNAMIC_MEMORY_LIMIT configuration variable affects how memory is released by calls to this function. See Appendix H in *ACUCOBOL-GT Appendices* for information on this configuration variable.

For more information on caching programs, see section 6.5.5, "Unloading Programs from Memory."

acu_unload_all()

The **acu_unload_all()** function removes all cached programs from memory.

Usage

```
void ASTDCALL acu_unload_all(void);
```

Note: The DYNAMIC_MEMORY_LIMIT configuration variable affects how memory is released by calls to this function. See Appendix H in *ACUCOBOL-GT Appendices* for information on this configuration variable.

For more information on caching programs, see section 6.5.5, "Unloading Programs from Memory."

6.5 Using the C API: Two Approaches

Using the C API, you can:

- Call ACUCOBOL-GT programs from C

- Call the ACUCOBOL-GT runtime from a C main program.

You can use the same basic method to perform both of these tasks.

6.5.1 Simple Use Case for acu_cobol()

The following example demonstrates a call to the **acu_cobol()** C function:

```
int
main(int argc, char *argv[])
{
    char *initv[3];
    struct a_cobol_info cblinfo;
    initv[0] = argv[0];
    initv[1] = "-c";
    initv[2] = "myconfig";
    acu_initv(3, initv);
    memset(&cblinfo, 0, sizeof(cblinfo));
    cblinfo.a_cobol_info_size = sizeof(cblinfo);
    cblinfo.pgm_name = "MYCBLPGM";
    acu_cobol(&cblinfo);
    acu_shutdown(1);
    return 0;
}
```

This is equivalent to running a program from the command line as:

```
runcbl -c myconfig "MYCBLPGM"
```

If you add the no_stop parameter to the a_cobol_info structure, the COBOL program returns to the caller if the program executes a STOP RUN:

```
struct a_cobol_info cblinfo;

memset(&cblinfo, 0, sizeof(cblinfo));
cblinfo.a_cobol_info_size = sizeof(cblinfo);
cblinfo.pgm_name = "MYCBLPGM";
cblinfo.no_stop = 1;
acu_cobol(&cblinfo);
```

Otherwise, you can remove the no_stop parameter and force the executable to halt:

```
struct a_cobol_info cblinfo;

memset(&cblinfo, 0, sizeof(cblinfo));
cblinfo.a_cobol_info_size = sizeof(cblinfo);
cblinfo.pgm_name = "MYCBLPGM";
acu_cobol(&cblinfo);
```

Note: The **acu_cobol()** function *does* return to the caller when the program performs a STOP RUN if the call is made through the Web runtime or if the runtime is running as an automation server.

6.5.2 Calling the Runtime From a C Main Program

ACUCOBOL-GT uses a runtime system to execute its compiled COBOL programs. This runtime system is a fully linked executable program and, as such, contains its own main routine. The runtime assumes that the first program to run will be a COBOL program.

If, however, you have a "server" program in an OLTP environment that waits for requests from clients and then calls COBOL to handle those requests, you can choose to replace the runtime's own main routine with another.

Note: You do not need to start a COBOL main program to call COBOL subroutines. If you use a C main program, note that certain COBOL verbs can prevent you from returning from a called COBOL subroutine to your C main program.

6.5.2.1 Creating the runtime

The ACUCOBOL-GT runtime library (used to relink the runtime system) comes with its own main routine. The library contains this routine in a separate object module. To create a runtime system that uses your own main routine, you simply add a routine called "main" to one of your C source files and re-create the runtime. Because the linker uses your C object files before searching the library, your main routine is used and the one in the library is not linked.

To relink the runtime system, follow the instructions found in section 6.3.6, "Relinking the Runtime System." The technique for relinking the runtime depends on the machine type.

6.5.2.2 Initializing the runtime

If you are calling the runtime from a C main program, you must initialize the runtime prior to calling any COBOL programs from your C code. To do this, call the following routine:

```
int
acu_initv( argc, argv )
int     argc;
char    *argv[];
```

You pass this routine's arguments in a format identical to those passed to the C **main** function. The COBOL program's name is used during initialization to determine the default title of any GUI-based window, but the program itself is not loaded or executed. Arguments to the COBOL program are not used by initialization and may be omitted. If you omit the COBOL program name, initialization occurs using the default name of "cbl.out".

The **acu_initv()** function calls the user-supplied routines **exam_args** (passing argc and argv) and **Startup** (passing the name of the COBOL program). These routines are normally empty, but may be modified by you. They can be found in the "sub.c" file supplied with the runtime system.

Note: Since argv is an array of character pointers that may be needed after **acu_initv()** returns, keep argv in scope until you shut down the runtime. For more information, see the entry for **acu_initv()** in section 6.4.3, "Function Reference."

Example

The following code might be used by a server routine that intends to use COBOL subroutines to perform various file operations. In this scenario, no screen operations are done from COBOL, so we want to inhibit ACUCOBOL-GT's default screen initialization. To do this, we must pass the

"-b" command-line option. Because the program will perform no screen operations, we do not care about the default window title and, thus, do not need to pass a COBOL program name.

```
int
main( argc, argv )
int     argc;
char    *argv[];
{
     char     *initv[2];
     initv[0] = argv[0];
     initv[1] = "-b";
     acu_initv( 2, initv );
     /* other code follows */
}
```

6.5.2.3 Shutting down the runtime

You can halt the runtime from COBOL by executing a STOP RUN statement and the runtime terminates normally. Note that this action calls the user-supplied C routine called **Shutdown** found in "sub.c".

To halt the runtime from C, call the following routine:

```
void
acu_shutdown( place_cursor )
int     place_cursor;
```

Example

The following C program calls a single COBOL routine once, passing it one argument. After the COBOL routine returns, it cancels that program and halts. While it is not necessary to cancel the program in this example, the code is shown for completeness.

```
#include <stdio.h>
#include "sub.h"
main( argc, argv )
int     argc;
char    *argv[];
    {
    struct a_cobol_info cblinfo;
    Argument cblargs[1];
    acu_initv(argc, argv);
```

```
    cblargs[0].a_address = "Hello World";
    cblargs[0].a_length = 11;
    memset(&cblinfo, 0, sizeof(cblinfo));
    cblinfo.a_cobol_info_size = sizeof(cblinfo);
    cblinfo.pgm_name = "cblprog";
    cblinfo.num_params = 1;
    cblinfo.params = cblargs;
    acu_cobol(&cblinfo);
    acu_cancel( "cblprog" );
    acu_shutdown( 0 );
    exit( 0 );
}
```

A simple example of “cblprog” might be:

```
identification division.
program-id.  cblprog.
data division.
linkage section.
77   hello-world     pic x(11).
procedure division using hello-world.
main-logic.
     display hello-world.
     exit program.
```

6.5.2.4 Notes on COBOL verbs

The following notes describe special considerations for COBOL verbs when you are calling the ACUCOBOL-GT runtime from a C main program.

CALL and CALL RUN

The CALL and CALL RUN verbs work normally. You return from a CALL with EXIT PROGRAM and from a CALL RUN with STOP RUN.

EXIT PROGRAM

The EXIT PROGRAM verb works normally. Use EXIT PROGRAM to return from a COBOL subroutine to the C function that called it.

STOP RUN

A STOP RUN causes the runtime to shut down (except when a STOP RUN returns to a CALL RUN statement). If you need to control the shutdown in C (to perform clean-up for example), do not code any STOP RUN statements. Alternatively, you can place any clean-up code you need in the **acu_shutdown()** routine found in "sub.c". This routine is automatically executed during runtime shutdown. Refer to the no_stop element of the ACUCOBOLINFO structure for more information.

CHAIN and CALL PROGRAM

Both the CHAIN and CALL PROGRAM verbs halt the current run unit and initiate a new run unit. Use these verbs with care as they prevent you from returning to your C main program. The chained-to COBOL program is now treated as the start of a new run unit, essentially meaning that it acts like a main program. Because it is treated like a main program, the EXIT PROGRAM verb in it is ignored. The only way to halt the program is with a STOP RUN (which halts the entire runtime) or with another CHAIN.

If you do execute a CHAIN or CALL PROGRAM, any C routines that are part of the call stack leading to the first COBOL subroutine will be left in place. Although you cannot access these routines, their stack memory remains allocated. On the other hand, any C routines that are on the call stack after the first COBOL subroutine will be removed from the stack.

There is one case where you can still return to your C routine after executing a CHAIN. This case occurs when you use the CALL RUN verb to initiate a new run unit without halting the original. The new run unit can use CHAIN. When the new run unit finally executes a STOP RUN, control returns to the original run unit, which may still return to your C routine via an EXIT PROGRAM.

However, we suggest avoiding these verbs.

6.5.3 Calling COBOL Routines

When calling a COBOL routine, you can either start a COBOL main program (which is what the runtime normally does), or you can just call COBOL subroutines.

6.5.3.1 Starting a COBOL main program

You can start a COBOL main program by calling the **acu_runmain()** routine as follows:

```
void
acu_runmain( argc, argv, program_arg )
int     argc;
char    *argv[];
int     program_arg;
```

This routine initiates a COBOL main program and never returns.

If you omit the COBOL program name (defaulting to “cbl.out”), then program_arg should be the same as argc. Any arguments in argv following program_arg are passed as CHAINING arguments to the COBOL program. Note that program_arg is usually the same as the return value from **acu_initv()**.

Note: The arguments in argv before program_arg are not actually used in **acu_runmain()**. This form is simply used for calling convenience.

6.5.3.2 Calling COBOL subroutines that call C routines

If you are calling COBOL subroutines in Windows, which in turn call C routines, you may want to locate those routines in the calling executable file. You can do this without relinking the runtime by registering your own “sub” function in the C main program. You must add the following declaration to your main program:

```
typedef    int    (*ACU_SUB_FUNC)(int argc, char **argv);
ACU_SUB_FUNC      *acu_register_sub( ACU_SUB_FUNC *pFunc );
```

The routine **acu_register_sub()** registers its argument (pFunc) as an additional “sub” function to call. If pFunc is set to NULL, any existing registration is removed.

The registered function is called just like “sub”, but it is called before “sub”. Note that “sub” is still called if the registered function returns the constant NotFound.

Example

The following C routine calls a COBOL program which, in turn, calls a C program named “MSGBOX”:

```
#include <stdio.h>
#include <windows.h>
#include "sub.h"


extern     int   __stdcall AcuInitialize( char *cmdLine );
extern    void   __stdcall AcuShutdown(void);

// This is the "sub" function that will be registered
int __cdecl local_sub( int argc, char **argv )
{
    if ( strcmp( argv[0], "MSGBOX" ) == 0 ) {
        MessageBox( NULL, argv[1], NULL, MB_OK );
        return Okay;
    }
    return NotFound;
}
// local_sub
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE
    hPrevInstance, LPSTR lpCmdLine, int nCmdShow )
{
    MessageBox( NULL, "Starting in 'C'", NULL, MB_OK );

// Initialize the runtime using the Visual Basic initializer.
// This prevents the runtime from calling exit() when halting.
    AcuInitialize( "" );

// Install our "sub" handler
    acu_register_sub( local_sub );
```

```
// Call the COBOL program. This one takes no parameters
// Assume that this program calls MSGBOX at some point.
// MSGBOX is located in "cblprog".  It will be found
// because "cblprog" got registered with the runtime.
    struct a_cobol_info cblinfo;
    memset(&cblinfo, 0, sizeof(cblinfo));
    cblinfo.a_cobol_info_size = sizeof(cblinfo);
    cblinfo.pgm_name = "cblprog";
    cblinfo.num_params = 0;
    cblinfo.params = cblinfo.no_stop = 1;
    acu_cobol(&cblinfo);

// De-initialize the runtime.
    AcuShutdown();
    MessageBox( NULL, "Back in 'C' - finished", NULL,
        MB_OK );
    return 0;
}
```

Here is the COBOL program called above:

```
program-id. test.
working-storage section.
77  message-text         pic x(80).
procedure division.
main-logic.
    display "In COBOL program".
    display "Calling MSGBOX"
    move "This is a message" to message-text
    inspect message-text replacing trailing spaces by
       low-values
    call "MSGBOX" using message-text
    display "Back from MSGBOX".
    display "Press enter to execute STOP RUN ", no
       advancing.
    accept omitted.
    stop run.
```

6.5.3.3 Canceling a COBOL subroutine

If you wish to simulate the COBOL CANCEL verb from C, you can do so by calling the following routine:

```
void
acu_cancel( program )
char    *program;
```

You pass this routine the name of the program you want to cancel. This name should be identical to the string that you passed to the **acu_cobol()** function to call that program. Canceling a program releases the memory it occupies and resets its internal data to its initial state if you call the program again. Also, any files left open in the program are closed. If *program* does not match the name of any COBOL program in memory, then nothing happens when you call this routine.

6.5.4 Exception Handling

The C library routines **setjmp** and **longjmp** provide functionality that cannot typically be used in the context of the ACUCOBOL-GT runtime. These routines provide "mark" and "goto" points in the call stack of C routines and sometimes provide a convenient exception-handling mechanism.

Because these routines do not know about the COBOL call stack, they cannot be used in cases where C routines call COBOL. However, the **acusavenv()** and **aculongjmp()** functions allow batch programs or programs written for a transaction processing system to restart or recover from a failed service component.

Note: The **acusavenv()** and **aculongjmp()** functions are only usable from C and are not recommended for programs that use the ACUCOBOL-GT graphical user interface, which are typically event-driven and multithreaded.

6.5.5 Unloading Programs from Memory

The ACUCOBOL-GT runtime allows you to cancel and unload programs from memory based on the state of the object module. By default, the CANCEL statements and corresponding C functions, **acu_cancel()** and **acu_cancel_all()**, perform a physical cancel, placing programs in their initial state. The runtime also allows you to enable logical cancels, which may improve performance. For a description of runtime memory management and physical and logical cancels, see section 6.3, "Memory Management," in *ACUCOBOL-GT User's Guide*.

Both logical and physical cancels close open files and ensure that any VALUE clauses are in effect when the program is called again. Physical cancels also release any memory used by the program

The following C functions unload one or all programs from memory based on the state of the object module:

acu_cancel()
acu_cancel_all()
acu_unload()
acu_unload_all()

The **acu_cancel()** and **acu_cancel_all()** functions perform a logical cancel. The **acu_unload()** and **acu_unload_all()** functions perform a physical cancel and force a subsequent call to load the new object file from disk.

Using these functions, an object module can be in one of four states:

- Active
- Loaded but inactive
- Not loaded
- Cached, which means the same as "Loaded but inactive" except that the files and data items are set to their initial state.

Note that the LOGICAL_CANCELS and DYNAMIC_MEMORY_LIMIT configuration variables affect how canceled programs are handled. For information on these variables, see Appendix H in *ACUCOBOL-GT Appendices*.

6.5.6 Signal Handling

The ACUCOBOL-GT runtime normally installs default signal handlers to ensure that the runtime system is cleaned up before terminating. In some environments, the runtime may frequently need to switch back and forth between executing COBOL programs and C programs. Used with the runtime option, "--no-signal-handlers", the **acu_abend()** function allows you to optimize performance by specifying when to preserve signal handlers based on the signal type.

In OLTP environments, for example, the server may need to install and uninstall signal handlers while processing an EXEC CICS statement. In this case, you can enable the ACUCOBOL-GT signal handlers when an application is running COBOL code and enable the previously installed signal handlers when the application is *not* running COBOL programs.

When you initialize the runtime with "--no-signal-handlers", the runtime does not call the ACUCOBOL-GT signal handlers. You can then install your own signal handlers and call the **acu_abend()** function to perform the equivalent of the ACUCOBOL-GT signal handlers when needed.

6.5.6.1 When to call acu_abend()

We recommend calling **acu_abend()** from your own signal handlers only when an application is executing COBOL code when the signal occurs. Otherwise, you can omit the call to **acu_abend()**. For example, under the following circumstances, you do not need to call **acu_abend()**:

1. You made a call to **acu_cobol()** to execute a COBOL program that then exited, and
2. After **acu_cobol()** returns, your application executes some other non-COBOL processing during which a signal is caught.

In fact, calling **acu_abend()** at this point may result in misleading error messages because the signal did not occur when running a COBOL program.

6.5.7 Setting a Debug Method with acu_cobol()

Using the a_cobol_info structure in the **acu_cobol()** function, you can debug COBOL programs from within a transaction processing environment using an xterm window, a terminal, or the thin client. Note that implementing this feature requires support from the vendor of the environment (for example, an OLTP environment provider).

For more information, see the ACUCOBOL-GT supplement for your OLTP environment and section 9.8, "Background Debugging Options."

6.6 Other Interface Paths for COBOL and C

Calling C programs through the runtime and calling ACUCOBOL-GT through the C API are the primary methods for interoperating with C. However, the following three methods may be valuable in special cases:

- Using the C$SOCKET runtime routine, you can establish interprocess communication via sockets. C$SOCKET is a low-level conduit that provides lots of flexibility.
- Using the C$SYSTEM runtime library routine, you can send a C command line to the program's host system.
- Using named pipes, you can pass data between COBOL and C programs. Named pipes are a method for exchanging information between two unrelated processes.

These methods are described in the next three sections.

6.6.1 Connecting with C$SOCKET

If desired, you can facilitate communication between C and COBOL programs at a socket level. ACUCOBOL-GT includes a C$SOCKET library routine, which supports socket-level interprocess communication.

When communicating with sockets between COBOL and C, or any other language, you must:

1. Determine which side is the client and which is the server.
2. Open a listening socket from the server process.
3. Open a connecting socket from the client process.

Of course, because the data format is totally open and undefined, the COBOL and C programmers must agree on a common format.

The following sample code demonstrates this capability:

```
*The following code creates a server socket.
CALL "C$SOCKET" USING AGS-CREATE-SERVER, 8765
   GIVING SOCKET-HANDLE-1.

*The following code waits for a connection.
CALL "C$SOCKET" USING AGS-NEXT-READ, SOCKET-HANDLE-1, TIMEOUT.

*If you have a connection request, accept the connection.
CALL "C$SOCKET" USING AGS-ACCEPT, SOCKET-HANDLE-1.

*Read data from the connecting socket.
CALL "C$SOCKET" USING AGS_READ, SOCKET-HANDLE-2,
   SOCKET-IN, IN-DATA-LENGTH
   GIVING READ-AMOUNT.

*Write outgoing data back to the client socket.
CALL "C$SOCKET" USING AGS-WRITE, SOCKET-HANDLE-2,
   SOCKET-OUT, OUT-DATA-LENGTH.
```

Refer to Appendix I in *ACUCOBOL-GT Appendices* for more information on the C$SOCKET library routine.

6.6.2 Starting a Program with C$SYSTEM

Another way to invoke a C program from COBOL is via the operating system command line using the C$SYSTEM library routine. This routine combines the functionality of SYSTEM and C$RUN.

To call a C program from COBOL via C$SYSTEM, you:

1. Call the C$SYSTEM library routine as described in Appendix I in *ACUCOBOL-GT Appendices*. The C$SYSTEM routine submits the command line to the host operating system as if it were a command keyed in from the terminal.

   Note that you can call C$RUN instead of C$SYSTEM to run the command asynchronously.

2. Pass data from the C program to the COBOL program through a disk file or database.

Refer to Appendix I in *ACUCOBOL-GT Appendices* for complete information on the C$SYSTEM and C$RUN library routines.

6.6.3 Passing Data with Named Pipes

Another way to pass data between COBOL and C programs is through named pipes. Named pipes are a method for exchanging information between two unrelated processes.

Note: To communicate via named pipes, the COBOL and C programs must be on the same host machine.

Technically, named pipes are files with known pathnames. Because a named pipe is associated with a pathname, unrelated processes can open the file to begin communications with one another. Because a C program can open a named pipe just as it would a normal file, no special code is required. By opening the file for reading, a process has access to the reading end of the pipe, and by opening the file for writing, a process has access to the writing end of the pipe. In effect, named pipes allow independent processes to "rendezvous" their I/O streams.

Named pipes can be created in two ways: via the command line or from within a program.

In UNIX, to create a named pipe with the file named "npipe" you can use the following command on the command line:

```
% mkfifo npipe
```

Alternatively, you could create the named pipe from within your program using:

```
int mkfifo(const char *path, mode_t mode)
```

where path is the path of the file and mode_t is the mode (permissions) with which the file should be created.

A named pipe can be opened using the **open( )** system call or the **fopen( )** standard C library function.

As with normal files, if the call succeeds, you get either a file descriptor or a "FILE" structure pointer, depending on how you opened the file. You can then use this information for reading or writing, depending on the parameters you passed to **open( )** or **fopen( )**.

Reading from and writing to a named pipe are very similar to reading from and writing to a normal file. You can use the standard C library function calls **read( )** and **write( )**.

Named pipes can also be used on Windows systems. You create Windows pipes with the **CreateNamedPipe( )** API. You can then use the **CreateFile( )** API to access the other end of the newly created named pipe.

Although named pipes can be very effective for communicating between COBOL and C applications, bear in mind the following:

- Named pipes work only for processes on the same host machine.
- Named pipes can be created only in the local file system of the host.
- Named pipe data is a byte stream, and no record identification exists.
- Named pipes provide only a half-duplex flow of data. They are also known as "fifos" for their method of "first in, first out" communication. To establish full-duplex communication, you must create and manage two pipes, which can be complicated and result in file deadlocks if you are not careful.

6.7 Tracking, Monitoring and Debugging Memory

Several tools help you track, test, and debug memory in ***extend*** products, including the runtime and linked C programs. These facilities are particularly helpful when ACUCOBOL-GT programs interface frequently with C programs.

All products have the ability to monitor, test, and debug memory allocations in three ways. They can:

- Track memory boundaries, so that if any boundaries are corrupted a report is generated. This facility is called *memory bounds checking*.
- Track how much memory is allocated in each of six subsystems. This facility is referred to as *memory tracking*.
- Output a description of each memory allocation, reallocation, and release by file and line. Each action can also include a programmed text message. This facility is called *memory handling descriptions*.

These facilities are described in detail in section 6.4.3 in *ACUCOBOL-GT User's Guide*. How these facilities are accessed via C is described in the following sections.

Note: The memory allocation features can be turned on and off using the appropriate runtime options (described in section 6.4.3 of the *ACUCOBOL-GT User's Guide*), and the available monitoring, testing, and debugging features are determined by the state of the parallel runtime option. For example, when boundary checking is turned off, memory boundaries are not set and executing **dbg_test()** does not test the boundaries of blocks allocated. Similarly, turning off memory tracking will stop keeping track of newly allocated memory, and the numbers reported will not reflect blocks allocated while the feature is off.

6.7.1 Memory Debugging via C

If you are using and writing C routines, you can access an abundance of memory tracking information.

If you allocate memory and want to tie into the ACUCOBOL-GT memory debugging facilities, you can use the following functions to allocate, reallocate, and free memory.

```
void *Amalloc(size_t size, int subsys, const char *file, long line,
             const char *call_desc);

void *Arealloc(void *ptr, size_t newsize, const char *file, long line,
             const char *call_desc);

void  Afree(void *ptr, const char *file, long line,
             const char *call_desc);
```

size, *ptr*, and *newsize* are equivalent to the same parameters passed to **malloc()**, **realloc()** and **free()**.

subsys is one of: M_OVERHEAD (0), M_PROGRAMS (1), M_FILES (2), M_WINDOWS (3), M_DYNAMIC (4), M_NOT_TRACKED (5). It can be modified by ORing with the value M_NO_ZERO (0x8000) in order for the memory to not be set to low-values before being returned to the application. This parameter is used only if the memory tracking debugging feature is turned on. The first five values (0 - 4) are what the runtime sends to the debugger to report memory usage when the “U” command is specified.

file and *line* are the filename and line number of the operation, and are best set by the standard C macros __FILE__ and __LINE__. These values are used only if the memory description debugging feature is turned on.

description is a text description of the memory and is printed in the memory description report.

call_desc is a text description of the call. If null, it defaults to M_alloc(size) for **Amalloc**, M_realloc(ptr, newsize) for **Arealloc**, and M_free(ptr) for **Afree**. This text is printed in the memory description report.

Note that you are not required to use these functions to allocate and free memory used by your C routines. However, it is essential that memory allocated by **Amalloc()** or **Arealloc()** not be freed by the system **free()** function. It must be freed with **Afree()**. Similarly, memory allocated by **malloc()**, **calloc()**, or **realloc()** must be freed with the system **free()** function, and not the ACUCOBOL-GT **Afree()** function. Disregarding these rules will almost certainly result in a memory access violation.

6.7.2 Turning Memory Debugging Features On and Off

To use the memory debugging functions from C, you must use the following functions:

```
Aget_memory_debug_flag()
```

This function gets the current state of the memory debugging feature. It returns an int.

```
Aset_memory_debug_flag(int flag)
```

This function sets the current state of the memory debugging feature. The int values are described as follows:

The **memory description value**, as used by the runtime, is the low six bits (though currently only three bits are used). In other words, the memory description value can be retrieved by ANDing the returned flag with 0x003F.

The **memory tracking feature** is turned on if (flag & 0x0040) is non-zero.

The **memory bounds checking feature** is turned on if (flag & 0x0080) is non-zero.

You can turn on memory handling descriptions by calling the function:

```
void Aset_mem_file_name(const char *filename);
```

Note that if memory descriptions are turned on but a filename is not given, memory descriptions are not reported.

6.7.3 Reporting Allocated Blocks

A list of all allocated blocks can be written to the memory description file by calling the function:

```
void Amem_dump(int final);
```

If final is non-zero, it is considered the final memory dump. The memory description file is closed, and the memory description feature is turned off.

6.7.4 Getting Memory Amounts

To get a report of how much memory each subsystem has allocated, you can call:

```
size_t Aget_mem_used(int subsys);
```

subsys must be a value from 0 – 5. The function returns the amount of memory allocated for that subsystem. The subsystem is the same as the subsys passed to **Amalloc()**.

6.7.5 Testing Memory Boundaries

When the boundary checking feature is turned on, the **dbg_test()** function can be used to check that all memory boundaries are still intact. **dbg_test()** has the following prototype:

```
void dbg_test(const char *description);
```

This checks all memory blocks allocated while boundary checking is on to verify that memory boundaries have not been violated. It checks both the beginnings and the endings of all such blocks.

Note: Memory bounds testing has a significant impact on performance and should only be enabled to assist in program testing and debugging.

---

7 Deploying ACUCOBOL-GT Applications on the Web

Key Topics

7.1 COBOL on the Web

We offer a variety of solutions for deploying COBOL applications on the Internet. Some allow you to make your COBOL programs and data accessible on the Web from popular Internet browsers. Others allow you to harness the Internet in a more secure TCP/IP networking configuration.

Following are some of the Web-based solutions available to our customers:

- Web thin client – You can add the ACUCOBOL-GT® Web Thin Client to your Web page so that when users visit your site, the thin client downloads and installs on their machines and automatically launches your application on the server. In thin client architectures, the application logic runs on the server. Only the user interface displays on the client.

- Web runtime – You can add the ACUCOBOL-GT Web Runtime to your Web page so that when users visit that page, the runtime downloads and installs on their machines and automatically launches your application locally.

- COBOL CGI interface – You can create a Web interface to your COBOL application and allow users to interact with pages on your Web site via an HTTP browser or mobile device using ACUCOBOL-GT's Common Gateway Interface (CGI) extensions.

- Internet helper application – If your users already have a licensed copy of the ACUCOBOL-GT runtime on their machine, they can gain access to your applications on the Web by setting up the runtime as an Internet helper application or viewer inside their browser. When they click a link on your Web site, the browser knows to associate the application with the ACUCOBOL-GT runtime.

- Web browsing from COBOL – ACUCOBOL-GT includes a Web browser control that lets you add a variety of Internet features to your COBOL program. With this control, your programs can support Web browsing; display HTML pages; invoke e-mail, telnet, and FTP services; and more. You can even give your program Windows print, file, and clipboard capabilities.

- COBOL Web services – Using our Java and .NET interfaces, you can expose your COBOL applications as Web services for use in Web services deployments.

This chapter provides an overview of these approaches. These options are described in detail in *A Programmer's Guide to the Internet*, available on your product distribution media.

7.2 Web Thin Client

If you want Windows users to launch applications from your Web site and have the applications run exclusively on the remote server, you can use the ACUCOBOL-GT Web Thin Client. In this scenario, end users simply visit your Web site. The Web browser searches for the Web thin client on their machines. If successful, it launches the program on the server. If it cannot locate the Web thin client, it provides the software automatically with users' permission. It then invokes the server application transparently and "projects" the user interface back onto the client. The Web thin client is an ActiveX version of our thin client solution.

Alternatively, end users can install the standard ACUCOBOL-GT Thin Client on their local machine. They can install it from any ACUCOBOL-GT media or, subject to appropriate licensing agreements, you can distribute it on your Web site so that end users can download and install it from there. Using an Active Server Page (ASP), Java Server Page (JSP), Visual Basic, or perl script, you can automate the download and install process for users if you like. Alternatively, they can download the thin client at no cost from the Acucorp Web site, http://www.acucorp.com/support/downloads/. Once they have the thin client installed, they can visit your Web site and click a link to invoke your application.

Thin client users always have the option of executing the **acuthin** command with an Internet server or IP address as part of the command parameters. **acuthin** can launch programs on any server in a TCP/IP network, including the Internet. The only components required on the client in this case are the thin client software and an Internet connection. Users don't even need to have a Web browser.

With any of these thin client options, all application processing is performed on the server. Usually, data access is considered local because the data resides on the same server machine as the application. If you want to keep data on a different server in a multi-tiered configuration, you can combine the thin client with our AcuServer® technology. Please note that although the thin client supports only Windows clients, it gives access to both Windows and UNIX servers running the AcuConnect® application server software.

7.3 COBOL CGI Interface

Perhaps you want customers or users to run your applications by clicking a link on your Web site, but you don't want to require anything special of the user's machine (for instance, the presence of any ACUCOBOL-GT runtime, be it a standard, thin client, or Web runtime). In this case, you can create a new interface to your application using a markup language such as HTML, WML, or XML. With a Web interface, your application can be interpreted directly by the user's HTTP browser or mobile device, and the processing logic can remain in COBOL on the Web server.

In this scenario, you create your Web interface using one of many popular authoring tools. Then you write a CGI program that can read CGI variables submitted by the client to the server. This program can launch your COBOL application or it can be a COBOL program itself. You can write it using ACUCOBOL-GT or any other language you choose. If you write the program in ACUCOBOL-GT, you do not have to UNSTRING the CGI variables in the program, because ACUCOBOL-GT takes care of this for you through special "IS EXTERNAL-FORM" syntax.

By default, your CGI program reads and writes HTML content for use in standard HTTP browsers and mobile devices. But using configuration variables, you can associate your program with the MIME content type for WML so that data can be displayed on WML-based devices as well.

Once you build a Web front end and write a CGI program, your customers or users can then visit your Web site and gain instant access to your COBOL application running on the server.

Note that CGI programs are inherently stateless—that is, they do not store information about previous browser actions. If you require a persistent connection to the browser, you can achieve this by adding pointers and cookies to your CGI program, or you may choose a different method.

This option runs on any platform where ACUCOBOL-GT runs, but it also requires the most coding. You can employ the CGI method wherever a user interface via DISPLAY/ACCEPT statements is *not* used. This includes batch processes, processes that use socket routines to communicate with an external UI, BEA® Tuxedo® processes, and processes launched via AcuConnect in distributed processing mode, to name a few.

7.4 Web Runtime

Another way to give end users access to your applications on the Web is to provide runtime services through the ACUCOBOL-GT Web Runtime.

Using this approach, you set up a Web site and embed a link to your ACUCOBOL-GT application. You embed the Web runtime in the link as well by designating the URL of Acucorp's download center in your HTML coding. Users can then visit your site and click a link to launch the program. If the Web browser detects that users do not yet have a runtime installed on their machine, it automates the install process, with the users' permission, and then launches the COBOL program locally.

The Web runtime is available only on supported Windows machines, but it gives users access to programs or data hosted on other platforms using AcuServer and AcuConnect.

The ACUCOBOL-GT Web Runtime is geared for Microsoft Internet Explorer environments. It relies on ActiveX technology and does not run on any current versions of Netscape.

7.5 Internet Helper Application

If your users already have a licensed copy of the ACUCOBOL-GT runtime on their machine and they want to be able to access COBOL applications on a Web site, one way to do this is to set up their runtime as an Internet helper application or viewer inside their browser. *Helper application* and *viewer* are browser terms referring to user-based software that can read and process files of a given type. Netscape uses the term "helper application" to refer to such software. Microsoft Internet Explorer uses the term "viewer."

Because the ACUCOBOL-GT runtime has the ability to read and process COBOL objects, it allows users to run COBOL programs that they encounter when browsing your Web page. The main difference between this and a standard runtime configuration is that your COBOL object files are on the Web server and transmitted to the client machine via HTTP.

Keep in mind that the helper application/viewer is a full-featured runtime; it has full access to your users' machines, including system calls, memory, disk and network access. Therefore, it should be used only with programs from trusted sources, or in conjunction with the Internet security you have in place.

7.6 Web Browsing from COBOL

The ACUCOBOL-GT Development System includes an Internet-related Web browser graphical control. The Web browser control lets you:

- Facilitate seamless Web browsing from your COBOL application.
- Display Web pages containing HTML, scripting, ActiveX controls, and Java applet content.
- Display HTML pages distributed with your COBOL application.
- Include a variety of graphical and multimedia file types in your COBOL application.
- Invoke e-mail, telnet, and FTP services from your COBOL application.

- Display word processing, accounting, or presentation documents from your COBOL application.
- Display Windows objects such as folders and files from your COBOL application.
- Display Windows dialog boxes such as “Print,” “Print Preview,” and “Page Setup,” allowing users to print the contents delivered by the control.
- Display the Windows “Save As” dialog box allowing the user to save the current control content to a file.
- Perform “Select All” and “Copy” clipboard operations.

ACUCOBOL-GT’s Web browser control is used in the same manner as other graphical controls in ACUCOBOL-GT, except it opens a resource such as an HTML page, graphical image, video, audio, e-mail program, file folder, or any other resource that a Web browser can open. When you include the Web browser control in your source code, your application launches Microsoft Internet Explorer on your user’s machine and displays the resource you specified.

For the control to work, your users must have Microsoft Internet Explorer Version 4.0 or later on their machine.

7.7 COBOL Web Services

With ***extend*** technologies, you can expose your COBOL applications as Web services for use in Web services deployments. Java technologies are discussed in Chapter 2 of this guide. .NET technologies are discussed in Chapter 5.

Providing Web Services

For those using the J2EE platform, we offer a native Java interface that encapsulates the ACUCOBOL-GT runtime in a JAR file. By invoking the Java class contained in this archive, a Web service running on J2EE can start the runtime and run your COBOL program.

For those using .NET, we offer a .NET interface that presents the runtime as a dynamic link library. By invoking the .NET class contained in this DLL, a Web service running on .NET can start the runtime and run your COBOL program.

Consuming Web Services

To enable a COBOL program to consume a Web service, we provide the C$JAVA library routine and the NETDEFGEN utility.

If the Web service is running on J2EE, the Java programmer packages the WSDL from the Web service in a WAR file with all the necessary resources including a Java client proxy, and the COBOL programmer invokes the service by calling the C$JAVA routine and naming the proxy as a USING parameter.

If the Web service is running under .NET, the .NET programmer generates a client proxy from the Web service and incorporates that proxy into a .NET assembly. The COBOL programmer then uses NETDEFGEN to create a COBOL COPY file for the proxy. The programmer then copies the COPY file into the COBOL program and uses the CREATE, DISPLAY, INQUIRE, and MODIFY statements to access the Web service methods and events.

7.8 Other Internet Solutions

Not all Internet deployments involve Web browsers and HTTP. Our file server, file interface, and/or application server technologies support IP addresses and URL syntax so that you may leverage the Internet in virtual LAN/WAN configurations and broaden the reach of your legacy assets in a highly controlled setting.

- AcuServer can be used to provide access to Vision data over the Internet.
- AcuConnect can be used to provide access to server-resident COBOL programs over the Internet, even programs that are distributed across a number of different servers.

- AcuXDBC® can be used to provide access to ODBC data over the Internet. AcuXDBC is combined with AcuXDBC Server for remote processing of SQL requests.
- Acu4GL® and AcuSQL® can be used to provide access to relational databases over the Internet.
- AcuXML can be combined with AcuServer to provide access to XML documents over the Internet.

All ***extend*** technologies are designed to work in TCP/IP networks. Because the Internet is just a large TCP/IP network, you can use these same proven technologies in Internet deployments.

8 Accessing ACUCOBOL-GT Applications from Mobile Devices

Key Topics

8.1 Overview of Mobile Computing

What do we mean when we use the term *mobile computing*? Is it sending a short text message via a mobile phone? Having wireless access to the Internet from a PDA? Connecting to the office computer system from a hotel room using a laptop? Using a barcode reader to collect data in the field? Mobile computing is all these things, and more. The exact definition varies with individual needs.

We can consider the concept of mobile computing as *computers on the road*: a device/system combination that you use to conduct business at a location removed from your office desktop machine. The remote location can be a hotel room or branch office, or it can be your office at home. Your computing needs can range from a simple query or "look-up" function from a handheld device to obtain important information, to a portable office system with real-time communication with the home office. For example, a field worker might use a handheld device to collect data, such as water or power usage or the current inventory level for a particular item. Or after finishing one job, a plumber might use a handheld device to identify his or her next service stop.

It's also possible that "mobile" employees aren't leaving your office building at all. So-called "campus workers" can use a notebook or tablet to continue working when they are not actually sitting at their desks. They can, for example, be working in the lunch room or taking notes in a meeting.

We see that mobile computing comprises an array of activities and devices. This chapter explores some of the concepts you'll need to know about as you consider how to take advantage of mobile systems. We'll start with definitions of some basic mobile technology terms and a description of the existing infrastructure. Then we'll discuss current trends in mobile platforms and some mobile system design issues. Finally, we describe a sample mobile system and some helpful methods for achieving mobile computing.

8.2 Key Mobile Terminology

Mobile computing today is associated with an interesting lexicon that seems to expand on a daily basis. WAP, WML, TCP/IP, 3G, WiFi—what do these acronyms mean and how do they apply to the business of mobile computing?

8.2.1 Languages

Hypertext Markup Language (HTML) is the familiar language used to display Web pages. A subset of HTML, Wireless Markup Language (WML) is specifically designed to present Web-based information on small handheld devices like mobile phones.

Extensible Markup Language (XML) is a language for documents that contain structured information. You can define custom tags and the structural relationships between them based on the content of your document. Because an XML document contains information about itself, it is an excellent vehicle for transporting data from one location to another, for example, between applications or between organizations.

8.2.2 Protocols

Several sets of protocols determine how voice and data are transmitted over short and long distances. They include protocols for software, hardware, and networks.

The standard protocol governing World Wide Web communications is the well-known Hypertext Transfer Protocol (HTTP), which supports HTML for Web page display. For wireless devices like mobile phones, the Wireless Application Protocol (WAP) controls your access to information on the Internet. WAP supports WML for display.

Bluetooth technology (named for the tenth century Danish king who unified Denmark) allows wireless, radio-based communication between devices. Bluetooth is a general-purpose standard that targets communication between technical “gadgets.” For example, it could connect your laptop to a major kitchen appliance like your refrigerator, if you so desired.

Transmission control protocol/Internet protocol (TCP/IP) is the layered network protocol that has become the global standard for system-to-system communications. It is designed to allow dissimilar systems to transmit data to one another.

8.2.3 Wireless Communication Standards

Mobile communications are managed by wireless networks, which have system standards that govern the communications process. Each succeeding generation of wireless standards has been developed to increase bandwidth and speed of data transmission as well as the quality of voice transmission. In this section, we present some background information about how these standards have evolved.

8.2.3.1 The past and the present

Nordic Mobile Telephone (NMT) and the Advanced Mobile Phone System (AMPS) were among the first standards for analog mobile phone systems. Both are first-generation, or 1G, technology standards.

GSM, named for the group that developed it (Groupe Speciale Mobile), is another wireless standard for mobile communication. Originally developed as a common digital wireless standard for Europe, GSM is second-generation, or 2G, technology. The GSM standard is based on Time Division Multiple Access (TDMA), which divides a radio frequency into time slots and then allocates each slot to a user.

Whereas TDMA has been the standard used in Europe, Code Division Multiple Access (CDMA) is the standard used in the United States. CDMA uses a digital encoding system to spread the signal for a call across a range of frequencies. This technique allows more users to share the network simultaneously.

The General Packet Radio Service (GPRS) uses GSM to handle data transmission. This standard is known as 2.5G technology. GPRS is a packet-switching technology, which means that users are always connected.

The 2.5G technology is currently the most widely used standard for mobile communications. Mobile devices using this standard can provide voice and data transmission, along with Web browsing capabilities. These types of phones can also transmit e-mail. Networks using the 2.5G standard have been implemented worldwide.

WiFi, short for *Wi*reless *Fi*delity, is more commonly known as the 802.11b standard for wireless communication. WiFi provides short-range wireless connectivity (approximately 150 feet) between devices, primarily for computer networks. So-called "WiFi zones" may be located in airports, hotels, and even warehouses in order to facilitate wireless communications.

8.2.3.2 The future

The Universal Mobile Telecommunications System (UMTS), also known as 3G, is a specification for the third generation of mobile communications. It is based on an enhanced version of CDMA called Wideband CDMA (W-CDMA).

Perceived as the successor to 2.5G technology, the 3G systems are intended to provide higher data transfer rates than GPRS and are based on packet-switching networks that are "always on." Unlike GPRS, the UMTS standard provides a dynamic connection, so that the bandwidth varies depending on the requirements, providing a more efficient utilization of the network.

An enhancement to GSM networks is EDGE, which stands for Enhanced Data rates for Global Evolution. This technology is complementary to the GPRS network upgrade to GSM and might be considered a "bridge" between the 2.5G GPRS and future 3G technologies. EDGE allows increased high-speed data transfer capabilities.

CDMA2000 wireless technologies build on CDMA to provide improved transfer rates for users. Evolution Data Optimized (EV-DO) technology is expected to provide high-speed data transmission, whereas Evolution Data Voice (EV-DV) would allow increased network capacity for voice transmissions and higher speeds for data transfers.

For enhanced wireless high-speed connectivity technology, the next step may be WiMAX. WiMAX will operate like WiFi, but is expected to provide high-speed connectivity from 1-10 miles, a much larger range than that in your neighborhood WiFi hot spot. WiMAX could be the key to Internet connectivity that covers entire communities without the huge investment in cable or phone networks.

8.2.3.3 3G status

A mobile device that uses the 3G standard with its higher frequency and larger bandwidth can be expected to provide users with more capabilities and much faster speeds for their mobile communications. The first applications available for 3G devices have been primarily entertainment-oriented rather than for business purposes. With phones that use 3G technology, users can access video services covering the latest news and sporting events and make video phone calls. At this time, however, 3G networks are implemented only on a limited basis worldwide, and compatible mobile devices are expensive and in short supply. Recent trends indicate that Europe may adopt a different 3G standard (UMTS) than Asia and North America (CDMA, EV-DO), setting the stage for global network incompatibilities. Expected high throughput rates have not been proved in reality, but the widespread rollout of this type of network is still eagerly anticipated by users who want faster, feature-rich mobile communications.

8.3 Mobile Platform Trends

With the proliferation of devices and operating systems in the market today, possible combinations for mobile systems seem limitless. Today, mobile computing technologies are evolving at an incredibly fast pace. From handheld devices like Personal Digital Assistants (PDAs) and intelligent mobile phones to solutions involving laptop computers and high-speed, wireless Internet connections—how do you begin to choose the solution that best fits your needs?

Research into current industry offerings reveals three major platforms competing for dominance in the mobile device market—Pocket PC, Palm, and Symbian. Each platform, in turn, uses its own operating system—Windows Mobile OS, Palm OS, and Symbian OS, respectively. Each has its own unique development and runtime environment. Because none of these players has achieved industry dominance and standards in this area are lacking, your choice among these three options is not clear-cut. And if you want a single application to run on all three devices/operating systems, you need third-party middleware to accomplish the interoperability.

But the basic question is, can a device running a non-COBOL front-end application communicate with your back-end COBOL program? The fact is, with a carefully designed system architecture for your information system, any one of these platforms/operating systems can be a viable front end to your legacy COBOL application. If you already have a particular preference for mobile device and operating system, the choice is easier. Odds are that it can connect to and run your COBOL application.

8.4 Mobile System Design Issues

Several important elements must be carefully researched during the design phase of a mobile computing system. User interface functions, security issues, degree of connectivity, and record-locking requirements are some topics that should be considered part of a complete system design. The following sections provide more detail in these areas.

8.4.1 User Interface

User interface design for a mobile device involves several issues. You should consider exactly what tasks your users need to accomplish. Are they monitoring inventory, recording sales, or inquiring about a customer's current credit status? Because the screen size is much smaller than a standard desktop computer, you cannot simply transfer your application from the desktop to a mobile device. Concentrating on specific user tasks can help you narrow down the information your screen should contain.

Ease of use is an important consideration in interface design. For example, keyboards on mobile devices are quite small and may be very difficult to use in the field. You might consider allowing users to choose options via check boxes and radio buttons in the device display area. If you still require an on-screen keyboard, be aware that it occupies valuable screen space.

Application operating rules are also different on mobile devices compared with desktop operating systems. For example, a PDA operating system allows only one application to be active at any one time. The user controls which application is active, and the system makes the previously active program dormant.

8.4.2 Security

When you consider security for wireless operations, you need to cover three main areas—device, application, and communications security.

Any valuable company data that resides on your mobile device must be protected. The small size of a mobile device increases the chances of its being misplaced or stolen. Device security can be achieved via the use of power-on authentication functions that are usually available on PDA devices. This password feature can protect data by ensuring that only the device's owner can access data. Another possibility is password protection for such mobile device functions as the infrared communications feature. You might decide to encrypt the data on your mobile device so it is unreadable to anyone who does not have an appropriate algorithm and key.

As on desktop systems, data on a mobile device can be corrupted. Although less common than on desktop systems, viruses can still attack and compromise data on a wireless device. Third-party anti-virus software can provide needed protection for the data stored on your mobile device.

Securing your application can involve a variety of user password authentication and permissions protections. Your application can require a password or other user authentication before it executes, or the database may require a password for access.

Security for your communications is essential for wireless systems. The nature of wireless networks can leave them more vulnerable to attack from external sources than wired solutions. Wireless access points reside in an unlicensed frequency band that can be easily breached by wireless hackers. Secure Sockets Layer (SSL), a secure protocol based on TCP/IP, uses key encryption and digital certificates to encrypt communication. A Virtual Private Network (VPN) can also provide authentication and encryption features to protect communications.

8.4.3 Degree of Connectivity

Another question is how your users are connected to the network. Does your application require users to be always connected, mostly not connected, or somewhere in between?

Some applications may require a constant connection to the network in order to function properly. This architecture is sometimes called *thin client*. Depending on a constant wireless connection can be risky. The technology is still not completely reliable, with long distances or other obstacles that can interrupt an important data transfer.

Will your users collect data in the field on a mobile device and be connected to the network only when they want to download the data to a server? In this situation, a device needs to allow local data storage until it can be "cradle-connected" to the server.

A mobile device can also store data locally for an application that may or may not be connected to the network at any point in time. With this *smart client* architecture, the application can still function and a user can access data with or without a network connection.

8.4.4 Record Locking

Record-locking issues require careful consideration in wireless operations. Business applications rely heavily on record locking to ensure consistent data quality and to guard against conflicting updates by multiple users, a concept known as *concurrency.* Unreliable wireless connections can complicate record-locking issues. Mobile solutions may generally support three types of concurrency—destructive, optimistic, or pessimistic concurrency.

With destructive concurrency, the last update to a record "wins." The application does not attempt to settle any update conflicts, giving it no control over data updates. This situation can leave your data in an inconsistent state if updates aren't completed in the correct order.

Optimistic concurrency assumes that no data update conflicts exist. When a user updates a data record, the original record is checked to see if it has changed since the user accessed the record. If a conflict is detected, the user is given the option to overwrite the record. If no change is detected, the record is overwritten.

Pessimistic concurrency is probably the most familiar version of record locking. This concept assumes a high probability of data update conflict. A record selected for update remains locked until the user writes the update to the database. Accomplishing this form of concurrency with desktop network

systems is relatively straightforward. For a mobile application, the process is a bit more complicated. In a mobile environment, we accomplish this by having the application maintain lock information in individual records or in a lock table. This solution is not foolproof. If the data is also available via ODBC or another data source, these locks do not prevent other users from accessing and updating data.

8.5 Service-oriented Architecture (SOA)

What we need to achieve our mobile computing solution is a technology that allows us to integrate our COBOL back-end application with a non-COBOL front end in a wireless environment. A business application designed with service-oriented architecture (SOA) is well suited to this task. What is SOA?

SOA is a methodology for developing applications with standards-based interfaces that accept specific inputs and deliver expected outputs. It uses the simple design principles from various structured programming methods. SOA has emerged in recent years as a logical way for organizations to keep and reuse existing legacy applications while integrating them with new technologies.

With SOA, the front-end program can be written in Java/JSP, VB/ASP, Delphi, XML, HTML, or a variety of other languages. Your front-end operating system can be Windows Mobile, Palm, or Symbian. This platform-independent solution provides insurance for your back-end architecture against any major changes in language or platform on the front end. Best of all, you can leverage your proven COBOL back-end application in a modern, wireless environment.

8.6 Methods for Mobile Computing

You can use the following ACUCOBOL-GT technologies as part of a mobile computing strategy:

- ACUCOBOL-GT COM Server

- ACUCOBOL-GT Common Gateway Interface (CGI) language extensions
- ACUCOBOL-GT runtime and Short Message Service (SMS) processing

The following sections provide brief descriptions of these technologies.

8.6.1 ACUCOBOL-GT COM Server

The ACUCOBOL-GT COM Server is a COM object that contains the ACUCOBOL-GT Windows dynamic link library (DLL). It allows your ACUCOBOL-GT application to be called by a program written in any COM-compliant language or by a third-party, off-the-shelf package that includes a COM interface for a mobile device. The front-end application connects to the ACUCOBOL-GT COM Server via the TCP/IP network. Your back-end application is in ACUCOBOL-GT. This deployment can be a handy solution when you want to transfer data to a server, which processes the raw data into useful information like summary reports and inventory lists.

8.6.2 ACUCOBOL-GT CGI Language Extensions

With the ACUCOBOL-GT runtime, users who need to access query or look-up services have a way to quickly and easily obtain important information. The front end consists of WML- or HTML-compliant software like a mobile device browser that can communicate via CGI with a Web server. The communication link is TCP/IP. On the back end, you have your COBOL application, a COBOL-based CGI program for interfacing between the mobile device and the COBOL application, and the ACUCOBOL-GT runtime. You can use this technology for applications in which you initiate a request for information, for example, up-to-the-minute currency exchange rates or the status of product shipments.

8.6.3 ACUCOBOL-GT Runtime and Short Message Service (SMS) Processing

The SMS system of text messaging is gaining in popularity as the use of mobile phones increases worldwide. An ACUCOBOL-GT application can send and receive SMS text messages by using the C$SOCKET library routine to communicate over TCP/IP with a telnet service available through a mobile service network provider. With this technology, your ACUCOBOL-GT application can be written to send an alert message to your mobile device, for example, when inventory for a particular item falls below a specified level or when report generation is completed.

Combining this solution with the ACUCOBOL-GT CGI option opens the door to an even more interesting possibility. For example, when SMS alerts you to that change in inventory, and your mobile device supports WAP, our CGI extensions allow access to a supplier's Web page so you can replenish your stock.

9 Working with Transaction Processing Systems

Key Topics

9.1 Introduction

Because much of the world's business data is processed by COBOL, and billions of COBOL transactions occur daily, it's important for enterprise-level COBOL applications to interoperate smoothly with online transaction processing (OLTP) software. Transaction processing software furnishes applications with secure access to one or more data sources, both within local networks and across wide-area networks or the Internet. Regardless of the individual transaction processing package, the key features of this type of software are its ability to maintain data integrity, its scalability, and the security mechanisms it provides.

ACUCOBOL-GT® can interoperate with a variety of online transaction processing packages. This chapter provides some background information about transaction processing in general, and provides some information about configuring OLTP software to work with ACUCOBOL-GT, specifically with IBM® CICS® and BEA Tuxedo®. It also contains a section describing some debugging options for use in a transaction processing environment. For more information on the use of ACUCOBOL-GT with IBM TXSeries® CICS or Sun™ Mainframe Rehosting environments, please contact your Acucorp Sales Professional.

9.2 What Is Transaction Processing?

A *transaction* is unit of work consisting of one or more requests or updates to a system. A transaction is successfully completed with a *commit* command that finalizes all changes. If the transaction is not successful, a *rollback* command is used to undo changes and return the system to its previous state. A transaction, therefore, processes either all or none of its changes.

Consider, for example, the common transaction of withdrawing money from a checking account. The customer enters the request at an automated teller, causing a program to check the availability of funds, dispense the money, and update the balance of the user account. Each step must be completed before a successful transaction can occur.

Transaction processing software keeps track of each part of the transaction, ensuring data integrity by verifying that the transaction has been successfully completed before performing any updates.

The four basic properties of a transaction are known as the **ACID** properties:

- Atomicity – A transaction as a whole is either done or undone. If a transaction involves more than one discrete piece of information, all pieces are committed or none are.
- Consistency – A transaction leaves the data in a valid state. Either a change has been committed and the data has a new valid state, or no change occurs and the data returns to its previous valid state.
- Isolation – Each transaction occurs independently of other transactions taking place in the same environment.
- Durability – The effects of a transaction are permanent, so that even in the event of a system failure, upon restart, the data is available in its correct state.

9.3 IBM CICS

The IBM Customer Information Control System (CICS) is a widely installed transaction processing software system. A large number of transaction processing requirements running on mainframes today are handled by CICS systems. IBM designed CICS to support large numbers of terminals and a large transaction volume with fast and consistent response time. CICS can be described as an interface between CICS applications and the operating system.

CICS or CICS-compatible environments are now available for many platforms, including server support for OS/2, AIX, Windows NT, MVS/ESA, AS/400, HP-UX, Digital OSF/1, Siemens SINIX, Sun Solaris, and other UNIX platforms.

In supported environments, ACUCOBOL-GT offers the following features:

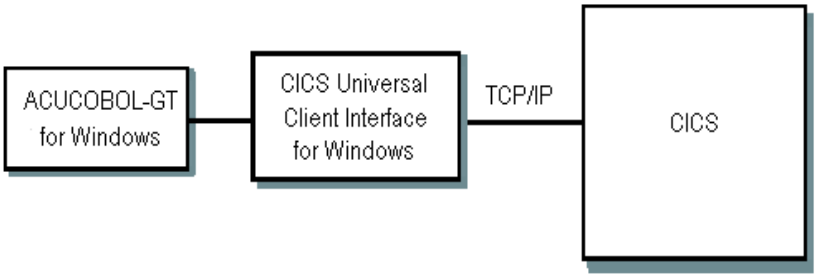
- Dynamic loading of shared libraries, including EXTFH libraries used for batch programs, so you can customize your environment by simply specifying an environment variable
- Debugging of CICS COBOL programs with the ACUCOBOL-GT runtime debugger
- The ability to interoperate with a wide variety of utilities that help you replicate mainframe functionality on open systems, such as external sort modules and utilities to handle complex calculations

9.4 Working with the IBM CICS Transaction Gateway

The Windows versions of the ACUCOBOL-GT runtime support calls to the IBM CICS Universal Client Interface for Windows. The runtime attempts to automatically load the CICS dynamic link library (DLL) if any subroutine beginning with "CICS" is called. If the DLL is available and loadable, the CICS function is called using the standard DLL call interface.

In some cases, the IBM CICS Transaction Gateway product may include a ".lib" file instead of a DLL. When this occurs, the Windows runtime DLL must be relinked to include the ".lib" file in order to access CICS functions. The relinking process is described below.

Whether you are working with a DLL or ".lib" file, a properly configured ACUCOBOL-GT runtime allows you to access IBM applications on a mainframe or other host, and to use ACUCOBOL-GT graphical interface capabilities as a front end to your IBM COBOL applications.



To take advantage of these capabilities, you must install the appropriate combination of the CICS Transaction Gateway and CICS Universal Client Interface on your machine and configure it according to the IBM documentation. IBM documentation includes specific information on CICS and sample COBOL programs that you can use to test your client/server connection.

9.4.1 Including the Transaction Gateway Routines in the Runtime

For Windows systems, IBM may supply the Transaction Gateway library as a ".lib" file rather than a DLL. As mentioned earlier, the Transaction Gateway routines must be linked into the ACUCOBOL-GT runtime. The process uses the direct call method described in Chapter 6 of this guide.

To link the Transaction Gateway library into "wrun32.dll", you must:

1. Include the following lines in the file "direct.c", located in the acugt\lib folder of your ACUCOBOL-GT installation.

```
extern short CICS_ExternalCall();
extern short CICS_EciListSystems();
struct DIRECTTABLE LIBDIRECT[] = {
   { "CICSEXTERNALCALL", FUNC CICS_ExternalCall, C_short },
   { "CICSECILISTSYSTEMS", FUNC CICS_EciListSystems, C_short },
   { NULL, NULL, 0 }
   };
```

2. Open Visual Studio .NET and load the solution file “wrun32.sln”, located in the acugt\lib folder.

3. Open the **Project/Properties** interface, expand the **Linker** folder, and do the following:

   a. Select **Input**, then add the name of the Transaction Gateway library, “cclwin32.lib”, next to “Additional Dependencies”.

   b. Select **General** and add the path to “cclwin32.lib” on the “Additional Library Directories” line. For example:

      ```
      C:\Program Files\IBM\IBM CICS Transaction Gateway\lib
      ```

4. On the main menu bar, select **Build/Rebuild Solution**. This command recompiles all of the required files, including “direct.c”, and links “cclwin32.lib” into “wrun32.dll”.

9.4.2 Connecting to CICS Applications

If you are using the CICS client, you must compile your ACUCOBOL-GT application with the “-Dw32” and “-Da4” options. These are data format switches. The runtime does not require any special switches to use the IBM library routines or COPY files once you have compiled with the correct options. You must also set the USE_CICS runtime configuration variable to a value of “1”.

Note: Compiling with any other “-D” flags could cause problems in compilation, so use them with caution.

ACUCOBOL-GT supports the External Call Interface (ECI) portion of CICS. The ECI allows a non-CICS application to call a CICS program that resides on a CICS server. The non-CICS application does not issue any CICS commands itself; these commands are issued by the called program running on the server. The call is no different from any other call built into a library or COBOL program. The CICSEXTERNALCALL ECI call can be used to:

- Call programs (synchronously or asynchronously).
- Request STATE information (synchronously or asynchronously).

- Check on previous asynchronous calls using a number of GET REPLY options.

ACUCOBOL-GT also supports the CICSECILISTSYSTEMS call, which is used to determine the servers available to receive CICSEXTERNALCALL requests.

9.5 Working with IBM TXSeries CICS

Based on the solid foundation of mainframe CICS, the IBM TXSeries products allow you to build on technologies that offer interoperability with mainframe systems and the flexibility to integrate with modern programming models. TXSeries CICS provides an application environment for running COBOL programs in high-volume, OLTP systems. Because TXSeries supports the CICS API and SPI (Systems Programming Interface), many CICS application programs can run on TXSeries without any changes.

The TXSeries CICS platform includes:

- A wide range of supported relational database managers
- Full interoperability with other CICS-compatible environments
- Familiar EXEC CICS APIs
- Integration with modern desktop environments
- Easy access to enterprise-wide e-business applications
- Options to extend existing investments while taking advantage of new technologies like the IBM WebSphere® platform
- Communication with back-end CICS and the Information Management System (IMS)

As with mainframe CICS, TXSeries launches a transaction and then initializes the entire operating environment, performing virtually all of the data access and communication services, launching the application programs, and handling calls from one program to another.

For more information on using ACUCOBOL-GT with TXSeries CICS, please contact your Acucorp Sales Professional.

For additional information on TXSeries, refer to the IBM Web site at http://www.ibm.com/software/htp/txseries/support.

9.5.1 How TXSeries CICS Works with ACUCOBOL-GT

With the integrated TXSeries and ACUCOBOL-GT environment, you can take advantage of features such as:

- A prelinked version of the ACUCOBOL-GT runtime system shipped with TXSeries that eliminates the manual build process as you configure and deploy on TXSeries CICS
- A range of security options that address processing requirements in the distributed environment, including CICS-based security for resource and transaction access, or open systems models such as the Distributed Computing Environment (DCE)
- Communication between TXSeries and the mainframe CICS TS via the standard CICS Intersystem Communication (ISC) facilities. With this feature, you can stage your migration process over time to help reduce risks and extend strategic ACUCOBOL-GT applications as you migrate them.

The existing mainframe process to compile and deploy COBOL programs remains largely unchanged: CICS programs are precompiled in TXSeries to translate EXEC CICS syntax to COBOL verbs. Programs are then compiled with the ACUCOBOL-GT compiler. TXSeries initializes the runtime, waits for requests from clients, and processes those requests, allowing ACUCOBOL-GT programs to communicate with databases and other programs or clients, and to display screens using CICS services.

9.5.2 Modernizing Applications

After you move your COBOL CICS applications to the TXSeries environment and ACUCOBOL-GT, you have extensive options for modernization, such as:

- Invoking CICS programs through the ECI to perform back-end processing for Web services.
- Increasing developer productivity with the AcuBench® integrated development environment from Acucorp, which extends and enhances the ACUCOBOL-GT compiler and runtime system with a powerful suite of GUI-based development tools.
- Updating the user interface for migrated applications by adding .NET clients to invoke CICS programs through the ECI.

9.6 Working with UniKix Mainframe Rehosting Software

UniKix™ Mainframe Rehosting Software, from Clerity Solutions, enables customers to rehost legacy mainframe applications on open systems platforms such as IBM AIX, HP-UX, Sun Solaris, and Linux, making it feasible to replicate mainframe functionality on open systems without needing to rewrite applications in a new language. Two products, Unikix Transaction Processing Environment (TPE) and UniKix Batch Processing Environment (BPE), together provide an environment for running CICS and batch applications on open systems.

Similar to mainframe CICS, Unikix TPE software launches a transaction and then initializes the entire operating environment, handling all the data access, communication, and calls between programs. UniKix also provides several facilities to share resources and data between UniKix regions and IBM CICS systems. An IBM CICS mainframe system appears as a remote region to UniKix through Intersystem Communication (ISC).

UniKix BPE software provides a complete batch job execution facility. The software is composed of independent processes that manage and schedule batch programs according to configuration parameters, such as start time and job priority. UniKix provides functionality to allow administrators to assign job attributes, change job attributes, and determine the current status of a job.

ACUCOBOL-GT supports CICS mainframe logic running in the UniKix TPE and UniKix BPE environment in a number of ways:

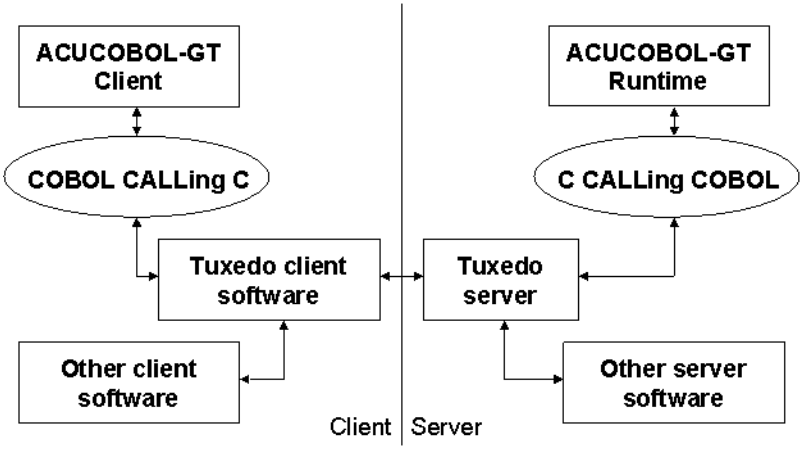
- Dynamic loading of shared libraries, including EXTFH libraries used to run batch programs
- Interoperability with a variety of utilities that replicate mainframe functionality, such as external sort modules and utilities that handle complex calculations
- Debugging of CICS COBOL programs using the ACUCOBOL-GT runtime debugger
- Familiar EXEC CICS APIs
- Integration with modern desktop environments
- Easy access to enterprise-wide e-business applications

For more information on using ACUCOBOL-GT with UniKix, please contact your Acucorp Sales Professional.

9.7 Working With BEA Tuxedo

ACUCOBOL-GT developers using the BEA Tuxedo platform for distributed transaction processing and message-based application development can create Tuxedo clients and Tuxedo services from ACUCOBOL-GT applications. Acucorp is certified for Tuxedo 9.1 on all supported platforms.

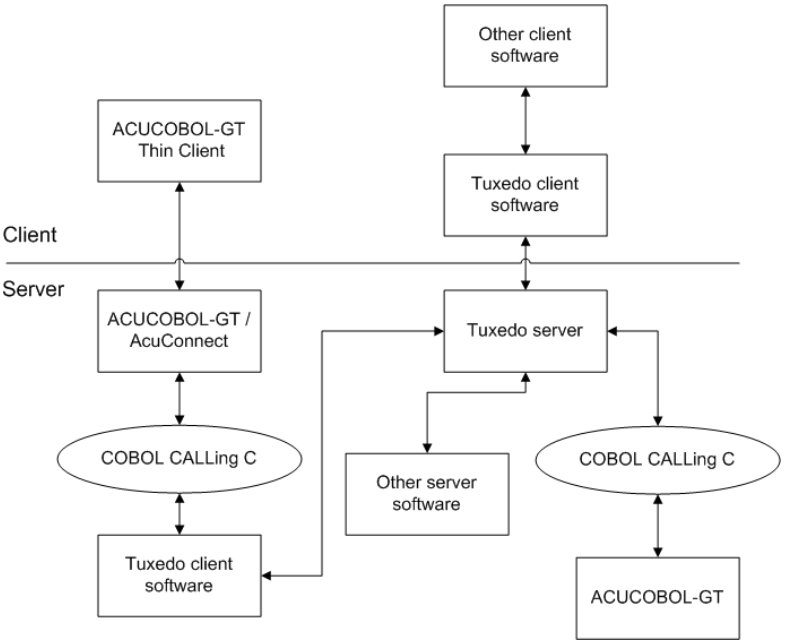
ACUCOBOL-GT and BEA Tuxedo can work together in either a distributed processing (client/server) environment or a thin client environment. In a distributed processing environment, the interaction occurs as shown in the following diagram:



In this environment, ACUCOBOL-GT interoperates with BEA Tuxedo using the same routines used to call C programs (COBOL-calling-C routines on the client and C-calling-COBOL routines on the server).

In BEA Tuxedo client/server distributed applications, the client requests the services of a server, which may have multiple services to provide, and the outcome of the request is returned to the client by the service. This architecture lets you divide application processing among multiple machines to optimize performance.

ACUCOBOL-GT and BEA Tuxedo can also work together using Acucorp's Thin Client technology. The thin client architecture uses AcuConnect®, Acucorp's remote application server, as shown in the following diagram:



Because BEA Tuxedo is based on C, the procedure for interfacing with Tuxedo is much like the procedure for interfacing with C routines in general. This means that external C variables need to be defined in "direct.c".

To prepare BEA Tuxedo to interface with ACUCOBOL-GT, C code must be added to the initialization and termination routines of BEA Tuxedo servers, then linked into the server using the BEA Tuxedo **buildserver** command. This process is further explained in the sections that follow.

9.7.1 Creating a Tuxedo Client Program

A BEA Tuxedo client is a software module that collects a user request and forwards it to a server that offers the requested service. To create a Tuxedo client with ACUCOBOL-GT, you must relink the ACUCOBOL-GT runtime with the Tuxedo libraries. This section provides an overview of the steps necessary to create the Tuxedo client.

1. Add the necessary Tuxedo calls to your ACUCOBOL-GT program. These calls—used to open and close resources, begin and end transactions, and support communication between clients and servers—are collected in a Tuxedo API known as the Application to Transaction Monitor Interface (ATMI). ATMI functions are described in the Tuxedo documentation.

2. Recompile your program to establish it as a BEA Tuxedo client.

3. Relink the ACUCOBOL-GT runtime on the client machine to include the BEA Tuxedo client libraries. The client libraries vary, depending on the BEA Tuxedo version and the platform you are using. To determine the Tuxedo client libraries used on a particular client, use the BEA Tuxedo command:

   ```
   buildclient -C -v -w
   ```

   where:

-C	Specifies COBOL compilation
-v	Specifies that **buildclient** should work in verbose mode, displaying the compilation command on standard output
-w	Specifies that the client is to be built using the workstation libraries; used only with the WSL

   Because the Tuxedo libraries are external C routines, link the libraries into the ACUCOBOL-GT runtime using the direct method of calling C routines described in Chapter 6.

4. Set client environment variables.

9.7.2 Creating a Tuxedo Server

A BEA Tuxedo server is a process that provides one or more services to a client. To build server processes, applications combine their service subroutines with a controlling program provided by the BEA Tuxedo system. Using ACUCOBOL-GT, this involves the following steps:

1. Create and compile an ACUCOBOL-GT program that performs a specific task or service.

2. Create a configuration file for the service to establish the identifiers

3. Edit the "appinit.c" program (provided by BEA) to include runtime initialization and shutdown functions specific to ACUCOBOL-GT.

   The program calls the standard BEA Tuxedo ATMI initialization and shutdown subroutines **tpsvrinit()** and **tpsvrdone()**. The **tpsvrinit** routine calls two functions: **tpopen** opens the resource manager, and **userlog** posts a message that the server has started. The **tpsvrdone** routine also calls a **tpclose** function, which closes the resource manager

4. Link the ACUCOBOL-GT runtime libraries into the controlling program provided by the BEA Tuxedo system.

   On UNIX/Linux servers, use the **buildserver** command, adding the ACUCOBOL-GT runtime libraries. Acucorp supplies a makefile that can be used to simplify this process.

   On Windows servers, relink the required Tuxedo libraries into the ACUCOBOL-GT runtime DLL, then issue the **buildserver** command with the ACUCOBOL-GT runtime libraries.

9.7.3 Running Your Tuxedo Application

1. Use the **tmloadcf** command to load the server configuration file.

2. Use the **tmboot** command to boot the application.

3. Run the client application using the **runcbl** command.

9.8 Background Debugging Options

Various methods are available for debugging programs running in background mode in a transaction processing environment. These are described in sections 9.8.1 and 9.8.2.

9.8.1 Background Debugging With an xterm

With a properly configured X Window System, you can debug programs executing in background mode (with the "-b" runtime flag) by specifying that debugging I/O be sent to an xterm window on a specified X server. Use the following procedures:

1. Set the DISPLAY environment variable on the system executing the program to point to a valid X server. This setting must be made before the runtime is invoked.
2. Give the user executing the program rights to start an xterm on the specified X server.
3. Modify the PATH environment variable on the system executing the program to point to the xterm directory.
4. Add both the "-b" and the "-d" flags to the runtime command line.

Use the standard debugging commands described in the rest of this section to manage your debug session.

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.

9.8.2 Defining debugging methods with “ADM_t”

All of the methods described in this section require support from the vendor of the transaction processing software. Please refer to your vendor’s documentation for implementation details.

The **a_cobol_info** structure described in lib/sub.h includes the variable ADM_t debug_method, which describes which debugging method is used. The ADM_t type is described in lib/sub.h as an enumeration with the following values:

ADM_NONE	no debugging
ADM_XTERM	debug using a new xterm
ADM_TERMINAL	debug using an existing terminal through **runcbl**
ADM_THINCLIENT	debug using a waiting thin client

A separate char *debug_method_string setting depends on debug_method, as described in the following sections.

9.8.2.1 Using an xterm

To debug using an xterm, first set debug_method to ADM_XTERM. If the DISPLAY environment variable is not set (or set incorrectly), set debug_method_string to point to your X server.

At execution time, the ACUCOBOL-GT runtime creates an xterm with the COBOL program stopped at the first line of the program.

Note that when the debugger starts, debugger settings are loaded from a file, and changes to settings are saved to that file as the debugger runs. By default, that file is given the name of the user who started the background or server process. In some environments, this could mean that multiple users of the

debugger could share the same debugger settings. In such a case, breakpoints set or cleared by one user, for example, would affect what appeared in every other user's debugger window.

To avoid this potential problem, use the "acudebug.user" X resource to specify separate debugger settings files for different users. For example, in your "~/.Xdefaults" file, you could add the line:

```
acudebug.user:  dsmith
```

Debugger settings would then be loaded from and saved to a file called "dsmith.adb".

See your X server resource database documentation ("man xrdb") for additional instructions on how to set X server resource properties.

9.8.2.2 Using a terminal

To debug using an existing terminal, use the following procedures:

1. Set debug_method to ADM_TERMINAL, then set debug_method_string to the tty string on which you will execute **runcbl**.

2. Log on to the UNIX machine that hosts the OLTP software and note the tty device used.

3. On that machine, execute the runtime with the "--wait" flag. When this option is used, the runtime waits for the OLTP runtime to contact it.

   Note that if you use an additional command-line option, "--restart", the runtime restarts itself after each debugging session. This option is particularly helpful if you have transactions that require multiple COBOL programs and you want to debug all of them. Keep in mind, however, that if you use the "--restart" option, you must use the interrupt sequence (Ctrl+C) to terminate the runtime process when you are finished debugging.

4. Execute the transaction you intend to debug, ensuring that the tty variable in your transaction debugging screen is set correctly. When the transaction executes, the runtime debugger appears on the terminal, allowing you to set breakpoints, step, evaluate variables, and so on. Once the OLTP runtime terminates the transaction, the terminal returns to a shell prompt.

9.8.2.3 Using the thin client

Set debug_method to ADM_THINCLIENT. Set debug_method_string to **client:port**, where *client* is the host on which **acuthin** is executing and *port* is the port on which it is listening.

On the Windows client

1. Execute the **acuthin** command with "--wait [--port *nnnn*]" where *nnnn* is the desired port number. The thin client waits for a runtime to connect to it and behaves as the thin client normally does.

   In this mode, **acuthin** has an additional command-line option, "--restart". When you specify this option, the thin client restarts itself after each debugging session. If you have transactions that require multiple COBOL programs, you can easily debug all of them. Keep in mind, however, that when you use the "--restart" option, you must use the Windows Task Manager to terminate the thin client process when you have finished debugging.

2. When the runtime debugger screen appears, you can debug your application with all the usual runtime debugger options.

Please note that in a transaction processing environment, if the server program attempts to connect to **acuthin** and **acuthin** is not running, the return value will be "7", which is COBOL_NONFATAL_ERROR.

On the server

If you are using a UNIX server, set the A_DEBUG_USING_THIN environment variable to a non-zero numeric value before the transaction server initializes the runtime. This tells the runtime to initialize in a mode which allows it to communicate with the thin client. After you enable this mode, you may not use terminal or xterm debugging.

In Windows server environments, the transaction runtime is automatically initialized in the mode that allows it to communicate with the thin client whenever it is run from within a transaction server (such as CICS).

10 Working with Messaging Middleware

Key Topics

10.1 Support for IBM WebSphere MQ

IBM WebSphere MQ, formerly MQSeries®, is messaging middleware designed to enable application integration. The WebSphere MQ products help business applications to exchange information across different platforms by sending and receiving data as messages. WebSphere MQ provides base messaging functions for servers and clients. It handles network interfaces, communications protocols, and workload distribution so that messages can be delivered promptly.

WebSphere MQ provides a consistent multiplatform, application-programming interface for coding messaging tasks. It supports many different platforms, including AIX, Compaq NSK, DOS, DYNIX/ptx, HP UX, Linux, Mac OS, MVS/ESA, NUMA-Q, OpenVMS Alpha, OpenVMS VAX, OS/2, OS/390, OS/400, Solaris, UNIX, Unisys 2200 Series, Unisys A Series, UnixWare, VM/ESA, VSE/ESA, Windows 2000, Windows 3.x, Windows 95, Windows 98, Windows NT, and Java.

ACUCOBOL-GT supports IBM WebSphere MQ in the following ways:

- The Windows versions of the ACUCOBOL-GT runtime support calls to WebSphere MQ. When properly configured, the Windows runtime attempts to automatically load the WebSphere MQ DLL if any subroutine beginning with “MQ” is called. If the DLL is available and loadable, the WebSphere MQ function is called via the standard DLL call interface.

- The ACUCOBOL-GT runtime for UNIX supports the shared libraries provided with IBM WebSphere MQ without relinking.

- Both the Windows and UNIX runtimes use the COBOL COPY files supplied by IBM WebSphere MQ

Properly configured, the ACUCOBOL-GT runtime allows you to access IBM applications on a mainframe or other host, and to use ACUCOBOL-GT GUI interfaces as a front end to your IBM COBOL applications.

10.2 Support for IBM Shared Libraries

For UNIX and similar systems, the ACUCOBOL-GT runtime supports the shared libraries provided by IBM WebSphere MQ. Your programs running on UNIX can call platform-specific shared libraries to load them, and then call the necessary WebSphere MQ routines. You should not need to relink the runtime with the libraries in order to call the routines.

If you are in a UNIX environment, after installation of the client, identify the location of the shared libraries that contain the call routines. Confirm that the path matches the path you provided in the shared library environment variable. If the paths don't match, modify the variable as required.

On Linux, “libmqic_r.so” and “libmqmcs_r.so” are the shared libraries that IBM provides containing WebSphere MQ routines.

Note: Windows-based ACUCOBOL-GT programs do not need to load the DLL file, “mqic32.dll”, provided by IBM for accessing the routines.

10.3 Support for WebSphere MQ COPY Files

Both the Windows and UNIX runtimes use the set of COBOL COPY files that IBM provides with WebSphere MQ. These COPY files contain constant definitions and data structures used in the WebSphere MQ calls. Listed below are the IBM COPY files that our compiler can use:

cmqbol.cpy	cmqbov.cpy	cmqcnol.cpy	cmqcnov.cpy
cmqdhl.cpy	cmqdhv.cpy	cmqdlhl.cpy	cmqdlhv.cpy
cmqgmol.cpy	cmqgmov.cpy	cmqiihl.cpy	cmqiihv.cpy
cmqmd1l.cpy	cmqmd1v.cpy	cmqmdel.cpy	cmqmdev.cpy
cmqmdl.cpy	cmqmdv.cpy	cmqodl.cpy	cmqodv.cpy
cmqorl.cpy	cmqorv.cpy	cmqpmol.cpy	cmqpmov.cpy
cmqrmhl.cpy	cmqrmhv.cpy	cmqrrl.cpy	cmqrrv.cpy
cmqtmc2l.cpy	cmqtmc2v.cpy	cmqtml.cpy	cmqtmv.cpy
cmqv.cpy	cmqxqhl.cpy	cmqxqhv.cpy	

Depending on the functionality of your application, you may not require all of the COPY files that IBM provides, but if you are not sure which COPY files to use, it is safest to include them all in your program.

10.4 Connecting to WebSphere MQ Applications

To use your ACUCOBOL-GT program with IBM's WebSphere MQ, you must perform the following steps:

1. Install and configure the WebSphere MQ client software as described in the IBM documentation.
2. Add the necessary WebSphere MQ calls to your ACUCOBOL-GT program as described in section section 10.4.1.
3. Define a message buffer and any other variables in Working-Storage as shown in section 10.4.2.
4. Compile your program using the "-D5" data format switch as described in section 10.4.3.
5. Configure the runtime and environment by setting the USE_MQSERIES configuration variable and MQSERVER environment variable as explained in section 10.4.4.

These steps are described in more detail in the following sections.

10.4.1 Adding WebSphere MQ Calls to Your ACUCOBOL-GT Program

In order to interface with IBM WebSphere MQ, your ACUCOBOL-GT program must be able to connect to the queue manager, open specific queues, read messages from queues, write messages to queues, close queues, and disconnect from the queue manager. Once the WebSphere MQ Client software is loaded and configured, you can set up a queue on the message queue manager or server to receive messages. We support calls to the message queues and the message queue manager.

Note: Once they are defined, the queue manager and queue names are case sensitive.

The runtime uses the following WebSphere MQ calls to communicate with other WebSphere MQ applications by sending and receiving messages:

Connection Calls

MQCONN - Connect to queue manager
MQCONNX - Connect to queue manager (extended)
MQDISC - Disconnect queue manager

Queue Manipulation Calls:

MQOPEN - Open object (usually a queue)
MQCLOSE - Close Object
MQINQ - Inquire about an object
MQGET - Get a message off of the queue
MQPUT - Put a message on the queue
MQPUT1 - Open, put single message, close in one call
MQSET - Set object attributes

Transaction Support

MQBEGIN - Begin Unit of work
MQBACK - Back out changes
MQCMIT - Commit changes

These routines can be called directly from COBOL, as shown in the subsequent sections. Please note that for Windows-based applications, it is not necessary to use the parameter qualifiers “BY VALUE” or “BY REFERENCE” if the runtime is configured to load the WebSphere MQ DLL automatically. (See section 10.4.4, “Configuring the Runtime and Environment,” for details.) However, if you load the DLL manually, or if you are operating under UNIX or Linux, you must use these qualifiers.

For all of these calls, we recommend that you check the COMP-CODE and REASON variables to verify that the calls executed successfully.

For a detailed description of all available commands and their usage, refer to the IBM WebSphere MQ manuals.

10.4.1.1 Connecting to the queue manager

To connect to the WebSphere MQ queue manager from your ACUCOBOL-GT program, use the following code:

```
CALL 'MQCONN'
    USING QM-NAME, by reference HCONN,
    COMPCODE, REASON
```

in the following modes:

Input Mode:

Variable	Description	Definition
QM-NAME	Variable string that contains the name of the queue manager	User-defined

Output Mode:

Variable	Description	Definition
HCONN	S9(9) BINARY handle to queue manager	User-defined
COMPCODE	S9(9) BINARY returns the completion code	User-defined
REASON	S9(9) BINARY returns the reason	User-defined

10.4.1.2 Opening specific queues

To open specific message queues in WebSphere MQ, use the following code:

```
CALL 'MQOPEN'
    USING by value HCONN, by reference OBJDESC,
    by value OPTS, by reference HQUEUE,
    COMPCODE, REASON
```

in the following modes:

INPUT Mode:

Variable	Description	Definition
HCONN	S9(9) BINARY handle to queue manager	User-defined
OBJDESC	Structure that identifies the object to be opened	cmqodv.cpy
OPTS	S9(9) BINARY options that control the action of MQOPEN	User-defined

The available OPTS S9(9) BINARY options are specified in “cmqv.cpy”. From the options that are available, at least one of the following options must be specified:

- MQOO-BROWSE - Open in browse messages
- MQOO-INPUT - Open in read messages
- MQOO-OUTPUT - Open to write messages
- MQOO-SET - Open to set properties of objects
- MQOO-INQUIRE - Open to inquire about objects

To specify multiple options, add them together.

OUTPUT Mode:

Variable	Description	Definition
HQUEUE	S9(9) BINARY handle to queue	User-defined
COMPCODE	S9(9) BINARY returns the completion code	User-defined
REASON	S9(9) BINARY returns the reason	User-defined

10.4.1.3 Reading messages from queues

To read messages from a WebSphere MQ queue, use the following code:

```
CALL 'MQGET'
    USING by value HCONN, by value HQUEUE,
    by reference MSGDESC, by reference GETMSGOPTS,
    by value BUFFER-LEN, by reference BUFFER-REP,
    by reference DATA-LEN,
    COMPCODE, REASON
```

in the following modes:

INPUT Mode:

Variable	Description	Definition
HCONN	S9(9) BINARY handle to queue manager	User-defined
HQUEUE	S9(9) BINARY handle to queue	User-defined
BUFFER-LEN	S9(9) BINARY length in bytes of buffer area	User-defined
DATA-LEN	S9(9) BINARY length in bytes of data in the message	User-defined

OUTPUT Mode:

Variable	Description	Definition
BUFFER-REP	Area to contain the message data	User-defined
COMPCODE	S9(9) BINARY returns the completion code	User-defined
REASON	S9(9) BINARY returns the reason	User-defined

I-O Mode:

Variable	Description	Definition
MSGDESC	Structure describes the attributes of the message required and the attributes of the message retrieved	cmqmdv.cpy

Variable	Description	Definition
GETMSGOPTS	Options that control the action of MQGET For example, if you want to read a specific message, then you can specify the option MQMO_MATCH_CORREL_ID which causes only messages with the specified correl-id to be retrieved.	cmqgmov.cpy

10.4.1.4 Writing messages to queues

To write messages to queues in WebSphere MQ, use the following code:

```
CALL "MQPUT"
    USING by value HCONN, by value HQUEUE,
    by reference MSGDESC,PUTMSGOPTS,
    by value BUFFER-LEN, by reference BUFFER-REQ,
    by reference COMPCODE, REASON.
```

in the following modes:

INPUT Mode:

Variable	Description	Definition
HCONN	S9(9) BINARY handle to queue manager	User-defined
HQUEUE	S9(9) BINARY handle to queue	User-defined
BUFFER-LEN	S9(9) BINARY length in bytes of buffer area	User-defined

OUTPUT Mode:

Variable	Description	Definition
BUFFER-REQ	Message data area	User-defined
COMPCODE	S9(9) BINARY returns the completion code	User-defined
REASON	S9(9) BINARY returns the reason	User-defined

I-O Mode:

Variable	Description	Definition
MSGDESC	Structure describes the attributes of the message sent and receives information about the message after sending	cmqmdv.cpy
PUTMSGOPTS	Options that control the action of MQPUT	cmqpmov.cpy

10.4.1.5 Closing queues

To close message queues in WebSphere MQ, use the following code:

```
CALL 'MQCLOSE'
    USING by value HCONN, by reference HQUEUE,
    by value OPTS,
    by reference COMPCODE, REASON.
```

in the following modes:

INPUT Mode:

Variable	Description	Definition
HCONN	S9(9) BINARY handle to queue manager	User-defined
OPTS	S9(9) BINARY options that control the action of MQCLOSE	User-defined

The available OPTS S9(9) BINARY options are specified in “cmqv.cpy”. Only one of the following options should be specified:

- MQCO_NONE
- MQCO_DELETE
- MQCO_DELETE_PURGE

OUTPUT Mode:

Variable	Description	Definition
COMPCODE	S9(9) BINARY returns the completion code	User-defined
REASON	S9(9) BINARY returns the reason	User-defined

10.4.1.6 Disconnecting from the queue manager

To disconnect from the WebSphere MQ queue manager, use the following code:

```
CALL 'MQDISC'
    USING by reference HCONN, COMPCODE, REASON.
```

in the following modes:

INPUT Mode:

Variable	Description	Definition
HCONN	S9(9) BINARY handle to queue manager	User-defined

OUTPUT Mode:

Variable	Description	Definition
COMPCODE	S9(9) BINARY returns the completion code	User-defined
REASON	S9(9) BINARY returns the reason	User-defined

10.4.2 Setting Up Working-Storage

In addition to adding WebSphere MQ calls to your program, you will need to define a message buffer and any other variables in Working-Storage. For example:

```
01 QM-NAME          PIC X(48) VALUE SPACES.
01 HCONN            PIC S9(9) BINARY.
01 COMPCODE         PIC S9(9) BINARY.
```

```
01 REASON          PIC S9(9) BINARY.
01 OPTIONS         PIC S9(9) BINARY.
01 HOBJ            PIC S9(9) BINARY.
01 BUFFER-LENGTH   PIC S9(9) BINARY.
01 max-buff-length PIC S9(9) BINARY.
01 BUFFER          PIC X(1024).
```

10.4.3 Compiling Your Application

When your ACUCOBOL-GT program is ready, compile it using the "-D5" data format switch. This causes data items declared as BINARY to be treated as COMP-5. This means that the data values are stored in the host machine's native byte-order instead of the machine-independent byte-order normally used. Such data items can contain values of magnitude up to the capacity of the native binary representation (2, 4, or 8 bytes) instead of being limited to the value implied by the number of 9s in the picture. The runtime will not require any special switches to use the IBM library routines or COPY files once you have compiled with this option.

Note: Use caution when compiling with any other "-D" flags because this could cause problems in compilation

---

10.4.4 Configuring the Runtime and Environment

The configuration variable USE_MQSERIES indicates that the program makes calls to WebSphere MQ. If you plan to load the WebSphere MQ DLL yourself, then there is no need for this variable (however you must use the "BY VALUE" and "BY REFERENCE" phrases when calling an MQ routine).

If you do not plan to load the DLL yourself, set this variable to "1" (on, true, yes). The runtime will automatically load the DLL and pass calls beginning with the string "MQ" to the WebSphere MQ client. If the named routine does not exist, the runtime uses the normal search sequence to find a matching function. The "BY VALUE" and "BY REFERENCE" phrases are not required if you are using the USE_MQSERIES variable.

If you are running on a Windows client machine, you must set the MQSERVER environment variable to the name of the physical server as defined in the Host file, or to a fixed IP address. On Windows NT, XP or 2000 systems, set the variable in the "Environment Variables" dialog, accessed from the Advanced tab of the Control Panel's "System" applet. On Windows 98/ME systems, set the variable in the "autoexec.bat" file. The variable takes effect after the system has been rebooted.

Note: Even though you are setting this definition on a Windows system, you must use only a forward slash. Back slashes are not recognized by WebSphere MQ.

---

11 Working with Non-Vision Data

Key Topics

11.1 Introduction

ACUCOBOL-GT® includes a native indexed sequential file system known as Vision. Although many of our customers convert their data to the Vision format, many others choose to keep their data in other COBOL and non-COBOL data formats. This preference is no problem for ACUCOBOL-GT. ACUCOBOL-GT programs can interoperate with many kinds of external data sources, including:

- XML documents
- SQL databases
- ODBC-compliant data sources
- C-ISAM and KSAM files
- File systems that use an EXTFH interface to access files, such as those in transaction processing environments

This chapter describes how to retrieve and update data from all of these data sources as well as how to configure and initialize external file systems.

In addition, ACUCOBOL-GT programs can work with a variety of external data types such as:

- Java data types and arrays
- C data types
- Variant data types and safearrays

These types are described elsewhere in this guide. See Chapter 2, section 2.4, for information on working with Java data. See Chapter 6 for information on working with C data. See Chapter 3, section 4.3.1 and section 4.3.2, for information on working with ActiveX data.

11.2 Working with XML Data

Applications written in ACUCOBOL-GT can interact with many different forms of data, including data marked with eXtensible Markup Language (XML).

ACUCOBOL-GT provides a variety of ways to interact with XML data. One of them is a runtime-resident file system interface known as AcuXML. AcuXML reads XML data and transparently converts it to sequential files for COBOL processing, and similarly converts COBOL output data into XML format when required. Using eXtended File Descriptors (XFDs) created at compile time, AcuXML maps the data transparently, making it easy to read and write XML records.

To facilitate the use of AcuXML, ACUCOBOL-GT includes a developer utility called **xml2fd** that creates File Descriptors (FDs) and SELECT statements from existing XML files. Although you may need to tune the results, you can include the FDs and SELECTS in your ACUCOBOL-GT program to prepare it for use with XML data.

For those who want more control over the parsing of data, ACUCOBOL-GT provides the C$XML library routine. This routine lets you define precisely which elements or attributes of the data to parse. For instance, you can request the handle of the entire XML tree, the first child element of the handle passed, the next sibling element, the number of attributes in an element, the name and value of the attributes, and so on. C$XML gives you much lower-level control over the parsing of XML data compared to AcuXML, but it requires more programming effort and XML knowledge.

Finally, ACUCOBOL-GT supports the IBM Enterprise COBOL XML GENERATE and XML PARSE statements. XML PARSE gives you a way to parse XML and process it in a COBOL program, associating processing procedures with the exception and non-exception cases that can result from the parse. XML GENERATE gives you a way to translate COBOL data into XML.

All three of these approaches can be used to parse record-based XML files, but please note that only C$XML and XML PARSE can be used to parse non-record-based XML files.

For Information On...	See...
AcuXML and **xml2fd**	Section 11.2.2 through 11.2.5 of this chapter
C$XML library routine	Section 11.2.6 of this chapter, and Appendix I in *ACUCOBOL-GT Appendices*
XML GENERATE and XML PARSE statements	*ACUCOBOL-GT Reference Manual*, Chapter 6, "Procedure Division."

11.2.1 XML Concepts

XML is a markup language for documents containing structured information. At first glance, XML is similar to HyperText Markup Language (HTML); with both, you mark up document content with descriptive tags surrounded by angle brackets. In XML, documents commonly include the following elements:

Comments	<!-- -->
Headers	<?xml version = "1.0"?>
Start tags	<record>
End tags	</record>

Elements can have children, which can have children, and so on. The main difference between HTML and XML, however, is that in HTML, the markup defines hypertext structure or display formatting. In XML, the markup defines the document content itself.

Unlike HTML, XML contains no predefined tag set or preconceived semantics for the markup. Rather, XML lets you define tags and the structural relationships between them based on the content of your document. For example, you may define tags called <lender>, <borrower>, and

<due-date>. Because XML documents contain information about themselves, they are an excellent vehicle for transporting data from one location to another.

11.2.1.1 XML documents

XML documents are strictly text files. In the context of data transport, the phrase "XML document" refers to a file or data stream containing any form of structured data. Examples include e-commerce transactions, server APIs, mathematical equations, customer information, and inventory status.

XML documents contain only markup and content. All of the rules and semantics of the document are defined by the applications that process them. Like HTML documents, XML documents can be displayed in a Web browser. In this case, the semantics are usually derived from style sheets or Resource Description Framework (RDF) files.

XML documents are said to be either "well-formed" or "valid." Well-formed documents follow all of the XML rules for a document—for example, for every start tag, there is a corresponding end tag. Valid XML documents are well-formed, but they also contain a document type definition (DTD), and they obey the constraints of that definition. For example, the following XML file contains a DTD that describes all the elements of the file:

```
<?xml version = "1.0"?>
<!DOCTYPE ORDERFILE [
<!ELEMENT ORDERFILE (CUSTOMER)*>
<!ELEMENT CUSTOMER (NAME, DATE, ORDERS)>
<!ELEMENT NAME (LAST-NAME, FIRST-NAME)>
<!ELEMENT LAST-NAME (#PCDATA)>
<!ELEMENT FIRST-NAME (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT ORDERS (ITEM)*>
<!ELEMENT ITEM (PRODUCT, NUMBER, (PRICE | CHARGEACCT | SAMPLE))>
<!ELEMENT PRODUCT (#PCDATA)>
<!ELEMENT NUMBER (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>
<!ELEMENT CHARGEACCT (#PCDATA)>
<!ELEMENT SAMPLE (#PCDATA)>
]>
<ORDERFILE>
    <CUSTOMER>
        <NAME>
            <LAST-NAME>Smith</LAST-NAME>
            <FIRST-NAME>Sam</FIRST-NAME>
```

```
            </NAME>
            <DATE>October 15, 2001</DATE>
            <ORDERS>
                <ITEM>
                    <PRODUCT>Tomatoes</PRODUCT>
                    <NUMBER>8</NUMBER>
                    <PRICE>1.25</PRICE>
                </ITEM>
                <ITEM>
            <PRODUCT>Apples</PRODUCT>
                    <NUMBER>12</NUMBER>
                    <PRICE>2.50</PRICE>
                </ITEM>
                <ITEM>
            <PRODUCT>Bananas</PRODUCT>
                    <NUMBER>6</NUMBER>
                    <PRICE>.50</PRICE>
                </ITEM>
            </ORDERS>
        </CUSTOMER>
        <CUSTOMER>
            <NAME>
                <LAST-NAME>Snead</LAST-NAME>
                <FIRST-NAME>Todd</FIRST-NAME>
            </NAME>
            <DATE>October 17, 2001</DATE>
            <ORDERS>
                <ITEM>
                    <PRODUCT>Slicer/Dicer</PRODUCT>
                    <NUMBER>1</NUMBER>
                    <CHARGEACCT>1234-5678-3456-7890</CHARGEACCT>
                </ITEM>
            </ORDERS>
        </CUSTOMER>
</ORDERFILE>
```

Documents that include and obey a schema rather than a DTD are considered to be "schema-valid." Schemas are files describing the XML document and its precise structure. XML documents that are accompanied by schemas produce the most reliable FDs and SELECT statements when run through the **xml2fd** utility, especially if the schema has information about data types and lengths.

Note: Please note that the **xml2fd** utility is used exclusively with AcuXML. If you are calling the C$XML library routine, you do not need to run XML documents through **xml2fd**.

Following is an example of a schema file for the same XML document. As you can see, it is much more descriptive than the DTD.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:annotation>
    <xs:documentation>
      Created by AcuXML(tm) version 6.0.0 (2002-11-12) on 2002/11/12
    </xs:documentation>
  </xs:annotation>

  <xs:element name="TEST-ORDERFILE">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="CUSTOMER"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="CUSTOMER">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="NAME"/>
        <xs:element ref="DATE"/>
        <xs:element ref="ORDERS"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="NAME">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="LAST-NAME"/>
        <xs:element ref="FIRST-NAME"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="ORDERS">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ITEM" maxOccurs="3"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="ITEM">
    <xs:complexType>
```

```
    <xs:sequence>
      <xs:element ref="PRODUCT"/>
      <xs:element ref="NUMBER"/>
      <xs:element ref="PRICE"/>
      <xs:element ref="CHARGEACCT"/>
      <xs:element ref="SAMPLE"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="LAST-NAME">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="5"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="FIRST-NAME">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="4"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="DATE">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="16"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="PRODUCT">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:maxLength value="12"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="NUMBER">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:totalDigits value="2"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

```
  <xs:element name="PRICE">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:maxLength value="4"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>

  <xs:element name="CHARGEACCT">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:maxLength value="19"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>

  <xs:element name="SAMPLE">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:totalDigits value="1"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>

</xs:schema>
```

ACUCOBOL-GT applications can read and write any type of XML data. Typically, the party with whom you are exchanging data will require that the data include a DTD. Occasionally, they may require a schema for their own development work.

With AcuXML, you can specify the type of XML output that you want to generate. You use the configuration variable AXML_CREATE_STYLE. If you want the output to include a schema, you set AXML_CREATE_STYLE to “schema”, and then use another configuration variable to define the schema name (AXML_SCHEMA_NAME). By default, a schema is then created each time ACUCOBOL-GT generates XML output. To prevent this for subsequent output, you can set a third configuration variable, AXML_CREATE_SCHEMA, to “false”, then just the name of the schema file is included. Configuration variables are described in Appendix H in *ACUCOBOL-GT Appendices*.

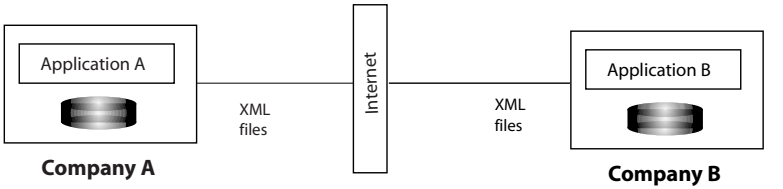
11.2.1.2 XML parsers

An XML parser is a processor that reads and interprets the contents of an XML document and determines the structure and properties of the data. Both AcuXML and C$XML are XML parsers. AcuXML automatically reads and interprets XML data based on XFDs created at compile time. C$XML parses XML data based on operation codes passed to the library call. The term "parse" is used throughout the following sections to mean "read" or "decode." ACUCOBOL-GT's XML parsers read and decode XML data so that it can be processed by your COBOL program.

11.2.1.3 Usage

AcuXML was designed for data exchange, but it can be beneficial for other uses as well. Following are some possibilities:

- Business-to-business data exchange – You can use AcuXML to enable efficient business-to-business data exchange in wide-area networks and over the Internet. Business partners such as manufacturers and suppliers, hospitals and insurance companies, buyers and sellers can exchange vital business data through the common adoption of XML.

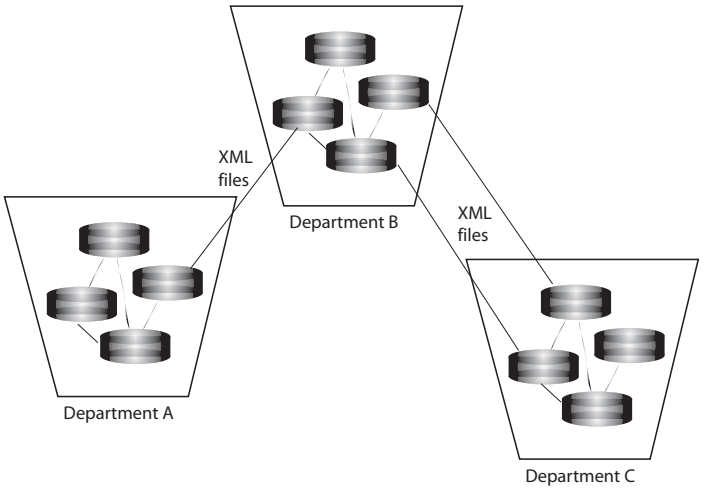


B2B Data Exchange Using XML

  In order for two parties to exchange documents, both parties need to explicitly agree on the meaning of tags and data within documents. For instance, in a procurement situation, both the buyer and seller must agree on what <STATUS> means in an invoice document.

- Application integration – If one of your goals is to merge data from multiple sources in order to integrate business processes, AcuXML can help. Converting your COBOL data to and from XML can enable it to be used throughout the enterprise, so long as the meaning of the data is

expressly understood. Similarly, data from other enterprise applications can be read and processed by your COBOL program, if your enterprise chooses to use XML as the common transport mechanism. Application integration can replace the isolated stovepipe systems of the past.



Application Integration with XML

- Enterprise Resource Planning (ERP) – ERP software promises to save companies money by centralizing processing functions that would otherwise be redundant in an organization. Enterprises that use ERP packages can choose XML as the data format used by the centralized system. AcuXML can be used to integrate your COBOL program with the data.

- Internet deployment – By combining AcuXML with AcuServer®, your end users can access XML documents residing on remote servers, whether those servers are in a local-area network, wide-area network, or Internet configuration. If you want to display your COBOL-based XML documents on the Web or inside a user's browser, you may do this as well using stylesheets or RDF files. With additional development effort, you may even develop an XML front end to interface with your COBOL program across the Internet. Most commonly, XML is being used as the language of Web services. It provides a standard by which Web services written in any language can communicate.

These are just a few of the many possible uses of XML. The capabilities of the XML language are far-reaching. However you choose to use it, you can make your ACUCOBOL-GT programs ready to read, process, and output XML data.

11.2.2 The XML-to-FD Utility

In order to prepare your COBOL program for use with XML documents, you must first determine the nature of the data in the documents and the meaning of the tags.

The ACUCOBOL-GT Development System includes an **xml2fd** utility to read XML documents and produce FDs and SELECT statements suitable for inclusion in a COBOL program. Because the structure of valid XML documents is very specific and well defined, the utility produces the best results using valid documents.

The FD created by the utility has the same structure as the XML file. The parent/child relationships of all the data elements are present. Array definitions are included. The data types are defined. If a schema is included with the file, the DTD has a high degree of accuracy. If a DTD is included, the utility analyzes the content and makes an intelligent guess at the data types (e.g., numerical data is assigned the NUMBER type, and so on). Raw XML produces FDs that are a best guess based on the structure of the elements. See section 11.2.2.1 for more details on **xml2fd** output.

The utility also creates a SELECT statement for each data file. Once the FDs and SELECTs are created, you should modify them as required, then include them in the COBOL program. When you include the FDs and SELECT statements in your COBOL program, it can understand, process, and even output XML data that is structured like the existing file.

For instructions on how to launch the **xml2fd** utility, refer to section 11.2.2.2.

11.2.2.1 **xml2fd** output

The utility understands three types of XML files:

- XML files with a DTD – When you run XML files with DTDs through the utility, the record structure created for the FD is structurally correct, but may require manual modification in the sizes and types of the data items.

- XML files with an associated schema file or pointers to a schema file that is available to the local system – When you run XML files with schemas through the utility, the resulting FD is structurally correct, but depending on the completeness of the schema may need to be manually modified in order to correct the sizes of the data items. If the schema includes size information, that information is used and no modification is necessary.

  Note that **xml2fd** supports *schema includes*. It detects the attributes "schema=filename" or "nonamespaceschemalocation=filename" and uses the named file to determine the structure of the FDs. However, the file must be available locally. You may need to locate the schema file on the Internet, download it onto your local development machine, and then point to it before you run the XML file through the utility.

- Raw XML files (with no DTD and no schema) – When you run raw files through the utility, the structure created is a best guess based on reading all or part of the XML file and creating a structure from the structure of the elements. The more homogeneous the XML data, the more precise the record structure is. Again, it may require manual modification to fix the sizes.

Note that **xml2fd** ignores attributes of XML tags, because no natural mapping exists between attributes and COBOL data items. In the following code snippet, type="personal" is an attribute that would be ignored:

```
<customer type="personal">
     <name>Acu</name>
</customer>
```

The **xml2fd** utility creates two files: a ".fd" file and a ".sl" file. The ".fd" file contains the FD and record structure for the data file. The ".sl" file contains the file type and SELECT for the data file. The base name of the files are the same as the base name of the XML file provided. For example, "mydata.xml" produces "mydata.fd" and "mydata.sl". It is your responsibility to include the output files in your COBOL program.

11.2.2.2 **xml2fd** command options

You can invoke the **xml2fd** utility from the command line or from the AcuBench® Tools menu. To invoke it from the command line, type:

```
xml2fd options xmlfile ...
```

where:

"xmlfile ..." is one or more XML files. For each XML file named, separate ".fd" and ".sl" files are created.

Note that **xml2fd** is not designed to work with remote schema files. If the XML file refers to a remote schema file (using http sytnax), the **xml2fd** utility will crash when trying to open the file. Make sure that the XML file used on this command line refers to a local schema or no schema at all.

Options include:

-d *output_directory*	Use this option to name the directory into which the ".fd" and ".sl" files should be placed. The default is the current directory. Note that the "-f" and "-s" options described below override this value.
-f *fd_directory*	Use this option if you want the ".fd" file to be placed into a different directory than the ".sl" file. The ".fd" file is then stored in the named directory.
-s *sl_directory*	Use this option if you want the ".sl" file to be placed into a different directory than the ".fd" file. The ".sl" file is then stored in the named directory.
-n *count*	Use this option to specify the number of records to read from the XML file before calculating the sizes of data items and table occurrences and arriving at field descriptions in the FD. In very large XML files, it may not be desirable to read the entire file to derive the FD. On the other hand, it may be necessary to read more than one record to confirm a "best guess"

for intended field descriptions in a record. The more data the utility reads, the better the guess will be, but the more time it will take to process the data and create the file. The value of "-n" must be numeric and greater than "0".

-o *occurs* Use this option to define a default value for any tables found in the XML file. Designating a numeric value greater than "0" instructs **xml2fd** to look for cases within the record where it is appropriate to assign an OCCURS clause, and gives a default number to assign in the clause. Schema parsed uses any *maxOccurs* attributes found, but DTDs don't have the syntax to describe a maximum number of occurrences. If records are read and the number of occurrences in any record is larger than the value you enter here, the larger value is used.

Please note that in complex cases where an FD has more than one OCCURS clause and where the number of OCCURS differs, you must manually change the generated representation of the FD.

-p prefix Use this option to prepend all data items in the FD with a standard prefix. This option is especially useful if the XML file has elements that use reserved words as their names. For example, If your XML file looks like the following:

```
<company>
  <name>Acucorp</company>
  <address>8515 Miralani
Drive</address>
  <city>San Diego</city>
  <state>CA</state>
  <ZIP>92126</ZIP>

  <remarks>This company sells
```

```
development
   products for COBOL</remarks>
</company>
```

Then **xml2fd** will generate an FD that looks like:

```
07 company.
   09 name pic x(30).
   09 address pic x(30).
   09 city pic x(15).
   09 state pic x(2).
   09 ZIP pic 9(5).
   09 remarks pic x(128).
```

However, ADDRESS and REMARKS are COBOL reserved words, so this would not compile. Instead, an error such as this would result: "Unexpected character. System will ignore it."

If you use "-p cust-" when running **xml2fd**, then the utility generates:

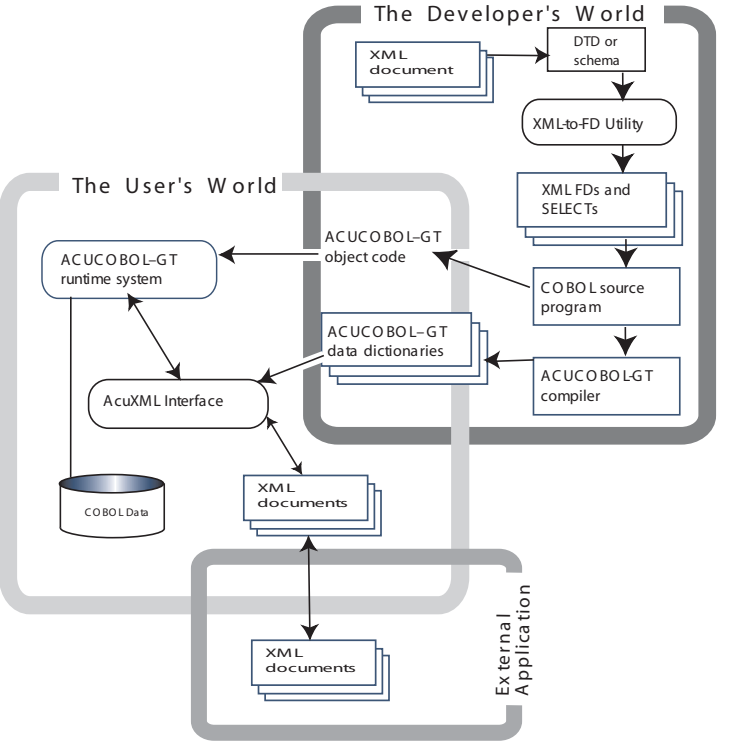
```
07 cust-company.
   09 cust-name pic x(30).
   09 cust-address pic x(30).
   09 cust-city pic x(15).
   09 cust-state pic x(2).
   09 cust-ZIP pic 9(5).
   09 cust-remarks pic x(128).
```

Using this option causes XFD NAME directives to be placed into the ".fd" file so that the resulting XFD file has data item names that match the XML file.

11.2.3 The AcuXML Interface

The AcuXML interface is a transparent interface between ACUCOBOL-GT applications and XML documents. Part of the standard ACUCOBOL-GT runtime system, AcuXML dynamically reads and generates XML using data dictionaries created at compile time. As the ACUCOBOL-GT runtime

module executes your COBOL application, the interface runs “behind the scenes” to match up the requirements of both the application and the data. You do not need to know XML to use the interface.



Transparent Access to XML Data

With the interface, an ACUCOBOL-GT program can read whichever XML files a user indicates. Because XML files are by nature sequential, AcuXML can read the files specified for input, process them, and return files in the desired data format.

The ACUCOBOL-GT program can also open any COBOL file that is required for a user request (whether it is indexed, relative, or sequential), read and process that data, and output a sequential file that is in XML format. Using configuration variables, you can specify which files you want to be

XML documents, and whether you want ACUCOBOL-GT to generate raw XML, XML documents with DTDs, or XML documents with schemas. For our purposes, the XML documents are considered transient—more of a data transport mechanism than a data storage mechanism.

Please note that reading an XML file and then writing it as XML with the same data does not necessarily produce an identical file. The information is the same, but the markup or "wrappers" likely are not.

11.2.3.1 Data dictionaries

ACUCOBOL-GT's XML interface depends on XFDs, or data dictionaries, to map COBOL records to the XML data type. These dictionaries are based on the standard COBOL file descriptors (FDs). (Refer to section 5.3 of the *ACUCOBOL-GT User's Guide* for more details.) These files are created at compile time. The structure of XFDs used by **alfred** and the Acu4GL® and AcuXDBC® interfaces has been enhanced in order to mirror the rich structure of XML.

To create the XFDs, you specify "-Fa" on the command line when you compile your COBOL source. This option signals the compiler to create an XFD file for every indexed, relative, or sequential file contained in the program. To prevent naming errors, you should also include the "-Fc" option, which tells the compiler that the field names in the resulting XFDs must match the element names in the COBOL source exactly.

XFDs include information about the structure of their associated data files. They provide a map between the COBOL files used by the program and the external data file that the program receives (in this case, an XML document).

Using the XFDs and the new XML interface, the COBOL program can open an XML document and read from it or write to it using COBOL file I/O syntax.

To control precisely how the XFDs are built, you can use data dictionary directives with AcuXML, as you can with any ACUCOBOL-GT file system interface. Directives are optional comments that you place into your FDs in your COBOL source code. (See section 4.1 of the *ACUCOBOL-GT User's Guide* for more information.)

For instance, to store items at group level, you can change the database table names in the XFDs. Likewise, you can store dates as database "date" fields. This is all done with directives. Note that if you use the XFD Name directive to define XML data elements, you must use the name exactly as it appears in the XML tag, including case. If you do not match the name or case exactly, the FD items do not get filled when the interface parses the XML data.

11.2.3.2 AcuXML configuration variables

ACUCOBOL-GT includes several configuration variables for configuring the XML output that is generated by the AcuXML interface. These include:

Configuration Variable	**Purpose**
AXML_CREATE_STYLE	Defines whether a DTD or schema is included with the XML output that is generated
AXML_CREATE_SCHEMA	Tells AcuXML whether to create a schema file, or simply include the name of a schema file in the output
AXML_SCHEMA_DOC	Adds a documentation element to the schema if created
AXML_SCHEMA_NAME	Defines the name of the schema file to generate, if any
AXML_SCHEMA_NAMESPACE_DATA	Defines the precise schema namespace string to include in the XML output
AXML_ENCODING	Specifies a character encoding method for the XML files that AcuXML creates
AXML_STYLESHEET_TYPE	Adds a sytlesheet comment to the beginning of generated XML files. This associates a style sheet with the XML documents.
AXML_STYLESHEET_HREF	Names the XML style sheet to use
AXML_IGNORE_EMPTY_DATA	When writing tags, ignores data items that are all blank (in the case of alpha data) or "0" (in the case of numeric data)

All of the variables are optional. If desired, you include them in your runtime configuration file, just as you would any ACUCOBOL-GT configuration variable. In addition, you should also use the configuration file to specify the location of your XML files, which particular files are of XML type, and perform additional configuration as desired. See section 11.2.4 below for more details on configuring the runtime system for XML data.

Refer to Appendix H in *ACUCOBOL-GT Appendices* for usage details on all of the configuration variables that are listed here.

11.2.4 Using AcuXML

To interact with XML data from an ACUCOBOL-GT application, you must prepare your application, then set up and configure the end-user system. Your program can be designed to read XML data and/or generate XML output.

To enable your program to read XML data:

1. Prepare your ACUCOBOL-GT application.

   a. Obtain a representative XML document from the anticipated source of your files. For the best results, the document should refer to a DTD or schema.

   b. Use the **xml2fd** utility to create FDs and SELECT statements for the XML document. Refer to section 11.2.2 for more information on using this utility.

   c. Include the new FDs and SELECT statements in your program as you would any FD and SELECT.

   d. Compile your program with the "-Fa" option specified. This option tells the compiler to generate XFDs for each COBOL data file included in the program. The "-Fa" option is described in section 2.1.6 of the *ACUCOBOL-GT User's Guide*. To prevent naming errors, you should also include the "-Fc" option, which tells the compiler that the field names in the resulting XFDs must match the element names in the COBOL source exactly, including case, hyphen usage, etc.

2. Set up and configure the runtime system.

   a. Install the ACUCOBOL-GT runtime, object code, data dictionaries, and COBOL data files on the end-user system.

   b. Acquire XML data from your data source, or if reading data from the Internet, acquire the exact URL of the XML data stream.

   c. Create a configuration file for the runtime as described in section 2.7 of the *ACUCOBOL-GT User's Guide*. In the configuration file:

      - Specify the location of your data files. If the files are local or accessed via AcuServer, use the FILE_PREFIX configuration variable to specify the location. If the files will be accessed over the Internet, map the data files directly to a URL. For example, to read "bookfile.xml" over the Internet, you could add the following line to the configuration file:

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

      - Specify which files should be treated as XML data files by setting the *filename*_HOST variable to "XML", where *filename* is the base name of the XML document with no file extension (e.g., "custdata_HOST XML"). Include a separate entry for each XML document name. Make sure that the files you indicate are sequential files. For example:

```
BOOKFILE_HOST XML
```

      - Tell the runtime to keep the case of your XFD field names intact by setting the 4GL-COLUMN-CASE variable to "unchanged". XML (unlike HTML) is case sensitive. If you do not set 4GL-COLUMN-CASE, the runtime converts the field names to lower case and hyphens to underscores. Then the runtime may not be able to read your XML data properly.

      - Configure the XML output as desired using a set of AcuXML-specific configuration variables. (These variables all start with the "AXML" prefix. See section 11.2.3.2.) With these variables, you can tell the runtime whether to create a DTD or schema when creating XML data, whether to associate a style sheet with the output, and more.

- Perform additional configuration as desired. You can use any configuration variables that affect XFD parsing, such as XFD-PREFIX and XFD-DIRECTORY. Note that XFD files must be available in the named XFD_DIRECTORY even when the XML data stream is read over the Internet via HTTP.

Refer to Appendix H in *ACUCOBOL-GT Appendices* for details on any of these configuration file entries. Following is a sample "cblconfig" file for use with AcuXML:

```
file-prefix           . /usr/data
orderfile-host        xml
bookfile-host         xml
customer-host         xml
4gl-column-case       unchanged
```

3. Run your ACUCOBOL-GT program normally.

The rest is automatic. When your ACUCOBOL-GT program READs a file designated as XML, the AcuXML interface converts the XML data to COBOL using XFDs, your program performs the request, and then returns data in desired format. Your program automatically WRITEs output in sequential XML format as specified in configuration file.

To enable your program to generate XML output:

If your program needs only to output XML data, the process is even simpler:

1. Prepare your ACUCOBOL-GT application.
   a. Decide what data is needed in the XML file.
   b. Write an FD that describes that data in the desired way. Note that if you have a sequential file that already describes the data, this step is not necessary.
   c. COPY the FD into your COBOL program, and compile with the "-Fa" option.
2. Set up and configure the runtime system.
   a. Install the ACUCOBOL-GT runtime, object code, data dictionaries, and COBOL data files on the end-user system.

b. Create a configuration file for the runtime as described in step 2c above. Include output variables, such as:

```
axml-create-style     schema
axml-schema-name      myschema
axml-create-schema    false
axml-stylesheet-type  text/css
axml-stylesheet-href  mystyle.css
```

3. Run your ACUCOBOL-GT program normally.

11.2.4.1 AcuXML output structures

When AcuXML generates XML data, the hierarchical structure of the XML file matches the record structure of the COBOL file. For example, **iobench** has a sequential file SEQ1, whose record structure is defined as:

```
01  SEQ-1-RECORD.
   03  SEQ-1-KEY                      PIC 9(10).
   03  SEQ-1-ALT-KEY.
       05  SEQ-1-ALT-KEY-A            PIC X(30).
       05  SEQ-1-ALT-KEY-B            PIC 9(10).
   03  SEQ-1-BODY                     PIC X(50).
```

If this file were written as an XML file, a typical record would look like this:

```
<SEQ-1-RECORD>
  <SEQ-1-KEY>20</SEQ-1-KEY>
  <SEQ-1-ALT-KEY>
    <SEQ-1-ALT-KEY-A>032472140976086473026412339002
    </SEQ-1-ALT-KEY-A>
    <SEQ-1-ALT-KEY-B>20</SEQ-1-ALT-KEY-B>
  </SEQ-1-ALT-KEY>
<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrst
</SEQ-1-BODY>
</SEQ-1-RECORD>
```

Note that in XML, it is possible for two data elements to have the same name. For instance:

```
<Lender phone="607.555.2222">
    <name>Doug Glass</name>
    <street>416 Disk Drive</street>
    <city>Medfield</city>
    <state>MA</state>
</Lender>
```

```
<Borrower phone="310.555.1111">
    <name>Britta Regensburg</name>
    <street>219 Union Drive</street>
    <city>Medfield</city>
    <state>CA</state>
</Borrower>
```

However, XFDs are designed to mirror the structure of databases which do not allow duplicate element names. For this reason, compiling a COBOL program with this record structure will result in an XFD compiler warning. If you are working with XML data files, you may disregard this warning.

11.2.4.2 Restrictions

Some restrictions are associated with reading or writing an XML document from ACUCOBOL-GT programs. For instance, programs can open files INPUT or OUTPUT, but not I-O or EXTEND. Attempting to open a file EXTEND or I-O fails and returns a NO-SUPPORT error (error 9B in the ANSI-85 code set).

In addition, the XML file system interface can write only sequential XML files. Because XML is intended as a data delivery mechanism and not a data store mechanism, this is not much of a limiting factor.

As any XML parser would, this interface fails to read a record if a parsing error occurs. The error returned is 9D,05 in this case. See section 11.2.5, "AcuXML Error Reporting," for a list of specific parsing errors.

Finally, the XML data file to be read must contain only a single document. Do not try to concatenate documents into a single data file. The top-level element of the XML document corresponds to a sequential file. Each element at the next level corresponds to a record in that file.

For example, the SEQ1 file "iobench" looks like the following after three records have been written to the file:

```
<SEQ-1-FILE>
 <SEQ-1-RECORD>
   <SEQ-1-KEY>10</SEQ-1-KEY>
   <SEQ-1-ALT-KEY>
     <SEQ-1-ALT-KEY-A>16442491499168412304649263901 4
   </SEQ-1-ALT-KEY-A>
```

```
      <SEQ-1-ALT-KEY-B>10</SEQ-1-ALT-KEY-B>
    </SEQ-1-ALT-KEY>

<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
</SEQ-1-BODY>
 </SEQ-1-RECORD>
 <SEQ-1-RECORD>
   <SEQ-1-KEY>20</SEQ-1-KEY>
   <SEQ-1-ALT-KEY>
     <SEQ-1-ALT-KEY-A>03247214097608647302641233900 2
   </SEQ-1-ALT-KEY-A>
     <SEQ-1-ALT-KEY-B>20</SEQ-1-ALT-KEY-B>
   </SEQ-1-ALT-KEY>

<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
</SEQ-1-BODY>
 </SEQ-1-RECORD>
 <SEQ-1-RECORD>
   <SEQ-1-KEY>30</SEQ-1-KEY>
   <SEQ-1-ALT-KEY>
     <SEQ-1-ALT-KEY-A>11064097190428274300669213901 0
   </SEQ-1-ALT-KEY-A>
     <SEQ-1-ALT-KEY-B>30</SEQ-1-ALT-KEY-B>
   </SEQ-1-ALT-KEY>

<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
</SEQ-1-BODY>
 </SEQ-1-RECORD>
</SEQ-1-FILE>
```

Note that the end of the file in this example is signaled by the XML command, </SEQ-1-FILE>. ACUCOBOL-GT ignores other documents in the file. For example, if you have an XML file which looks like the this:

```
<SEQ-1-FILE>
 <SEQ-1-RECORD>
   <SEQ-1-KEY>10</SEQ-1-KEY>
   <SEQ-1-ALT-KEY>
     <SEQ-1-ALT-KEY-A>16442491499168412304649263901 4
     </SEQ-1-ALT-KEY-A>
     <SEQ-1-ALT-KEY-B>10</SEQ-1-ALT-KEY-B>
   </SEQ-1-ALT-KEY>

<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
</SEQ-1-BODY>
```

```
 </SEQ-1-RECORD>
</SEQ-1-FILE>
<SEQ-1-FILE>
 <SEQ-1-RECORD>
   <SEQ-1-KEY>20</SEQ-1-KEY>
   <SEQ-1-ALT-KEY>
     <SEQ-1-ALT-KEY-A>03247214097608647302641233900 2
     </SEQ-1-ALT-KEY-A>
     <SEQ-1-ALT-KEY-B>20</SEQ-1-ALT-KEY-B>
   </SEQ-1-ALT-KEY>

<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
</SEQ-1-BODY>
 </SEQ-1-RECORD>
</SEQ-1-FILE>
<SEQ-1-FILE>
 <SEQ-1-RECORD>
   <SEQ-1-KEY>30</SEQ-1-KEY>
   <SEQ-1-ALT-KEY>
     <SEQ-1-ALT-KEY-A>11064097190428274300669213901 0
     </SEQ-1-ALT-KEY-A>
     <SEQ-1-ALT-KEY-B>30</SEQ-1-ALT-KEY-B>
   </SEQ-1-ALT-KEY>

<SEQ-1-BODY>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
</SEQ-1-BODY>
 </SEQ-1-RECORD>
</SEQ-1-FILE>
```

then ACUCOBOL-GT ignores everything after the tenth line, because the </SEQ-1-FILE> XML command signaled the end of the file. You should use the C$XML library routine instead of AcuXML to read files like these.

11.2.5 AcuXML Error Reporting

If AcuXML encounters a parsing error, it returns the error code “9D05,nn” where *nn* is one of the following:

1	Out of memory
2	Syntax error

3	No element found
4	Not well formed (invalid token)
5	Unclosed token
6	Partial character
7	Mismatched tag
8	Duplicate attribute
9	Junk after document element
10	Illegal parameter entity reference
11	Undefined entity
12	Recursive entity reference
13	Asynchronous entity
14	Reference to invalid character number
15	Reference to binary entity
16	Reference to external entity in attribute
17	XML processing instruction not at start of external entity
18	Unknown encoding
19	Encoding specified in XML declaration is incorrect
20	Unclosed CDATA section
21	Error in processing external entity reference
22	Document is not standalone
23	Unexpected parser state – please send a bug report
24	Entity declared in parameter entity

11.2.6 Using the C$XML Routine

The C$XML library routine is designed for those who want low-level control over the parsing of XML data. It lets you define precisely which elements or attributes of the data to parse. C$XML can be used for both record- and non-record-based XML documents.

Appendix I in *ACUCOBOL-GT Appendices* contains detailed descriptions of the library routine's individual operation codes, parameters, usage, etc. The following sections describe how to use the C$XML library routine in more general terms.

11.2.6.1 General procedure

When using the C$XML routine, you should perform the following basic functions:

1. Parse an XML file.
2. Move to an element in the file.
3. Retrieve data from the element.
4. Add, modify, or delete data as desired.
5. Write to the file to save your changes.
6. Release the parser from memory.

If an error occurs, you can also retrieve error information. If desired, you can retrieve element attributes or comments as well.

Each function is performed through C$XML operation codes (op-codes) as described in sections 11.2.6.2 through 11.2.6.10. Most C$XML operations return element handles. A separate handle is provided for each element. You should move the return codes to an ACUCOBOL-GT handle that you've declared in Working-Storage. The handle should be defined as USAGE IS HANDLE. You will refer to these handles when you move deeper into a record and retrieve or update specific data.

Refer to section 11.2.6.12 for sample code.

Note: You don't necessarily have to open a file to read XML data. You can parse a string directly using the CXML-PARSE-STRING op-code.

11.2.6.2 Understanding C$XML terminology

The C$XML routine uses XML terminology that may not be familiar to you—terms like *element*, *attribute*, *parent*, *child*, and *sibling*. To understand the terminology of C$XML, consider the following XML file (line numbers added for discussion purposes):

```
 1 <?xml version="1.0"?>
 2 <bookfile
 3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4   xsi:noNamespaceschemaLocation="bookfile.xsd">
 5   <transaction borrowDate="2001-10-15">
 6     <Lender phone="607.555.2222">
 7       <name>Doug Glass</name>
 8       <street>416 Disk Drive</street>
 9       <city>Medfield</city>
10       <state>MA</state>
11     </Lender>
12     <Borrower phone="310.555.1111">
13       <name>Britta Regensburg</name>
14       <street>219 Union Drive</street>
15       <city>Medfield</city>
16       <state>CA</state>
17     </Borrower>
18     <note>Lender wants these back in two weeks!</note>
19     <books>
20       <book bookID="123-4567-890">
21         <bookTitle>Earthquakes for Breakfast</bookTitle>
22         <pubDate>2001-10-20</pubDate>
23         <replacementValue>15.95</replacementValue>
24         <maxDaysOut>14</maxDaysOut>
25       </book>
26       <book bookID="123-4567-891">
27         <bookTitle>Avalanches for Lunch</bookTitle>
28         <pubDate>2001-10-21</pubDate>
29         <replacementValue>19.99</replacementValue>
30         <maxDaysOut>14</maxDaysOut>
31       </book>
32       <book bookID="123-4567-892">
33         <bookTitle>Meteor Showers for Dinner</bookTitle>
34         <pubDate>2001-10-22</pubDate>
35         <replacementValue>11.95</replacementValue>
36         <maxDaysOut>14</maxDaysOut>
37       </book>
38       <book bookID="123-4567-893">
```

```
39          <bookTitle>Snacking on Volcanoes</bookTitle>
40          <pubDate>2001-10-23</pubDate>
41          <replacementValue>17.99</replacementValue>
42          <maxDaysOut>14</maxDaysOut>
43        </book>
44      </books>
45    </transaction>
46 </bookfile>
```

When you call C$XML with the CXML-PARSE-FILE op-code, you get a handle to this entire file.

An “element” is a name that appears immediately after a less than sign (<), and terminated with a greater than sign (>) or by a forward slash followed by a greater than sign (/>). So “bookfile” (line 2) is the top-level element. “transaction” (line 5) is an element, as is “city” (lines 9 and 15 - these are two separate elements, both named “city”).

There are two different ways in XML to signify that an element is finished:

1. With a tag that is the element name preceded immediately by a less than sign and a forward slash (</).
2. The corresponding greater than sign (>) is immediately preceded by a forward slash (/).

This sample has instances of the first method only. For example, the termination of the bookfile element is on line 46.

An “attribute” is included within the element description as a name followed by an equals sign (=), followed by a quoted string. The name is the attribute name, and the quoted string is the attribute value. There is no limit to the number of space-delimited attribute name/value pairs in an element. For example, the “transaction” element (line 5) has a single attribute whose name is “borrowDate” and whose value is “2001-10-15”.

The “data” of an element is any non-element, non-comment text between the element start and end. Elements that end with a /> can’t have data. And some elements don’t have data. For example, the Lender element (line 6) has no data. The “street” element on line 8 has data of “416 Disk Drive”.

A “child” of an element is an element one level below an element. For example, “Lender” is a child of “transaction”, as is “Borrower”. “name” is a child of “Lender”, and there is a separate element named “name” which is a child of “Borrower”. But neither of those “name” elements is considered a child of “transaction” because there is an element at a level between “name” and “transaction”. There are two separate “city” elements, one of which is a child of “Lender”, and the other a child of “Borrower”. The elements between “city” and “Lender” are all at the same level as “city”, which is why “city” is considered a child of “Lender”.

A “parent” of an element is one who has that element as a child.

A “sibling” of an element is an element with the same parent.

So “transaction” is the parent of “Lender”, “Borrower”, “note” and “books” (lines 6, 12, 18 and 19, respectively). “Lender”, “Borrower”, “note” and “books” are all siblings.

A “record” is child of the top-level element.

11.2.6.3 Parsing an XML file

You can use three main op-codes to parse an XML file with C$XML: CXML-PARSE-FILE, CXML-OPEN-FILE, and CXML-NEW-PARSER. Each has a slightly different function, as described below. Choose the one that best suits your needs.

Op-code	**Description**
CXML-PARSE-FILE	Opens and parses the specified file
CXML-OPEN-FILE	Opens the specified file, so you can parse individual records
CXML-NEW-PARSER	Opens a new, empty XML file

Note: C$XML can parse local or remote files—even files located on the Internet. You simply specify the server, IP address, or URL in the pathname. You cannot write to URLs, however.

If you prefer, you can parse an XML file directly without opening a file with the CXML-PARSE-STRING op-code. Examples for using these op-codes appear in the following paragraphs.

Opening and parsing a file

To open and read an entire XML file, use the CXML-PARSE-FILE op-code. Once a file is parsed, all its elements are immediately retrievable. Parsing entire files can take some time, depending on the size of the file. Use this option when you plan to work extensively with the whole file or when the file is small.

For instance:

```
call "C$XML" using CXML-PARSE-FILE "http://www.nws.noaa.gov/data/current_obs/KMYF.xml"
move return-code to parser-handle
```

Opening a file, parsing individual records

To simply open the file, but not parse it, use the CXML-OPEN-FILE op-code. With the file open, you can then parse individual records using the CXML-PARSE-NEXT-RECORD op-code. For instance:

```
CALL "C$XML" using CXML-OPEN-FILE "http://www.nws.noaa.gov/data/current_obs/KMYF.xml"
move return-code to parser-handle
CALL "C$XML" using CXML-PARSE-NEXT-RECORD parser-handle
move return-code to record-handle
```

This option is more efficient than parsing entire files, but you must remember to parse the record before you try to retrieve its elements.

Creating a new parser

To create a new XML file, use the CXML-NEW-PARSER op-code. An empty file is opened, into which you can add children, siblings, attributes, and comments as described in section 11.2.6.6. Note that the file is not "created" until you write to the file using the CXML-WRITE-FILE op-code.

Parsing an XML string directly

If you get XML text from another source and need to parse it, you can parse the string directly using the CXML-PARSE-STRING op-code. You don't have to write the data to a file, then parse the file. You simply specify the string directly in the call. For example:

```
call "C$XML" using CXML-PARSE-STRING,
"<?xml version=""1.0""?><group1><subgroup1><item1>data</item1></subgroup1></group1>".
move return-code to parse-handle.
```

Then you can use the return code elsewhere in your program.

11.2.6.4 Moving to an element

Once a file has been opened and parsed, you can navigate the file in many different ways. In XML, you must move to an element of interest before you can retrieve its data. Listed below are several op-codes that you can call from the C$XML routine for this purpose. Be sure to specify the handle of the element or parser to move to, as in:

```
call "C$XML" using CXML-GET-FIRST-CHILD
                   parser-handle.
```

Op-code	**Description**
CXML-GET-FIRST-CHILD	Moves to the first child of an element
CXML-GET-NEXT-SIBLING	Moves to the subsequent sibling of an element
CXML-GET-PREV-SIBLING	Moves to the previous sibling; one way to move backward in a file
CXML-GET-PARENT	Moves to the parent; another way to move backward in a file

More advanced options include:

- CXML-GET-CHILD-BY-NAME
- CXML-GET-CHILD-BY-CDATA
- CXML-GET-CHILD-BY-ATTR-NAME

- CXML-GET-CHILD-BY-ATTR-VALUE
- CXML-GET-SIBLING-BY-NAME
- CXML-GET-SIBLING-BY-CDATA
- CXML-GET-SIBLING-BY-ATTR-NAME
- CXML-GET-SIBLING-BY-ATTR-VALUE

These options allow you to navigate to a specific child or sibling in an element rather than simply the next or previous one. Details are provided in Appendix I in *ACUCOBOL-GT Appendices*.

Note that these op-codes do not retrieve element data. They simply move to the element and return the element handle. Once you have the handle to the element of interest, you can call CXML-GET-DATA to retrieve the data associated with the element.

Note: If desired, you can retrieve element attributes and comments as well. Refer to Section 11.2.6.10, "Retrieving attributes,"and Section 11.2.6.11, "Retrieving comments," for more information.

11.2.6.5 Retrieving data

To retrieve data from an element, you use the CXML-GET-DATA op-code. In this operation, you must specify the handle for the element of interest, down to the field level. You can obtain the handle by looking at the return code for the other operations, such as CXML-GET-SIBLING-BY-NAME. For example:

```
*Get the handle for the Weather element
   call "C$XML" using CXML-GET-SIBLING-BY-NAME
                      ele-1-handle
                      "weather"
                      0.
   move return-code to ele-2-handle.
*Get the weather data using that handle
   call "C$XML" using CXML-GET-DATA
                      ele-2 handle
                      throw-away-info
                      weather-val.
```

Once the XML data is returned, you can then pass it to other parts of your COBOL program for processing or display.

11.2.6.6 Adding, modifying, or deleting data

You can add, modify, or delete data in the XML document using any of the following C$XML op-codes:

- CXML-MODIFY-CDATA
- CXML-MODIFY-ATTRIBUTE
- CXML-ADD-CHILD
- CXML-ADD-SIBLING
- CXML-ADD-ATTRIBUTE
- CXML-ADD-COMMENT
- CXML-APPEND-COMMENT (useful for documenting why you modified an XML document)
- CXML-DELETE-ATTRIBUTE
- CXML-DELETE-ELEMENT
- CXML-DELETE-COMMENT

For the majority of these op-codes, you must pass the element handle as a parameter as well as the new value of that element or attribute. Other parameters may apply as well. Refer to Appendix I in *ACUCOBOL-GT Appendices* for more details.

11.2.6.7 Writing a file

If you modify the XML file in any way, you must write to the file in order for your changes to take effect. Use the CXML-WRITE-FILE op-code for this purpose. For example:

```
call "C$XML" using CXML-WRITE-FILE
                   parser-handle
                   "bookfile.xml"
```

You can write to the same file that you opened, or you can write to a new file. The filename that you specify can be anything you want. You cannot, however, write to a URL.

Caution: If you do not write to the file, your changes are lost.

11.2.6.8 Releasing the parser

When you are finished writing to the file, you are responsible for releasing the parser from memory. This operation is not performed automatically. To release the parser, use the CXML-RELEASE-PARSER op-code. For example:

```
call "C$XML" using CXML-RELEASE-PARSER
                   parser-handle.
```

11.2.6.9 Retrieving errors

If a return code from any other operation is “0” or “1”, a call to C$XML has failed. To retrieve error information, call the CXML-GET-LAST-ERROR op-code. For example:

```
77 errors-storage PIC x(60).
77 errors-value PIC x(70).
.
.
.
CALL "C$XML" using CXML-GET-LAST-ERROR error-storage.
move "Err No:" to error-value(1:).
move return-code to errors-value(9:).
move errors-storage to errors-value(20:).
display message " " errors-value.
```

Appendix I in *ACUCOBOL-GT Appendices* lists all of the error codes that can be returned by this library routine.

11.2.6.10 Retrieving attributes

If desired, you can retrieve element attributes using one of the following C$XML op-codes:

Op-code	**Description**
CXML-GET-ATTRIBUTE-COUNT	Gets the number of attributes in an element
CXML-GET-ATTRIBUTE	Gets the next attribute
CXML-GET-ATTRIBUTE-BY-NAME	Gets the named attribute

Attributes have special characteristics:

- Attributes don't have a handle; they are attached to the element handle for which they are an attribute.
- There can be more than one attribute per element, thus more than one attribute per element handle.
- You don't need to call CXML-GET-DATA to fetch the attribute data. The CXML-GET-ATTRIBUTE op-codes retrieve the data.

Before you use CXML-GET-ATTRIBUTE, we recommend that you use CXML-GET-ATTRIBUTE-COUNT. This op-code tells you how many attributes are in the element of interest. Then when you use CXML-GET-ATTRIBUTE, you know which attribute you are getting (the first attribute is number 0, the second number 1, etc.).

Alternatively, if you know the name of the attribute you want and not the number of the attribute, you can use CXML-GET-ATTRIBUTE-BY-NAME. If you have an unknown number of attributes like this:

```
Limousine
xmlns:tsd="http://namespaces.softwareag.com/tamino/TaminoSche
maDefinition"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

and you are searching for a particular attribute, say “xmlns:xsi”, you can search for the value with CXML-GET-ATTRIBUTE-BY-NAME, or you can get the count and read the attributes one by one searching for the one you want.

Refer to Appendix I in *ACUCOBOL-GT Appendices* for more details on the CXML-GET-ATTRIBUTE op-codes.

11.2.6.11 Retrieving comments

You can obtain element comments using CXML-GET-COMMENT op-code. You don’t need to call CXML-GET-DATA to fetch the comment data. The CXML-GET-COMMENT op-code retrieves the data.

11.2.6.12 C$XML examples

Sample 1: Open, parse, and read weather file

The following sample program calls an XML file from the National Weather Service Web site and retrieves weather data for use in a COBOL program. (It does not modify or write to the file in any way.)

```
*Retrieve the xml data and parse it
   call "C$XML" using CXML-PARSE-FILE
       "http://www.nws.noaa.gov/data/current_obs/KMYF.xml
   move return-code to parser-handle

*Move to the data item of the record, which is a
*child of the record name element.
   call "C$XML" using CXML-GET-FIRST-CHILD
                      parser-handle.
   move return-code to ele-1-handle

*Get the desired fields, which are all siblings of the first
*child.  Get the handle to the desired sibling, then get the
*data for that element using that handle.

*General outlook field
   call "C$XML" using CXML-GET-SIBLING-BY-NAME
                      ele-1-handle
                      "weather"
                      0
```

```
           move return-code to ele-2-handle
           call "C$XML" using CXML-GET-DATA
                              ele-2-handle
                              throw-away
                               weather-val.

        *Temperature field
           call "C$XML" using CXML-GET-SIBLING-BY-NAME
                              ele-1-handle
                              "temperature_str"
                              0
           move return-code to ele-2-handle
           call "C$XML" using CXML-GET-DATA
                              ele-2-handle
                              throw-away
                              temp-val.

        *Relative humidity field
           call "C$XML" using CXML-GET-SIBLING-BY-NAME
                              ele-1-handle
                              "relative_humidi"
                              0
           move return-code to ele-2-handle
           call "C$XML" using CXML-GET-DATA
                              ele-2-handle
                              throw-away
                              humid-val.

        *Wind direction field
           call "C$XML" using CXML-GET-SIBLING-BY-NAME
                              ele-1-handle
                              "wind_dir"
                               0
           move return-code to ele-2-handle
           call "C$XML" using CXML-GET-DATA
                              ele-2-handle
                              throw-away
                               wind-dir.

        *Wind speed field
           call "C$XML" using CXML-GET-SIBLING-BY-NAME
                              ele-1-handle
                              "wind_mph"
                              0
        move return-code to ele-2-handle
```

```
   call "C$XML" using CXML-GET-DATA
                      ele-2-handle
                      throw-away
                       wind-mph.

*Visibility field
   call "C$XML" using CXML-GET-SIBLING-BY-NAME
                      ele-1-handle
                      "visibility"
                      0
   move return-code to ele-2-handle
   call "C$XML" using CXML-GET-DATA
                      ele-2-handle
                      throw-away
                       vis-val.
```

Sample 2: Create file, add elements and attributes, write data to file

The following sample shows how to create an XML file, add elements to it, and write data to the elements.

```
program-id. test.
working-storage section.
01 parser-handle usage is handle.
01 element-handle usage is handle.
COPY "acucobol.def".
procedure division.
main-logic.
*Create a new XML file
   call "C$XML" using CXML-NEW-PARSER
   move return-code to parser-handle.

*Add a top element (using the name of the file)
   call "C$XML" using CXML-ADD-CHILD
                  parser-handle
                  "custRec"
   move return-code to element-handle.

*Add some namespace information
   call "C$XML" using CXML-ADD-ATTRIBUTE
                  element-handle
                  "xmlns:xsi"
```

```
                    "http://www.w3.org/2001/XMLSchema-instance".

      *Add the first field of the record, which will be a child of
      *the last element.
         call "C$XML" using CXML-ADD-CHILD
                       element-handle
                       "cus-key"
                       "555-55-5555"
         move return-code to element-handle.

      *Add the rest of the record
         call "C$XML" using CXML-ADD-SIBLING
                       element-handle
                       "cus-name"
                       "Acucorp"
         move return-code to element-handle.

         call "C$XML" using CXML-ADD-SIBLING
                       element-handle
                       "cus-addr"
                       "8515 Miralani Drive"
         move return-code to element-handle.

         call "C$XML" using CXML-ADD-SIBLING
                       element-handle
                       "cus-city"
                       "San Diego"
         move return-code to element-handle.

         call "C$XML" using CXML-ADD-SIBLING
                       element-handle
                       "cus-state"
                       "CA"
         move return-code to element-handle.

         call "C$XML" using CXML-ADD-SIBLING
                       element-handle
                       "cus-zip"
                       "92126"

      *write the file
         call "C$XML" using CXML-WRITE-FILE,
                       parser-handle
                       "custRec.xml".
```

```
call "C$XML" using CXML-RELEASE-PARSER,
                   parser-handle.
stop run.
```

11.3 Working with Relational Data

We provide two main ways to integrate COBOL applications with relational databases based on Structured Query Language (SQL):

- using the seamless Acu4GL interface
- using embedded SQL (ESQL) and an ESQL pre-compiler

While the Acu4GL interface is designed to provide easy access to SQL data from ACUCOBOL-GT applications, embedded SQL gives you more precise control over your database queries. The main difference is that you do not need to know SQL with Acu4GL. With ESQL, you embed SQL into your COBOL code, then run a pre-compiler to perform the I/O conversions.

Some sites use a combination of Acu4GL and ESQL to optimize data access for each application. Output from Acu4GL can even be used as a template for ESQL.

11.3.1 Acu4GL Interface

We offer a family of interface products that can automatically convert COBOL I/O into database-specific SQL requests and vice versa. These include:

- Acu4GL for Oracle®
- Acu4GL for Informix®
- Acu4GL for Sybase®
- Acu4GL for Microsoft® SQL Server
- Acu4GL for ODBC

The first four Acu4GL products act as seamless interfaces between ACUCOBOL-GT applications and specific RDBMSs. They dynamically generate industry standard SQL from COBOL I/O statements using data dictionaries created at compile time. As the ACUCOBOL-GT runtime module executes your COBOL application, Acu4GL runs "behind the scenes" to match up the requirements of both the application and the database. You do not need to know SQL to use Acu4GL.

Acu4GL for ODBC works the same as the other Acu4GL products, except that it interfaces between ACUCOBOL-GT applications and Open Database Connectivity (ODBC)-compliant data sources. See section 11.4 for more information on working with ODBC data sources.

See the *Acu4GL User's Guide* for instructions on using any of the Acu4GL interfaces.

11.3.2 Embedded SQL

Developers interested in generating SQL themselves can embed database-specific SQL directly into their ACUCOBOL-GT programs. This is known as "embedded SQL" or "ESQL." Most database vendors offer a pre-compiler that translates ESQL into external CALLs recognized by the RDBMS. We provide an ESQL pre-compiler known as AcuSQL®.

ESQL can be used by itself or with Acu4GL to add specialized control over database functions. ACUCOBOL-GT fully supports the use of ESQL to provide database connectivity.

11.3.2.1 Embedding SQL statements into ACUCOBOL-GT

Although the process for using ESQL in ACUCOBOL-GT programs varies slightly for each database, the general process is as follows:

1. Embed the SQL statements into the ACUCOBOL-GT program. (Your database vendor can supply a list of supported SQL statements.)
2. Run a pre-compiler to translate the SQL into COBOL CALL statements. See section 11.3.2.2 for a list of supported pre-compilers.
3. Compile the new program using ACUCOBOL-GT.

4. Relink the COBOL runtime with the appropriate database access library routines.

5. Execute your program normally.

To guide you through the process of compiling and relinking, we can supply you with instructions specific to your database. Relinking typically involves updating an Acucorp-supplied interface file, updating your system makefile (or batch or command file), and rebuilding the runtime. With ACUCOBOL-GT, you relink the runtime only once, not for each program that uses SQL. This results in much smaller object files.

11.3.2.2 Supported ESQL pre-compilers

ACUCOBOL-GT works well with many different types of ESQL. Listed below are some of the pre-compilers available to our customers:

- Oracle Pro*COBOL (certified by Oracle for use with ACUCOBOL-GT)
- Informix-ESQL for COBOL
- Sybase Embedded SQL/COBOL
- Our AcuSQL (for IBM DB2, Microsoft SQL Server, and ISO/ANSI SQL92 compliant data sources)

If your database vendor offers a COBOL-ESQL pre-compiler, you can effectively link your ACUCOBOL-GT application to that database using ESQL. If your vendor does not offer a pre-compiler, you may still be able to access the database through a group of C routines that can be CALLed directly by the COBOL compiler.

For more information on using AcuSQL, refer to the *AcuSQL User's Guide*. For more information on CALLing C routines, see Chapter 6 of this guide.

11.4 Working with ODBC Data

ACUCOBOL-GT can interface with many ODBC-compliant data sources, such as Microsoft Access, Oracle, and Informix. It does this through Acu4GL for ODBC, a transparent interface that translates COBOL I/O statements into ODBC calls to Windows-based data sources.

With Acu4GL for ODBC, your Windows-based application can access the most common database formats found on desktop computer systems, as well as relational database management systems on UNIX or Windows NT servers.

Acu4GL for ODBC gives ACUCOBOL-GT users access to information stored in data sources that comply with the ODBC standard, a library of standardized data access functions designed for the Windows operating system environment. This capability gives ACUCOBOL-GT users access to data in multiple data sources, databases, and file systems—all from a single interface—and provides access to many Windows-based data sources.

For more information, please refer to the *Acu4GL User's Guide.*

11.5 Working with File Systems like C-ISAM and KSAM

We provide interfaces between ACUCOBOL-GT programs and data on file systems such as C-ISAM and KSAM.

C-ISAM files can be accessed through a special interface routine that is linked into the ACUCOBOL-GT runtime system and is invoked when your program is executed. You do not need to embed C-ISAM commands in your code. This interface is described in a supplement to the ACUCOBOL-GT manual set titled, "Interfacing to the C-ISAM File System."

KSAM files can be accessed through HP e3000 system intrinsic functions or via standard COBOL I/O statements. Specific instructions for accessing these files are provided in section 4.2.3 of *Transitioning to ACUCOBOL-GT.*

11.6 Working with an EXTFH Interface

You can readily configure the ACUCOBOL-GT runtime to use an EXTFH interface for communicating with external file systems that contain indexed, relative, or sequential files. The EXTFH interface is enabled and linked into the runtime by default.

The EXTFH interface provides a way for applications to transparently access a file system such as DB2 for record storage. In transaction processing environments, EXTFH is typically used to handle data access for batch programs. For online programs, you can easily add data access methods to an application.

11.6.1 Using the EXTFH Interface

Two basic requirements are needed to configure the ACUCOBOL-GT runtime to utilize the EXTFH interface:

1. Specify the file system(s) that is using the EXTFH interface to access files.
2. Dynamically load the EXTFH libraries using runtime configuration variables or by statically linking the EXTFH-compatible libraries with the runtime.

After you configure the runtime, the COBOL data access methods are unchanged. The same COBOL file I/O statements are used whether accessing native file systems or using EXTFH.

11.6.2 Making EXTFH Libraries Available to the Runtime

To use an EXTFH interface, you must make a library available to the runtime that implements an EXTFH-compatible function and include all the necessary support libraries.

Note: The vendor of your environment must supply the libraries that contain EXTFH-compatible functions for any data sources you want to access. ACUCOBOL-GT does not supply these libraries.

Use the following procedures to make the libraries available:

1. Specify that the file system is EXTFH.
2. Specify the library using one of these options:
   a. Rely on the default EXTFH function names or specify one or more function names in a configuration variable.
   b. Specify a single EXTFH library name or different library names for relative, sequential, and/or indexed files using configuration variables.
   c. Relink the ACUCOBOL-GT runtime.

We recommend that you set the A_EXTFH runtime configuration variable to dynamically load the EXTFH library or use other A_EXTFH configuration variables to specify different libraries for specific file types.

Regardless of what method you use, you also need to set at least one of the file system configuration variables to access files through the EXTFH interface. See section 11.6.2.1, "Accessing files through EXTFH," or Appendix H in *ACUCOBOL-GT Appendices* for more information.

11.6.2.1 Accessing files through EXTFH

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 the following configuration variables to "EXTFH":

DEFAULT_FILESYSTEM
DEFAULT_IDX_FILESYSTEM
DEFAULT_REL_FILESYSTEM
DEFAULT_SEQ_FILESYSTEM
*filename*_FILESYSTEM

Note that when you use *filename*_FILESYSTEM, the file suffix is not included in the configuration variable definition. If the filename is "outputfile.tmp", for example, set this variable as:

```
outputfile_FILESYSTEM
```

To use the DB2 library to access indexed files and the ACUCOBOL-GT native file handler for sequential and relative files, set the following two configuration variables:

```
A_EXTFH_LIB=/pathname/libraryname.o
DEFAULT_IDX_FILESYSTEM=EXTFH
```

For more information on these configuration variables, see Appendix H in *ACUCOBOL-GT Appendices*.

11.6.2.2 Searching for function names

By default, the runtime searches for functions named "cics_xfh", "cobol_extfh", and "EXTFH" in that order. If your EXTFH function uses a different name, you may also need to set a configuration variable to specify the function name (such as A_EXTFH_FUNC). If necessary, you can also specify a different function name for relative, sequential, and/or indexed files. For more information on A_EXTFH_FUNC and related configuration variables, see Appendix H in *ACUCOBOL-GT Appendices*.

Specifying a single EXTFH library name

The simplest way to implement the EXTFH interface is to specify the file name or absolute path of a single EXTFH shared object library as the value of A_EXTFH_LIB. When you use this method, the runtime loads the specified library in order to find the necessary EXTFH function or functions.

For example, to use the EXTFH library supplied with TXSeries CICS for DB2, specify:

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

where "libxfhdb2_shr.o" contains the entry point to the DB2 EXTFH library and resides in the "libxfhdb2sa.a" AIX archive. In this case, if your LIBPATH variable includes the path where the library resides, you can omit the path; otherwise, include the path as part of A_EXTFH_LIB.

You can also specify the EXTFH library as an environment variable. For example, you can specify the same EXTFH library for accessing DB2 files from TXSeries using:

```
export A_EXTFH_LIB="/usr/lpp/cics/lib/libxfhdb2sa.a(libxfhdb2_shr.o)"
```

Note: An AIX archive file (".a") can contain shared objects. You must specify the shared object in parentheses after the archive name unless you link to the archive.

11.6.2.3 Setting libraries for indexed, relative, and sequential files

To use different EXTFH libraries for indexed, relative, and sequential files, you can set the file names or paths of the libraries in one or more of the following variables:

A_EXTFH_IDX_LIB
A_EXTFH_REL_LIB
A_EXTFH_SEQ_LIB

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.

Remember that when you use configuration variables to identify the library names and specify file names instead of absolute paths, the operating system must be able to find the files in the locations it normally searches for shared libraries. You may need to set the correct search path environment variable, such as LIBPATH on AIX.

Calling conventions for DLLs

All the configuration variables that allow you to specify an EXTFH library name also allow you to specify the calling convention for a particular DLL and thereby override the setting of the DLL_CONVENTION runtime configuration variable. For more information, refer to Appendix H in *ACUCOBOL-GT Appendices*.

11.6.2.4 Statically linking EXTFH-compatible libraries

If you prefer to statically link the EXTFH-compatible libraries into the ACUCOBOL-GT runtime, you can use the **makerun** utility. For example, enter:

```
cd $ACUCOBOL/lib
makerun -bE:extfh.exp -L/usr/lpp/cics/lib -lxfhdb2sa -lcicssa
```

where "extfh.exp" is a text file that contains a single line with the name of the EXTFH function.

When the runtime initializes, it searches for the EXTFH functions in the specified libraries. If it cannot find the EXTFH functions or if you have not specified any libraries, the runtime searches all symbols in the current process.

Note: If you are familiar with using the **make** utility, you can also choose to modify the EXTFH_LIB and EXTFH_FLAGS macros in $ACUCOBOL/lib/Makefile and use **make** to relink the runtime.

For more information on relinking the runtime, see Chapter 6.

11.7 File System Configuration

As described previously in this chapter, you can use many other file systems in place of, or in combination with ACUCOBOL-GT's Vision file system. Supported file systems include:

XML	enabled and available on all platforms
RMS	used in place of Vision on VMS and OpenVMS platforms
EXTFH interface	enabled and available on all platforms; allows access to most files for which an EXTFH library is available
MPE/KSAM	enabled on HP MPE/iX platforms in addition to Vision

C-ISAM	licensed separately; available on most UNIX/Linux platforms
P.SQL/Btrieve	licensed separately; available on most Windows platforms
DB2	licensed separately; through Acu4GL, or by embedded SQL and the DB2 preprocessor
Oracle	licensed separately; through Acu4GL, or by embedded SQL and the Oracle preprocessor
MS SQL	licensed separately; through Acu4GL, or by embedded SQL with AcuSQL
Informix	licensed separately; through Acu4GL, or by embedded SQL with AcuSQL
Sybase	licensed separately; through Acu4GL, or by embedded SQL with AcuSQL
ODBC-compliant	licensed separately; through Acu4GL, or by embedded SQL with AcuSQL

On most platforms, ACUCOBOL-GT comes preconfigured to work with Vision, XML, and EXTFH. In addition, on HP MPE/iX systems the MPE/KSAM file system is enabled. On VMS and OpenVMS systems, Vision is replaced by RMS. For more information about Vision, see section 6.1 in Book 1, *ACUCOBOL-GT User's Guide*. For more information about XML, see section 11.2 in this chapter. For more information on the EXTFH interface, see section 11.6 in this book. For information about MPE/KSAM, see Chapter 4, “HP COBOL Conversions,” in *Transitioning to ACUCOBOL-GT.*

Configuration information for file systems that use a separately licensed product is included in the documentation provided with that product. For information about Acu4GL, AcuSQL, and compatible embedded SQL technologies, see their respective user’s guides, or contact your Micro Focus ***extend*** Sales Professional.

11.8 File System Initialization

By default, with the exception of the EXTFH interface, ACUCOBOL-GT initializes enabled file systems upon runtime startup, before the first COBOL program begins execution. Initialization of the EXTFH interface, by default, is deferred until the first file operation on the file system. This provides the best runtime performance.

If you want to force initialization of EXTFH at startup, or you want to defer initialization of another file system, you can do so by setting a variable in "filetbl.c" and relinking the runtime. "filetbl.c" holds the ACUCOBOL-GT file system table. Settings in "filetbl.c" determine which file systems are enabled and when each system is initialized. "filetbl.c" is located in the lib subdirectory of your ACUCOBOL-GT installation.

To specify file system initialization for a given file system:

1. Open "filetbl.c" in a text editor.
2. Locate the TABLE_ENTRY for the file system you want to control, and change the *defer_init* parameter to the desired value. A value of "0" causes the file system to be initialized at startup. A value of "1" causes file system initialization to be deferred. The *defer_init* flag is set to "0" by default for all file systems except EXTFH.

   For example, following is the TABLE_ENTRY for Vision:

   ```
   TABLE_ENTRY  file_table[] = {
   #if   USE_VISION
      {   &v5_dispatch,   "VISIO", 0 },
   #endif /* USE_VISION */
   ```

   The *defer_init* value is the value that follows "VISIO" on the third line.
3. When you have made your changes, save and close the file and relink the runtime. Instructions on relinking the runtime are located in section 6.3.6, "Relinking the Runtime System."

If a deferred file system initialization fails, the file status is set to 9B (32 for IBM DOS/VS) to indicate that the requested operation is not supported.

Note: Starting the runtime with the "-v" option forces startup initialization of all enabled file systems, as well as output of version information. For a complete description of the "-v" option, see section 2.2 in Book 1, *ACUCOBOL-GT User's Guide*.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X