LIANT

CodeBridge™

Calling Non-COBOL Subprograms

Version 8.0 for UNIX® and Windows®

This manual is a reference guide for Liant Software Corporation's CodeBridge, a cross-language call system designed to simplify communication between RM/COBOL programs and non-COBOL subprogram libraries written in C (or C++). It is assumed that the reader is familiar with programming concepts and with the COBOL and C (or C++) languages in general.

The information contained herein applies to systems running under Microsoft 32-bit Windows and UNIX-based operating systems.

The information in this document is subject to change without prior notice. Liant Software Corporation assumes no responsibility for any errors that may appear in this document. Liant reserves the right to make improvements and/or changes in the products and programs described in this guide at any time without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted.

Document Number 401215-0303

Table of Contents

List of Tables

Preface

Welcome to CodeBridge Version 8.0

This document describes CodeBridge, Liant Software Corporation's cross-language call system that is designed to simplify communication between RM/COBOL programs and non-COBOL subprogram libraries that are written in C.

CodeBridge for Windows and UNIX allows RM/COBOL programs to call non-COBOL subprograms built from external Application Programming Interfaces (APIs) or custom-developed C libraries without introducing "foreign" language data dependencies into either the COBOL program or the called C functions. This means that developers can write COBOL-callable C functions using C data types as usual, without worrying about the complexities of COBOL calling conventions or data types.

CodeBridge runs on Microsoft Windows 32-bit and UNIX-based operating systems.

What's New

CodeBridge version 8.0 includes several defect corrections, and the product complies with the RM/COBOL 8.0 release level.

Note For information on the significant enhancements in previous releases of CodeBridge, see the information beginning on page xviii.

Who Should Use CodeBridge

CodeBridge is intended for the following audiences:

1. Developers who may or may not be proficient in the C programming language and who wish to call existing C function libraries or system APIs without writing any additional C code.
2. Developers who are proficient in C programming and who wish to write new C function libraries that may be called from RM/COBOL version 7 (or later).
3. Developers who have previously written non-COBOL subprogram libraries in the form of Windows DLLs that are callable from RM/COBOL and who wish to take advantage of data conversion and validation features that are available in CodeBridge.

Organization of Information

The following lists the topics that you will find in the CodeBridge manual and provides a brief description of each.

Chapter 1—Introduction. This chapter provides a general overview of the CodeBridge cross-language call system, including components, benefits, requirements, information on how use this manual, and a typical development process with a basic, illustrative example. More examples are provided in Appendix B, *CodeBridge Examples*.

Chapter 2—Concepts. This chapter describes the concepts that are central to an understanding of CodeBridge, including using the template file components, passing information to a C function, returning C error values, associating C parameters with COBOL arguments, working with a variable number of C parameters, modifying COBOL data areas, using P-scaling, working with arrays, and using the CodeBridge Builder.

Appendix A—CodeBridge Errors. This appendix lists and describes the messages that can be generated during the use of either the CodeBridge Builder or the CodeBridge Library. These messages also include the CodeBridge Builder exit codes.

Appendix B—CodeBridge Examples. This appendix contains additional examples that use the typical CodeBridge development process outlined in Chapter 1, *Introduction*. The examples build from simple to complex, as a means of introducing CodeBridge concepts.

Appendix C—Useful C Information. The explanations in this appendix are intended to introduce basic C concepts to developers who are inexperienced in C. This information is intended to serve as a starting point for those developers who may not be proficient with C programming and who wish to call existing C function libraries without writing any additional C code.

Appendix D—Global Attributes. This appendix provides detailed descriptions of the global attributes used in a template file. See Chapter 2, *Concepts*, for more information about the basic components of a template file.

Appendix E—Parameter Attributes. This appendix provides detailed descriptions of the parameter attributes used in a template file. See Chapter 2, *Concepts*, for more information about the basic components of a template file.

Appendix F—CodeBridge Library Functions. This appendix describes each function in the CodeBridge Library. These descriptions will help you understand the C code generated by the CodeBridge Builder and will assist you in debugging applications developed using CodeBridge. Information on specifying the *Flags* parameter is also covered.

Appendix G—Non-COBOL Subprogram Internals for Windows. This appendix describes the internal details of how a non-COBOL subprogram is called from an RM/COBOL program running under Microsoft 32-bit Windows. It also provides information on preparing a non-COBOL subprogram for use by an RM/COBOL program on 32-bit Windows.

Appendix H—Non-COBOL Subprogram Internals for UNIX. This appendix describes the internal details of how a non-COBOL subprogram is called from an RM/COBOL program running under UNIX. It also provides information on preparing a non-COBOL subprogram for use by an RM/COBOL program on UNIX.

Appendix I—Calling CodeBridge Library Directly. This appendix includes guidelines for calling the CodeBridge Library directly rather than having the CodeBridge Builder generate the interface code from a template file. In order to call the CodeBridge Library directly, you must use an alternate method for preparing non-COBOL subprograms, as described in Appendices G and H.

The *CodeBridge* manual also includes an index.

Related Publications

For additional information, refer to the following publications:

RM/COBOL User's Guide

RM/COBOL Language Reference Manual

RM/COBOL Syntax Summary

Symbols and Conventions

The following typographic conventions are used throughout this manual to help you understand the text material and to define syntax:

1. Words in all capital letters indicate COBOL reserved words, such as statements, phrases, and clauses; acronyms; configuration keywords; environment variables, and RM/COBOL Compiler and Runtime Command line options.
2. Text that is displayed in a monospaced font indicates user input or system output (according to context as it appears on the screen). This type style is also used for sample command lines, program code and file listing examples, and sample sessions.
3. Bold, lowercase letters represent filenames, directory names, programs, C language keywords, and CodeBridge attributes.

   Words you are instructed to type appear in bold. Bold type style is also used for emphasis, generally in some types of lists.
4. Italic type identifies the titles of other books and names of chapters in this guide, and it is also used occasionally for emphasis.

   In COBOL syntax, italic text denotes a placeholder or variable for information you supply, as described below.
5. The symbols found in the COBOL syntax charts are used as follows:
   a. *italicized words* indicate items for which you substitute a specific value.
   b. UPPERCASE WORDS indicate items that you enter exactly as shown (although not necessarily in uppercase).
   c. ... indicates indefinite repetition of the last item.
   d. | separates alternatives (an either/or choice).

e. [ ] enclose optional items or parameters.

f. { } enclose a set of alternatives, one of which is required.

g. {| |} surround a set of unique alternatives, one or more of which is required, but each alternative may be specified only once; when multiple alternatives are specified, they may be specified in any order.

6. All punctuation must appear exactly as shown.

7. Key combinations are connected by a plus sign (+), for example, Ctrl+X. This notation indicates that you press and hold down the first key while you press the second key. For example, "press Ctrl+X" means to press and hold down the Ctrl key while pressing the X key. Then release both keys.

8. The term "Windows" in this document refers to 32-bit Microsoft Windows operating systems, including Windows 95, Windows 98, Windows Me, Windows NT 4.0, Windows 2000, or Windows XP, unless specifically stated otherwise. As you read through this guide, note that Liant may use two shorthand notations when referring to these operating systems. The term "Windows 9x class" refers to the Windows 95, Windows 98, or Windows Me operating system. The term "Windows NT class" refers to the Windows NT 4.0, Windows 2000, or Windows XP operating system.

9. RM/COBOL Compile and Runtime Command line options may be preceded by a hyphen. If any option is preceded by a hyphen, then a leading hyphen must precede all options. When assigning a value to an option, the equal sign is optional if leading hyphens are used.

10. In the electronic PDF file, this symbol represents a "note" that allows you to view last-minute comments about a specific topic on the page in which it occurs. This same information is also contained in the README text file under the section, Documentation Changes. In Adobe Reader, you can open comments and review their contents, although you cannot edit the comments. Notes do not print directly from the comment that they annotate. You may, however, copy and paste the comment text into another application, such as Microsoft Word, if you wish.

    To review notes, do one of the following:

    - To view a note, position the mouse over the note icon until the note description pops up.
    - To open a note, double-click the note icon.
    - To close a note, click the Close box in the upper-left corner of the note window.

Registration

Please take a moment to fill out and mail (or fax) the registration card you received with RM/COBOL. You can also complete this process by registering your Liant product online at: **http://www.liant.com**.

Registering your product entitles you to the following benefits:

- **Customer support.** Free 30-day telephone support, including direct access to support personnel and 24-hour message service.
- **Special upgrades.** Free media updates and upgrades within 60 days of purchase.
- **Product information.** Notification of upgrades, revisions, and enhancements as soon as they are released, as well as news about other product developments.

You can also receive up-to-date information about Liant and all its products via our web site. Check back often for updated content.

Technical Support

Liant Software Corporation is dedicated to helping you achieve the highest possible performance from the RM/COBOL family of products. The technical support staff is committed to providing you prompt and professional service when you have problems or questions about your Liant products.

These technical support services are subject to Liant's prices, terms, and conditions in place at the time the service is requested.

While it is not possible to maintain and support specific releases of all software indefinitely, we offer priority support for the most current release of each product. For customers who elect not to upgrade to the most current release of the products, support is provided on a limited basis, as time and resources allow.

Support Guidelines

When you need assistance, you can expedite your call by having the following information available for the technical support representative:

1. Company name and contact information.
2. Liant product serial number (found on the media label, registration card, or product banner message).
3. Product version number.
4. Operating system and version number.
5. Hardware, related equipment, and terminal type.
6. Exact message appearing on screen.
7. Concise explanation of the problem and process involved when the problem occurred.

Test Cases

You may be asked for an example (test case) that demonstrates the problem. Please remember the following guidelines when submitting a test case:

- The smaller the test case is, the faster we will be able to isolate the cause of the problem.
- Do not send full applications.
- Reduce the test case to one or two programs and as few data files as possible.
- If you have very large data files, write a small program to read in your current data files and to create new data files with as few records as necessary to reproduce the problem.
- Test the test case before sending it to us to ensure that you have included all the necessary components to recompile and run the test case. You may need to include an RM/COBOL configuration file.

When submitting your test case, please include the following items:

1. **README text file that explains the problems.** This file must include information regarding the hardware, operating system, and versions of all relevant software (including the operating system and all Liant products). It must also include step-by-step instructions to reproduce the behavior.

2. **Program source files.** We require source for any program that is called during the course of the test case. Be sure to include any copy files necessary for recompilation.

3. **Data files required by the programs.** These files should be as small as possible to reproduce the problem described in the test case.

Enhancements

The following sections summarize the major enhancements available in earlier versions of CodeBridge.

Version 7.5

Version 7.5 of CodeBridge, Liant Software's cross-language call system, has been enhanced to handle 64-bit integers on most UNIX platforms, providing the C compiler on the platform supports 64-bit integers.

A new runtime callback, GetCallerInfo, has been added to the CodeBridge Library. This function allows CodeBridge non-COBOL subprograms to obtain information about the calling COBOL program. Such information is particularly useful in error messages because it helps identify the offending CALL statement. See Appendix F, *CodeBridge Library Functions*, for more information.

Two new parameter attributes, called error base attributes, have been added to CodeBridge for retrieving error information set by C library and Windows API functions. The [[**errno**]] attribute supports obtaining the value of the external variable errno that was set by a call to a C library function. The [[**get_last_error**]] attribute supports obtaining the value returned by the Windows API function GetLastError called immediately after another Windows API function has been called. Prior to version 7.5, such error information was not available to the COBOL program because the runtime system uses C library and Windows API functions during the process of returning from the CodeBridge called C function to the COBOL program. Editing of generated code is undesirable and requires advanced knowledge of the C language. The new error base attributes in version 7.5 allow return of the error information by editing the CodeBridge template instead of the generated code. For additional information on error attributes, see "Returning C Error Values" in Chapter 2, *Concepts*, and "Error Base Attributes" in Appendix E, *Parameter Attributes*, of this manual.

Version 7.1

New to CodeBridge version 7.1 is support for UNIX. CodeBridge, Liant Software's cross-language call system, is in the RM/COBOL version 7.1 system. The CodeBridge Builder uses a template file to produce a C source file. The C source file provides the COBOL/C interface that may be used in an optional support module callable from COBOL programs.

The CodeBridge Builder generates C source modules that are platform-independent. Thus, you can use the CodeBridge Builder on a Windows platform to generate C source files that may be used on either a Windows or UNIX system.

Version 7.0 of the CodeBridge Builder produced C source code if the template file contained errors. Version 7.1 will not unless the -f (force) option is specified.

Version 7.0

The initial release of CodeBridge, version 7.0 for Windows, allows RM/COBOL programs to call non-COBOL subprograms built from external Application Programming Interfaces (APIs) or custom-developed C libraries without introducing "foreign" language data dependencies into either the COBOL program or the called C functions. This means that developers can write COBOL-callable C functions using C data types as usual, without worrying about the complexities of COBOL calling conventions or data types.

Chapter 1: Introduction

This introductory chapter provides an overview of CodeBridge and describes the following topics:

- CodeBridge technology and its components
- Benefits of using CodeBridge
- Requirements for developing applications using CodeBridge
- Information on how to use this manual
- An overview of a typical development process and example

What is CodeBridge?

CodeBridge allows RM/COBOL applications to call C functions without being concerned about the conversion between COBOL arguments and C parameters.

CodeBridge version 7.5 or later allows RM/COBOL programs to call non-COBOL subprograms built from external Application Programming Interfaces (APIs) or custom-developed C libraries without introducing "foreign" language data dependencies into either the COBOL program or the called C functions. This means that developers can write COBOL-callable C functions using C data types as usual, without worrying about the complexities of COBOL calling conventions or data types.

The developer augments C function prototypes with global and parameter attributes described in this manual to produce a template file. The developer uses the CodeBridge Builder utility to generate a C source file from the template file. This generated C source file contains the interface logic that, with the help from the CodeBridge Library, connects the calling COBOL program to the C function. The developer compiles this C source file, along with the C functions to be called, and links the generated object files together to form the completed non-COBOL subprogram library. In many cases, existing C library functions may be used to generate a non-COBOL subprogram library without writing any C code.

Note For Windows platforms, the generated non-COBOL subprogram library is a 32-bit dynamic link library (DLL). For UNIX platforms, the generated non-COBOL subprogram library is a "shared object" (normally referred to as an optional support module).

CodeBridge Components

CodeBridge consists of two main components:

- **CodeBridge Builder.** CodeBridge Builder is a standalone program that functions like a pre-compiler by reading a template file to generate a C source code file. The template file consists of C function prototypes that have been augmented with descriptive information. The output of the CodeBridge Builder is compiled and linked with the C functions to produce a non-COBOL subprogram library. The CodeBridge Builder is included in the RM/COBOL version 7 (or later) development system.
- **CodeBridge Library.** CodeBridge Library is a set of functions that performs conversion operations from COBOL arguments to C parameters and back again. The CodeBridge Library also contains functions to validate data and enforce interface constraints. The CodeBridge Library is part of the RM/COBOL version 7 (or later) runtime system.

Benefits of Using CodeBridge

CodeBridge provides the following benefits:

- Converts between COBOL and C data formats, eliminating the need for either the COBOL program or the C function having to deal with “foreign” language-dependent data types.
- Allows existing C libraries and standard APIs (such as the WIN32 API) to be used, in many cases, without writing any additional C code.
- Supports basic COBOL data types, including numeric, non-numeric, and pointer data items.
- Supports basic C data types, including integer and floating-point data items, numeric ASCII-encoded strings, and standard null-terminated C strings.
- Provides access to elements of COBOL data descriptors, which describe the properties of COBOL arguments.
- Provides C functions with the COBOL argument count, the COBOL initial state flag, and the Windows handle of the calling program.
- Provides data range and integrity checks for COBOL arguments and C parameters.

- Provides support for omitted arguments and null-valued pointer arguments.
- Provides limited support for calling C functions that allow a variable number of parameters.

Requirements

In order to develop applications using CodeBridge, you must have the following:

1. An RM/COBOL version 7 (or later) development system to develop applications using CodeBridge.
2. RM/COBOL version 7 (or later) runtime systems for deployment of applications based on CodeBridge technology.
3. A contemporary C development system:
   - For Windows, it must be capable of generating 32-bit dynamic link libraries (DLLs). Liant Software selected Microsoft's Visual C++ compiler for the development of the Windows version of CodeBridge. The Windows examples used in this manual are based on Microsoft command line syntax.
   - For UNIX, the C development system must be capable of generating shared objects. The command line syntax for the UNIX examples used in this manual is typical of many C compilers on UNIX. A makefile is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object to be used as a support module with the RM/COBOL runtime system. For additional information, see "Preparing C Subprograms" on page H-10.
4. Some knowledge of C programming. The skill level varies depending on what the developer wishes to accomplish. For those developers who are not proficient in C programming and who wish to call existing C function libraries, only a cursory knowledge of C is required. Appendix C, *Useful C Information*, contains brief explanations of some C language concepts and terminology, and may be useful for those developers who are not proficient in C.

Using this Manual

Depending on your experience level and how you to plan to use CodeBridge, this section contains information to help you learn to use CodeBridge effectively and quickly.

Developers Who are New to C Programming

A limited understanding of the C programming language is required to use CodeBridge effectively. If you are unfamiliar with the C programming language, you will want to refer first to Appendix C, *Useful C Information*. The explanations in this appendix are intended to introduce basic C concepts to developers who are inexperienced in C. More in-depth information can be found in the many resources published about programming in C. Appendix C also contains information on compiling and linking C functions.

Developers Who are Evaluating CodeBridge

It is recommended that all CodeBridge developers read and study Chapter 1, *Introduction*. This chapter presents the main features of CodeBridge, and acquaints you with an overview and general appearance of a typical CodeBridge program.

Another good way to become familiar with CodeBridge is to look at the examples in Appendix B, *CodeBridge Examples*. This appendix contains examples that introduce and illustrate several CodeBridge concepts and features. These examples may be helpful in generating CodeBridge template files that are based on existing C function prototypes.

In addition to these examples, several CodeBridge sample programs are included with the development system in the CodeBridge samples subdirectory. Within the **cbridge** subdirectory on Windows, the file **sample.txt** discusses the sample programs, including the **.bat** files to compile and run them, the **.tpl** and **.cbl** files, and the output they produce. These sample programs include a template file that contains definitions for a rich subset of the SQL function calls defined by Microsoft's ODBC API reference. The **README.txt** file in the **cbsample** subdirectory on UNIX discusses the CodeBridge sample programs that are included and how to run them.

Developers Who Wish to Use Existing C Libraries or Write New Non-COBOL Subprograms

For background information, you may wish to refer to the chapters and appendixes recommended for developers who are inexperienced in C programming and those who are evaluating CodeBridge for background information.

Then, study Chapter 2, *Concepts*, which focuses on the fundamentals and structure of CodeBridge.

Appendix D, *Global Attributes*, and Appendix E, *Parameter Attributes*, serve as reference guides to the attributes and attribute lists that are used in template files while developing CodeBridge applications.

Developers Who Have Written Non-COBOL Subprograms for Previous Versions of RM/COBOL

For background information, please refer to the previously recommended topics for developers who wish to use existing C libraries or who want to write new non-COBOL subprograms.

Next, read Appendix F, *CodeBridge Library Functions*, and Appendix I, *Calling the CodeBridge Library Directly*. Please note that the information in these two appendixes is not intended for a general audience. Rather, it is targeted to those developers who have previously written non-COBOL subprogram libraries in the form of Windows DLLs that are callable from RM/COBOL, and who wish to take advantage of the data conversion and validation features available in CodeBridge.

Finally, review either Appendix G, *Non-COBOL Subprogram Internals for Windows*, or Appendix H, *Non-COBOL Subprogram Internals for UNIX*. These appendices document the interface between the RM/COBOL runtime system and a C subprogram.

Developers Who Need Assistance in Testing and Debugging

Developers in this category may refer to Appendix A, *CodeBridge Errors*, which lists the error messages produced by the CodeBridge Builder and CodeBridge Library.

The information in Appendix F, *CodeBridge Library Functions*, would also prove useful to developers who are debugging applications developed using CodeBridge.

Typical Development Process Overview

Note In order to avoid confusion, the term “argument” is used when referring to COBOL data items; the term “parameter” is used when referring to C data items.

A typical CodeBridge development process would include the following steps:

1. **Selecting the C functions.** The first step is to select the C functions that are to be called from COBOL.

   These C functions may be ones that you have written or that you have acquired from a software vendor, or received as part of the standard C library that came with your C compiler, or obtained as part of a standard API for your operating system or one of its add-on components. Regardless of the source of these C functions, there will be one or more header files that contain descriptions of the functions (using C function prototypes), and, possibly, definitions of new data types and constants (using macros defined with **#define** C preprocessor directives and data types defined with C **typedef** statements). The information from these header files will be augmented with additional information as described in step 2.

2. **Creating the template file.** The next step is to create a template file that describes the relationship between the COBOL arguments and the C parameters.

   The template file (described in Chapter 2, *Concepts*) contains modified C function prototypes, where the modifications provide additional information describing each C parameter and the function return value. Each block of descriptive information is called an attribute list. Each attribute list contains one or more attributes. There are two kinds of attribute lists: parameter and global. Attributes and attribute lists are described in Appendix D, *Global Attributes*, and Appendix E, *Parameter Attributes*.

   Template files are generally free format in the sense that a line break may be placed wherever a blank may be placed. A template file line should not exceed 255 characters in length.

   Note C-style comments (/* comment */) may be included in the template source file. If comments are included, they are accepted by the CodeBridge Builder, but are not placed in the C source created from the template file.

   In addition to the annotated C function prototypes, it is necessary to add **#include** C preprocessor directives to the template file so that the C code generated by CodeBridge Builder can correctly resolve C data types. For example, if you are using the standard Windows API function, **MessageBox**, you must include the header file, **windows.h**. (“Example 2: Calling a Windows API Function” on page B-2 in Appendix B, *CodeBridge Examples*, demonstrates this requirement.) If you did not write the C functions, documentation that came with the software, your

C compiler, or an SDK (Software Development Kit), should provide this information.

3. **Invoking the CodeBridge Builder.** The CodeBridge Builder program uses the template file to generate C source code that contains the interface calls to connect the calling COBOL program to the C functions, and to convert COBOL arguments to and from C parameters.

   CodeBridge Builder is normally executed from a command line or script environment. It has two command line options: a required input parameter (the name of the template file) followed by an optional output parameter (the name of the generated C source file).

   Template files typically have a **.tpl** extension. If the optional output filename is not specified, the output is written to a file with the same name as the input file with the extension changed to **.c**.

   Any errors that occur are written to a file with the same name as the output file, but with the extension changed to **.err**. Errors encountered by the CodeBridge Builder should be fixed before continuing. Although the CodeBridge source code is generated when there are errors, it should not be considered valid.

   For more information, see "CodeBridge Builder" in Chapter 2, *Concepts*, and "CodeBridge Builder Error Messages" in Appendix A, *CodeBridge Errors*.

4. **Building the non-COBOL subprogram library.** CodeBridge Builder generates a C source program that must be compiled. Once the generated source has been compiled, it must be linked with the object code for the functions you wish to call from COBOL and with any libraries required by those functions or by the operating system. This linking process will produce a non-COBOL subprogram library that your COBOL program will use. Various compilers can be used to build the non-COBOL subprogram library, including Microsoft's Visual C++.

   Note 1 When calling existing object libraries other than the standard C library, you must specify the libraries needed in the link command.

   Note 2 When calling an existing Windows DLL, you must supply either a definition file (**.def**) or an import library file in the link command.

5. **Modifying or creating a COBOL program.** The next step is to modify an existing COBOL program or create a new one that calls the C functions you have selected.

   The USING phrase of the RM/COBOL CALL statement allows you to specify arguments you wish to pass to the C function. The GIVING (RETURNING) phrase of the RM/COBOL CALL statement allows you to specify an argument that would normally receive the return value of the C function.

CodeBridge is designed to give maximum flexibility in choosing COBOL data types to be converted to and from the C data types required by the C function. See Chapter 2, *Concepts*, for more information.

CodeBridge also allows wide latitude in mapping C function parameters to COBOL arguments. For more information, see "Associating C Parameters with COBOL Arguments" on page 2-21.

6. **Compiling the COBOL program.** Use the RM/COBOL compiler to compile your COBOL program.

7. **Running the application.** Execute the COBOL program, specifying the name of the non-COBOL subprogram library using the L Option of the RM/COBOL Runtime Command (**runcobol**). Alternatively, you may use the Command Line Options Registry property on Windows or the command line options in the UNIX resource file to specify the name of the non-COBOL subprogram library. (For more details, see "Setting Miscellaneous Properties" in Chapter 3, *Installation and System Considerations for Microsoft Windows*, and the "UNIX Resource File" section in Chapter 2, *Installation and System Considerations for UNIX*, in the *RM/COBOL User's Guide*). You may specify the name of the non-COBOL subprogram with the appropriate file extension. See page 1-11 for an example.

   Note On UNIX, there is an option to automatically load your subprogram library without the need to specify the L Option on the Runtime Command. Once your subprogram library is tested to your satisfaction, you may copy the **.so** (support module) to the **rmcobolso** subdirectory of the runtime execution directory (normally, **/usr/bin**). For additional information, see "Preparing C Subprograms" on page H-10. For a general discussion of support modules and how RM/COBOL uses them, see Appendix D, *Support Modules (Non-COBOL Add-Ons)*, in the *RM/COBOL User's Guide*.

Typical Development Process Example

The following example uses the typical development process outlined in the previous section. More examples can be found in Appendix B, *CodeBridge Examples*, and in the CodeBridge samples subdirectory (**cbridge** on Windows and **cbsample** on UNIX).

Example 1: Calling a Standard C Library Function

This example demonstrates calling a standard C library function without writing any C code. Parameter attribute lists are also presented.

1. Start with the function prototype for the standard C library cosine function, **cos**:

```
double cos(double x);
```

2. Create a template file called **trig.tpl** in the **src** directory that consists of the following lines:

```
#include <math.h>

[[float out rounded]] double cos(
[[float in]]                    double x);
```

   The **#include** C preprocessor directive is added to the template file so that the generated C source code can correctly resolve C data types. Because the cosine function is defined in the header file **math.h**, you should include this file in the template.

   Parameter attribute lists (for example, `[[float out rounded]]`) are constructed by placing the attributes between sets of double brackets. The parameter attribute lists are placed just before C data type references (in this example, **double**).

   A parameter attribute list must contain a base attribute (in this case, **float**, for floating-point). A parameter attribute list may contain a direction attribute (either **in** or **out**, or both), although a direction attribute is not always required. Optionally, a parameter attribute list may contain base modifier attributes (in this case, **rounded**, to indicate that COBOL rounding rules are to be applied).

   Note Unlike COBOL, C is a case-sensitive programming language. Thus, the case is significant for words in this example template file.

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\trig.tpl
```

   This command reads the input file from **src\trig.tpl** and writes its output file to **src\trig.c**. Any errors would be written to the file **src\trig.err**.

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice, using commands similar to the following:

   For Windows

```
cl –c –MD –Zp1 src\trig.c

link –nologo –machine:IX86 –section:.edata,RD –dll
     -subsystem:windows –out:trig.dll
     trig.obj kernel32.lib user32.lib
```

   For UNIX

   A makefile is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object to be used as a support module with the RM/COBOL runtime system. For additional information, see "Preparing C Subprograms" on page H-10.

   To compile:

```
cc -c src/trig.c
```

   Note Some compilers may require that the ELF (Executable and Linking Format) be specified, as follows:

```
cc -b elf -c src/trig.c
```

   To link:

```
cc -G -o trig.so trig.o
```

   Note Some linkers may require that you explicitly specify the math (or other) libraries, as follows:

```
cc -G -o trig.so trig.o -lm
```

5. Create a COBOL program in a file called **trig.cbl** that contains the following source fragments:

```
77 X-DEGREES PIC S999V99.
77 X-RADIANS PIC  S99V9(16).
77 RESULT    PIC  S99V9(06).
78 PI        Value 3.14159265359.

COMPUTE X-RADIANS = X-DEGREES / 180 * PI.
CALL "cos" USING X-RADIANS GIVING RESULT.
```

   Note Either numeric edited or any COBOL numeric usage may be specified in the data descriptions for X-DEGREES, X-RADIANS, and RESULT.

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol trig
```

7. Run the application, specifying the name of the COBOL program and the name of the non-COBOL subprogram library.

   You may specify the name of the non-COBOL subprogram with the appropriate file extension. The following two commands illustrate how to specify a Windows DLL or a UNIX shared object (generally known as support modules). Since the COBOL program and the non-COBOL subprogram have the same root name (trig), it is necessary to specify the correct file extension.

   For Windows

```
runcobol trig -l trig.dll
```

   For UNIX

```
runcobol trig -1 trig.so
```

   If the preceding examples had used different root names for the COBOL program and the non-COBOL subprogram, it would not be necessary to specify the file extension. For example, if the COBOL program were named “myprog”, then the following command could be used for either Windows or UNIX:

```
runcobol myprog -l trig
```

   This example assumes that both the COBOL program and the non-COBOL subprogram are located in the current directory.

Chapter 2: Concepts

This chapter describes concepts that are central to an understanding of CodeBridge, including the following:

- Using template file components (attributes and attribute lists)
- Passing information to a C function (categories of information that can be passed include COBOL arguments, COBOL argument properties, and miscellaneous information), including passing null-valued pointer arguments and managing omitted arguments (see page 2-6)
- Returning C error values (see page 2-18)
- Associating C parameters with COBOL arguments (see page 2-21)
- Working with a variable number of C parameters (see page 2-28)
- Modifying COBOL data areas (see page 2-29)
- Using P-scaling (see page 2-32)
- Working with arrays (see page 2-33)
- Using the CodeBridge Builder (see page 2-37)

Using Template File Components

In order to use the CodeBridge Builder (see page 2-37), you must provide a template file that describes each C function to be called from COBOL. Attribute lists are used in the template file to supplement information from the C function prototypes. An attribute list is a collection of attributes. Detailed information about attributes is provided in Appendix D, *Global Attributes*, and Appendix E, *Parameter Attributes*.

Note 1 As you read through this manual, keep in mind that the term “parameter attribute” is a shorthand notation for an attribute that occurs in a parameter attribute list. Likewise, “global attribute” indicates that the attribute can be found in a global attribute list.

Note 2 C-style comments (/* comment */) may be included in the template source file. If comments are included, they are accepted by the CodeBridge Builder, but are not placed in the C source created from the template file.

Attributes

An attribute is a keyword, such as **integer**, or a keyword with an associated value in parentheses, such as **occurs(3)**. Attribute keywords are case-sensitive and must be entered as shown.

The associated value is a constant. The CodeBridge Builder does not detect errors in the construction of the associated value.

A collection of attributes is known as an attribute list.

Attribute Lists

Two kinds of attribute lists, parameter and global, are used in a template file.

A parameter attribute list (described in the next section) is formed by enclosing one or more attributes in double brackets. For example:

```
[[integer in occurs(3)]]
```

A global attribute list (see page 2-5) is formed by enclosing one or more attributes between the characters [# and #]. For example:

```
[# replace_type (VOID_PTR; void *) #]
```

Sample template files using parameter and global attribute lists can be found on pages 2-4 and 2-5.

Parameter Attribute Lists

A parameter attribute list is associated with a C parameter or function return value. Each parameter attribute list describes the following:

- How COBOL arguments are to be validated and converted into C parameters before the C function is called.
- How C parameters are to be validated and converted back to COBOL arguments when the C function returns.

Zero or more parameter attribute lists may immediately precede the type information for each C parameter or function return value.

Attribute lists for a parameter or function return value may be omitted if the parameter or function return value is to be ignored.

Within a parameter attribute list, the parameter attributes need not be presented in any particular order. For example, `[[integer in]]` is the same as `[[in integer]]`. When a parameter is used for both input and output, specify both the **in** and **out** direction attributes in either order.

The attributes in a parameter attribute list belong to one of the following categories:

- **Base.** Base attributes indicate the general classification of a parameter (numeric, string, string length, pointer, descriptor, or error). Each parameter attribute list must contain exactly one base attribute, except that the **alias**(*name*) base modifier attribute may be used by itself if the return value is to be ignored. Therefore, within this document, a parameter attribute list is sometimes identified by its base attribute. For example, the phrase "an **integer** attribute list" refers to an attribute list that contains the **integer** base attribute. (Base attributes are covered in more detail beginning on page E-3.)

- **Base Modifier.** Base modifier attributes perform several tasks, such as: parameter conversion, parameter validation, error handling, array processing, handling of a variable number of C parameters, overriding the default size of a parameter, or supplying default values for omitted arguments. (More information about the base modifier attributes begins on page E-3.)

- **Direction.** A direction attribute, **in** and/or **out**, is sometimes required so that CodeBridge knows whether to generate code to convert a COBOL argument to a C parameter before calling the C function and/or to convert a C parameter to a COBOL argument when returning to the COBOL program.

  The base attributes, **float**, **general_string**, **integer**, **numeric_string**, **pointer_base**, **pointer_offset**, **pointer_size**, and **string**, apply to both input parameters and output parameters, and, therefore, require that a direction attribute be specified.

  All other base attributes apply only to input parameters, and, therefore, assume the presence of the **in** direction attribute. These base attributes do not allow the **in** direction attribute to be specified. (For more details about direction attributes, see page E-2.)

- **Argument Number.** CodeBridge provides a default automatic method of associating the C parameters and function return value from the C function prototype with COBOL arguments from the USING phrase and GIVING (RETURNING) phrase of the CALL statement. This default automatic association method is able to handle most cases. Note that for the more than 60 functions described in the file **sql.tpl** in the **cbridge** subdirectory (Windows only), none required using argument number attributes.

There are, however, situations that the default automatic association method will not handle (see "Example 4: Accessing COBOL Pointer Arguments" in Appendix B, *CodeBridge Examples*, and "Associating C Parameters with COBOL Arguments" on page 2-21). For these cases, use the explicit association method by specifying argument number attributes, **arg_num** or **ret_val**, to override the automatic association method. (For more information on the argument number attributes, see page E-2.)

For an alphabetized summary of the parameter attributes, see Table E-2 on page E-24.

Sample Template File Using Parameter Attribute Lists

The following C function prototype:

```
int MyFunction(char *Name, short NameSize);
```

may be modified by adding parameter attribute lists to produce the following template file:

```
[[integer out]]   int MyFunction(
[[string in]]                    char *Name,
[[buffer_length]]                short NameSize);
```

For each usage of a data item in the C function prototype (either for the function return value or for a parameter), a parameter attribute list has been added.

Since the C function returns an **int**, the **integer** base attribute and the **out** direction attribute are used.

For the Name parameter, the **string** base attribute and the **in** direction attribute are used to specify that the C function expects a string (array of **char**) as input.

The **buffer_length** base attribute is used to specify the size (in bytes) of the buffer used to contain the converted COBOL argument. By default, the **buffer_length** base attribute refers to the same COBOL argument that was used in the attribute list that preceded the **buffer_length** attribute list. Because the **buffer_length** base attribute may be used only with input parameters, it is neither necessary nor allowed to add the **in** direction attribute to the attribute list.

The COBOL program would call the C function with the following statement:

```
CALL "MyFunction" USING Name-1, GIVING Result-1.
```

Global Attribute Lists

A global attribute list provides information about one or more C function prototypes that is not specific to any given parameter. This information also could be used to modify the default behavior of CodeBridge Builder.

Global attribute settings take effect at the point the global attribute list occurs and are valid until another global attribute list alters these settings. A global attribute list is not associated with any particular function, argument, or parameter.

Sample Template File Using Global Attribute Lists

The following C function prototype:

```
SQLRETURN SQL_API SQLParamData(SQLHSTMT    StatementHandle,
                               SQLPOINTER *ValuePtrPtr);
```

may be modified by adding global and parameter attribute lists to produce the following template file:

```
#include "sqltypes.h"

  [# replace_type(SQLPOINTER; void *) #]
  [# convention(SQL_API) #]

[[integer out]] SQLRETURN SQL_API SQLParamData(
[[integer in]]                    SQLHSTMT    StatementHandle,
[[address]]                       SQLPOINTER *ValuePtrPtr);
```

The **replace_type** global attribute is used to expand the definition of SQLPOINTER to void *. The **convention** global attribute is used to identify function calling conventions.

Note 1 This example is based on the ODBC API, which is provided by Microsoft on Windows platforms. Other companies provide ODBC API implementations for some UNIX platforms.

Note 2 The header file, **sqltypes.h**, is included so that the C source code generated by CodeBridge will be able to resolve the data types, SQLRETURN and SQLHSTMT.

Passing Information to a C Function

CodeBridge is designed to simplify the process of calling C functions from COBOL programs. It is possible to call existing C library and standard API functions without writing additional C code. Even though no additional C code is required when using only existing C library or standard API functions, some knowledge of C programming is required in order to create the CodeBridge template file and to compile and link the CodeBridge non-COBOL subprogram library. Further knowledge of C programming is required if the developer desires to write new C programs or if intermediate functions must be written to pack scalars into structure or union parameters.

CodeBridge handles the conversion between COBOL and C data formats, which eliminates the need for either the COBOL program or the C function having to deal with "foreign" language-dependent data types. During the conversion process, CodeBridge can also perform data range and validity checks to verify that specified interface constraints are maintained.

CodeBridge allows three categories of information to be passed to the C function:

- COBOL arguments (see the following topic)
- COBOL argument properties (see page 2-15)
- Miscellaneous information (see page 2-17)

Furthermore, a COBOL program may omit an argument in the information passed to a C function, as discussed in "Managing Omitted Arguments" on page 2-17.

Passing COBOL Arguments

COBOL arguments may be numeric, non-numeric, or pointer data items. COBOL numeric arguments may be passed to C integer, floating-point, and numeric string parameters. COBOL non-numeric arguments must be passed to C string parameters. As a special case for C functions designed to interpret a null-valued pointer as an omitted parameter, a COBOL null-valued pointer argument may be passed in place of a numeric or non-numeric argument and the C function parameter will be set to a null-valued pointer. COBOL pointer data items contain three components: base address, offset, and size. The address component must be passed to C pointer parameters; the offset and size components must be passed to C numeric parameters.

Passing COBOL Numeric Arguments

CodeBridge supports all RM/COBOL numeric data types, including display, numeric edited, packed, unpacked, and binary. A COBOL numeric argument may be passed to one of three C parameter types: integer, floating-point, and string. When passed to a string, the numeric value is converted to and from a string representation. Therefore, in this document, this form is referred to as a numeric string.

Note While the COBOL language defines the numeric edited category as belonging to the alphanumeric class, CodeBridge treats numeric edited data items as numeric. It is currently an error to pass a numeric edited argument to a parameter described with the **string** base attribute. Instead, a numeric edited argument should be passed to a parameter described with either the **numeric_string** or **general_string** base attributes.

Numeric Arguments with C Integer Parameters

A C integer parameter is described in the template file using the **integer** base attribute. The **integer** base attribute may be used with any of the C integer data types, including **char**, **short**, **int**, and **long**, with or without the C signed type specifier keywords **signed** and **unsigned**. These data types can be used directly (such as "int *Name*"), indirectly ("int **pName*"), and with array declarations ("int *ArrayName*[]").

When used directly ("int *Name*"), the parameter is passed to the C function "by value". As such, it is unable to modify the value of the actual parameter. Passing a parameter "by value" usually means that it is an input parameter, which indicates that the **in** direction attribute should be specified in the attribute list for the parameter.

When used indirectly ("int **pName*"), the parameter is passed to the C function "by reference". This means that the C function is given a pointer to the parameter and, therefore, is able to modify the value of the actual parameter. Passing a parameter "by reference" usually means that it is an output (or input/output) parameter, which indicates that the **out** direction attribute (or both the **in** and **out** direction attributes) should be specified in the attribute list for the parameter.

As a special case for C integer parameters that are passed indirectly, CodeBridge will pass the C null pointer to the C function when the COBOL argument is a null-valued COBOL pointer. For more information, see "Passing Null-Valued Pointer Arguments" on page 2-13.

When used as an array ("int *ArrayName*[]"), the address of the array is passed to the C function. For more information, see "Working with Arrays" on page 2-33.

The conversion process for C integer parameters may be modified by using the following base modifier attributes: **no_size_error**, **occurs**(*value*), **repeat**(*value*), **rounded**, **scaled**(*value*), **silent**, **unsigned**, and **value_if_omitted**(*value*). For more information, see "Base Modifiers that Apply to Numeric Base Attributes" on page E-7.

Interface constraints for C integer parameters may be specified by using the following base modifier attributes: **assert_digits**(*min;max*), **assert_digits_left**(*min;max*), **assert_digits_right**(*min;max*), **assert_length**(*min;max*), **assert_signed**, **assert_unsigned**, **integer_only**, **no_null_pointer**, and **optional**. For more information, see "Base Modifiers that Apply to Numeric Base Attributes" on page E-7.

Numeric Arguments with C Floating-Point Parameters

A C floating-point parameter is described in the template file using the **float** base attribute. The **float** base attribute may be used with either of the C floating-point data types, **float** or **double**. These data types can be used directly (such as "float *Name*"), indirectly ("float **pName*"), and with array declarations ("float *ArrayName*[]").

When used directly ("float *Name*"), the parameter is passed to the C function "by value". As such, it is unable to modify the value of the actual parameter. Passing a parameter "by value" usually means that it is an input parameter, which indicates that the **in** direction attribute should be specified in the attribute list for the parameter.

When used indirectly ("float **pName*"), the parameter is passed to the C function "by reference". This means that the C function is given a pointer to the parameter and, therefore, is able to modify the value of the actual parameter. Passing a parameter "by reference" usually means that it is an output (or input/output) parameter, which indicates that the **out** direction attribute (or both the **in** and **out** direction attributes) should be specified in the attribute list for the parameter.

As a special case for C floating-point parameters that are passed indirectly, CodeBridge will pass the C null pointer to the C function when the COBOL argument is a null-valued COBOL pointer. For more information, see "Passing Null-Valued Pointer Arguments" on page 2-13.

When used as an array ("float *ArrayName*[]"), the address of the array is passed to the C function. For more information, see "Working with Arrays" on page 2-33.

The conversion process for C floating-point parameters may be modified by using the following base modifier attributes: **no_size_error**, **occurs**(*value*), **repeat**(*value*), **rounded**, **silent**, and **value_if_omitted**(*value*). For more information, see "Base Modifiers that Apply to Numeric Base Attributes" on page E-7.

Interface constraints for C floating-point parameters may be specified by using the following base modifier attributes: **assert_digits**(*min;max*), **assert_digits_left**(*min;max*), **assert_digits_right**(*min;max*), **assert_length**(*min;max*), **assert_signed**, **assert_unsigned**, **no_null_pointer**, and **optional**. For more information, see "Base Modifiers that Apply to Numeric Base Attributes" on page E-7.

Numeric Arguments with C Numeric String Parameters

A C numeric string parameter is described in the template file using either the **numeric_string** or the **general_string** base attributes. The **numeric_string** or **general_string** base attributes may be used with any of the C string data types: **char ***, **signed char ***, and **unsigned char ***.

Note 1 The C parameter declarations "char **String*" and "char *String*[]" are equivalent.

Note 2 C strings are one-dimensional arrays of characters. C always passes array parameters "by reference", which means that the address of the first character of the string is passed to the C function.

Although string parameters are always passed "by reference", this does not mean that a C string parameter is always an output parameter. Depending on its use in the C function, it may be an input parameter, an output parameter, or an input/output parameter. Its use indicates whether the **in** direction attribute (input), the **out** direction attribute (output), or both the **in** and **out** direction attributes (input/output) should be specified in the attribute list for the parameter.

As a special case for C numeric string parameters, CodeBridge will pass the C null pointer to the C function when the COBOL argument is a null-valued COBOL pointer. For more information, see "Passing Null-Valued Pointer Arguments" on page 2-13.

During the conversion process, CodeBridge dynamically allocates a buffer to hold the converted COBOL argument (for input conversions) or hold the C string generated by the C function (for output conversions). While processing string parameters, the C function may need to know the size of the string or the size of the string conversion buffer. CodeBridge provides three attributes for obtaining this string length information. The **length** base attribute provides the length of the COBOL argument. The **buffer_length** base attribute provides the size of the allocated string buffer. The **effective_length** base attribute provides the actual number of characters stored in the string buffer, not including the null character terminating the string.

When passing an array of C strings ("char **StringArray*[]"), the address of the first string pointer is passed to the C function. For more information, see "Working with Arrays" on page 2-33.

The conversion process for C numeric string parameters may be modified by using the following base modifier attributes: **leading_minus**, **leading_sign**, **no_size_error**, **occurs**(*value*), **repeat**(*value*), **rounded**, **silent**, **size**(*value*), **trailing_credit**, **trailing_debit**, **trailing_minus**, **trailing_sign**, and **value_if_omitted**(*value*). For more information, see "Base Modifiers that Apply to Numeric Base Attributes" on page E-7.

Interface constraints for C numeric string parameters may be specified by using the following base modifier attributes: **assert_digits**(*min;max*), **assert_digits_left**(*min;max*), **assert_digits_right**(*min;max*), **assert_length**(*min;max*), **assert_signed**, **assert_unsigned**, **no_null_pointer**, and **optional**. For more information, see "Base Modifiers that Apply to Numeric Base Attributes" on page E-7.

String base modifier attributes that are allowed when the **general_string** base attribute is specified are ignored for numeric arguments.

Passing COBOL Non-Numeric Arguments

CodeBridge supports all RM/COBOL non-numeric data types, including alphabetic and alphanumeric elementary items. CodeBridge also supports passing group items. A COBOL non-numeric argument must be passed to a C string parameter.

Note While the COBOL language defines the numeric edited category as belonging to the alphanumeric class, CodeBridge treats numeric edited data items as numeric. It is currently an error to pass a numeric edited argument to a parameter described with the **string** base attribute. Instead, a numeric edited argument should be passed to a parameter described with either the **numeric_string** or **general_string** base attributes.

Non-Numeric Arguments with C String Parameters

A C string parameter is described in the template file using either the **string** or the **general_string** base attributes. The **string** or **general_string** base attributes may be used with any of the C string data types: **char ***, **signed char ***, and **unsigned char ***.

Note 1 The C parameter declarations "char **String*" and "char *String*[]" are equivalent.

Note 2 C strings are one-dimensional arrays of characters. C always passes array parameters "by reference", which means that the address of the first character of the string is passed to the C function.

Although string parameters are always passed “by reference”, this does not mean that a C string parameter is always an output parameter. Depending on its use in the C function, it may be an input parameter, an output parameter, or an input/output parameter. Its use indicates whether the **in** direction attribute (input), the **out** direction attribute (output), or both the **in** and **out** direction attributes (input/output) should be specified in the attribute list for the parameter.

As a special case for C string parameters, CodeBridge will pass the C null pointer to the C function when the COBOL argument is a null-valued COBOL pointer. For more information, see “Passing Null-Valued Pointer Arguments” on page 2-13.

During the conversion process, CodeBridge dynamically allocates a buffer to hold the converted COBOL argument (for input conversions) or hold the C string generated by the C function (for output conversions). While processing string parameters, the C function may need to know the size of the string or the size of the conversion buffer. CodeBridge provides three attributes for obtaining this string length information. The **length** base attribute provides the length of the COBOL argument. The **buffer_length** base attribute provides the size of the allocated string buffer. The **effective_length** base attribute provides the actual number of characters stored in the string buffer, not including the null character terminating the string.

Note If a COBOL non-numeric argument contains a C null character (0x00), conversion of the argument to a C string parameter may produce unexpected results. The input conversion process ends when all characters have been copied or a C null character is encountered.

When passing an array of C strings (“char **StringArray*[]”), the address of the first string pointer is passed to the C function. For more information, see “Working with Arrays” on page 2-33.

The conversion process for C non-numeric string parameters may be modified by using the following base modifier attributes: **leading**(*value*), **leading_spaces**, **occurs**(*value*), **repeat**(*value*), **silent**, **size**(*value*), **trailing**(*value*), **trailing_spaces**, and **value_if_omitted**(*value*). For more information, see “Base Modifiers that Apply to the String Base Attribute” on page E-11.

Interface constraints for C non-numeric string parameters may be specified by using the following base modifier attributes: **assert_length**(*min;max*), **no_null_pointer**, and **optional**. For more information, see “Base Modifiers that Apply to the String Base Attribute” on page E-11.

Numeric string base modifier attributes that are allowed when the **general_string** base attribute is specified are ignored for non-numeric arguments.

Groups with C String Parameters

COBOL group items are hierarchical data structures that contain subordinate groups and elementary data items. CodeBridge does not provide support for accessing data items subordinate to a group.

A COBOL group is non-numeric but may contain numeric and pointer data. Because it is non-numeric, a group can be passed to a C string parameter. Since it may contain numeric and pointer data, the likelihood of unexpected results from encountering a C null character (0x00) is greater than when passing elementary non-numeric arguments.

An RM/COBOL variable-length group argument is always passed as a fixed-length group of the maximum size so that the called program has the opportunity to increase the variable size if desired. Thus, passing variable-length groups does not support passing variable-length strings to C.

Passing COBOL Pointer Arguments

The pointer data type is a new feature introduced in version 7.0 of RM/COBOL. A COBOL pointer describes a block of memory and consists of three components: base address, offset, and size.

CodeBridge provides two methods for passing COBOL pointers. The first method is useful when the C function only wishes to access memory referenced by the pointer. The second method is useful if the C function wishes to access the components of the COBOL pointer data item directly. For more information, see "Pointer Base Attributes" on page E-15.

Method 1: Passing Pointer Address and Pointer Length

With this method, you can pass the address or the length of the block of memory to an input parameter in the C function. Given the address and length of the memory to which the pointer refers, the C function may read or modify the contents of that memory block. It is the C programmer's responsibility to confine any such references to lie wholly within the memory block described by the given pointer values. However, the C function cannot change the base address, offset, or size of the COBOL pointer.

Use the **pointer_address** base attribute in the template file to describe a C pointer parameter and instruct CodeBridge to pass the effective address of the memory block (base address plus offset) to the C function as the parameter value.

Use the **pointer_length** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the effective length of the memory block (size minus offset) to the C function as the parameter value.

Method 2: Passing and Modifying Pointer Components

With this method, you can pass the base address, offset, or size of the block of memory to an input, output, or input/output parameter in the C function. Given the base address, offset, and size of the memory to which the pointer refers, the C function may read or modify the contents of that memory block. It is the C programmer's responsibility to confine any such references to lie wholly within the memory block described by the given pointer values. In addition, for output and input/output parameters, the C function can also modify the base address, offset, or size component values of the COBOL pointer.

Use the **pointer_base** base attribute in the template file to describe a C pointer parameter, instruct CodeBridge to pass the base address of the memory block to the C function for input conversions, and set the base address component of the COBOL pointer for output conversions. The output conversion process may be modified by using the following base modifier attributes: **pointer_max_size** and **pointer_reset_offset**. For more information, see "Base Modifiers that Apply to Pointer Base Attributes" on page E-16.

Use the **pointer_offset** base attribute in the template file to describe a C numeric parameter, instruct CodeBridge to pass the offset component of the COBOL pointer to the C function for input conversions, and set the offset component of the COBOL pointer for output conversions. The output conversion process may be modified by using the **pointer_max_size** base modifier attribute.

Use the **pointer_size** base attribute in the template file to describe a C numeric parameter, instruct CodeBridge to pass the size component of the COBOL pointer to the C function for input conversions, and set the size component of the COBOL pointer for output conversions. The output conversion process may be modified by using the **pointer_reset_offset** base modifier attribute.

Passing Null-Valued Pointer Arguments

Null-valued pointer arguments arise in one of three ways: the argument is the figurative constant NULL (NULLS), the argument is a COBOL pointer that has been set to NULL (NULLS), or the argument is a pointer that has been set from another null-valued pointer. Based on the properties of the C parameter associated with a pointer argument, CodeBridge handles pointer arguments as follows:

- **Numeric or non-numeric parameter (direct or indirect)**

  For related information, see "Passing COBOL Numeric Arguments" on page 2-7 and "Passing COBOL Non-Numeric Arguments" on page 2-10.

A COBOL program may pass a COBOL null-valued pointer data item as an argument that is associated with any of these base attributes: **float, general_string, integer, numeric_string**, or **string**. Associating a null-valued pointer with a parameter having one of these base attributes has meaning only when the C parameter is a pointer (indirect) parameter.

Some C functions are designed to interpret the occurrence of a null-valued pointer parameter to indicate that the parameter is omitted and that the function should not read or write indirectly through the parameter pointer value. If a COBOL program passes a COBOL null-valued pointer, the C function will receive a C null-valued pointer in order to support this design.

If the C parameter is not a pointer, it is meaningless to pass a COBOL null-valued pointer argument. For a direct numeric or non-numeric parameter, an uninitialized variable will be passed as the parameter value when a null-valued pointer argument is provided. The **no_null_pointer** base modifier attribute may be specified to cause CodeBridge to return an error if a COBOL null-valued pointer is passed to the parameter.

If a null-valued pointer argument is used for an output parameter that is numeric or non-numeric, the parameter result value is ignored as if the **out** direction attribute had not been specified.

A null-valued pointer argument may not be used for a numeric or non-numeric parameter that specifies the **no_null_pointer** base modifier attribute.

A pointer argument with a value other than null always causes an error for a numeric or non-numeric parameter. Since COBOL pointer data items are not typed (that is, they are essentially equivalent to (**void ***) in C), CodeBridge does not have enough information to dereference the COBOL pointer (that is, to convert the data that the pointer references).

- **Pointer parameter, where the C function needs a COBOL pointer value**

  For related information, see “Passing COBOL Pointer Arguments” on page 2-12.

  When a COBOL program passes a pointer argument associated with a parameter described with the **pointer_address** or **pointer_base** base attributes, the pointer value is passed to the C function as the parameter value, regardless of whether the pointer value is null or non-null.

  The **out** direction attribute may be specified with the **pointer_base** base attribute to modify the base address of the pointer argument upon return from the C function. It is an error to specify either of the **in** or **out** direction attributes with the **pointer_address** base attribute.

The **pointer_offset**, **pointer_size**, and **pointer_base** base attributes yield a zero for a null-valued pointer argument on input to the C function but allow the corresponding component of the pointer argument to be changed on output if the **out** direction attribute is specified and the base address of the pointer is also changed to a non-zero value.

Passing COBOL Argument Properties

CodeBridge supports two categories of COBOL argument properties, each of which may be passed to the C function:

- COBOL descriptor data (see the following topic)
- String length information (see page 2-16)

Passing COBOL Descriptor Data

Prior to CodeBridge, if a developer wanted information about the properties of the COBOL arguments, it was necessary for the C program to obtain the information for each argument from a structure known as the COBOL data descriptor. The COBOL data descriptor contains properties of the COBOL argument, including its address, length and type, digit count and scale factor (for numeric arguments), and encoded picture (for numeric edited and alphanumeric edited arguments). CodeBridge supports the passing of all these properties except for the encoded picture. (See either Appendix G, *Non-COBOL Subprogram Internals for Windows*, or Appendix H, *Non-COBOL Subprogram Internals for UNIX*, for more information about the earlier method of calling non-COBOL subprograms.)

In CodeBridge, the following descriptor base attributes (which are described in detail on page E-17) may be used to pass a component of the COBOL argument to the C function.

Use the **address** base attribute in the template file to describe a C pointer parameter and instruct CodeBridge to pass the address of the COBOL argument to the C function as the parameter value.

Note Passing the address of the COBOL argument data item to a C function as a parameter value should be a rare occurrence when using CodeBridge. Use of the data item address requires the C function to know the details of COBOL data formats and is not subject to the data validation and interface constraints that CodeBridge provides.

Use the **digits** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the digit count, that is, the number of 9's in the PICTURE character-string, of the COBOL numeric argument to the C function as the parameter value. For non-numeric arguments, the value is not defined.

Use the **length** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the length of the COBOL argument to the C function as the parameter value.

Use the **scale** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the scale factor of the COBOL numeric argument to the C function as the parameter value. For non-numeric arguments, the value is not defined. The value of the scale passed is the arithmetic complement of the value in the COBOL argument descriptor.

Use the **type** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the type of the COBOL argument to the C function as the parameter value.

See also the discussion of passing miscellaneous information to a C function on page 2-17.

Passing String Length Information

In addition to COBOL data descriptor components, CodeBridge can supply string length information for input conversions. The C function can be supplied the length of the COBOL argument (from the COBOL data descriptor), the length of the conversion buffer, or the effective length of the C string (after conversion).

Use the **length** base attribute (for more information, see "Descriptor Base Attributes" on page E-17) in the template file to describe a C numeric parameter and instruct CodeBridge to pass the length of the COBOL argument to the C function as the parameter value.

Use the **buffer_length** base attribute (for more information, see "String Length Base Attributes" on page E-14) in the template file to describe a C numeric parameter and instruct CodeBridge to pass the length of the conversion buffer to the C function as the value of the parameter. The length of the buffer is determined by the base attribute that is used to describe the string parameter associated with the same argument, as follows:

- For the **string** base attribute, the buffer length defaults to one more than the length of the passed COBOL argument, which allows space for the characters of the argument value and a null-termination character.
- For the **numeric_string** base attribute, the buffer length defaults to four more than the digit length of the passed COBOL argument, which allows space for the digits of the argument value and the sign, decimal-point, and null-termination characters.
- For the **general_string** base attribute, the buffer length defaults to the greater of one more than the length of the passed COBOL argument and four more than the digit

length of the passed COBOL argument, which allows space for either a non-numeric or numeric argument conversion.

The default values for **buffer_length** may be overridden by using the **size**(*value*) base modifier attribute (see page E-13) in the attribute list that contains the **string**, **numeric_string**, or **general_string** base attribute that is associated with the same argument as **buffer_length**.

Use the **effective_length** base attribute (see also page E-14) in the template file to describe a C numeric parameter and instruct CodeBridge to pass the actual number of characters stored in the conversion buffer, not including the null character that terminates the string (after the input conversion process is complete), to the C function as the parameter value.

Passing Miscellaneous Information

CodeBridge also can supply the number of COBOL arguments specified in the USING phrase of the CALL statement, the COBOL initial state flag, and the Windows handle for the COBOL program. For more information, see "Descriptor Base Attributes" on page E-17.

Use the **arg_count** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the number of COBOL arguments to the C function as the parameter value.

Use the **initial_state** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the COBOL initial state flag to the C function as the parameter value.

Use the **windows_handle** base attribute in the template file to describe a C numeric parameter and instruct CodeBridge to pass the Windows handle for the COBOL program to the C function as the parameter value.

Managing Omitted Arguments

A COBOL program may omit an argument by specifying fewer arguments in the USING phrase of the CALL statement than expected by the C function or by explicitly specifying the OMITTED keyword for an argument in the USING phrase of the CALL statement. The GIVING argument may be omitted by not specifying the GIVING (RETURNING) phrase in the CALL statement.

An omitted argument will cause an error if it is passed to a numeric or non-numeric parameter that does not also specify either the **optional** or **value_if_omitted** base modifier attributes. The descriptor base attributes are implicitly optional and return

default values for an omitted argument; the **optional** base modifier attribute is not allowed with the descriptor base attributes.

For an omitted argument passed to a parameter described with the **optional in** attributes, an appropriate default is passed to the C function as the parameter value. The default value associated with an **integer** or **float** base attribute is a numeric zero. The default value associated with a **general_string**, **numeric_string**, or **string** base attribute is an empty string (the first character of the string is a null character). If the **value_if_omitted**(*value*) base modifier attribute has been specified, *value* is passed instead of the default value.

An omitted argument is assumed to satisfy any of the assertion base modifier attributes. If a default value is provided with the **value_if_omitted**(*value*) base modifier attribute, it is the user's responsibility to provide a default value that satisfies all interface constraints.

For the descriptor base attributes, an omitted argument has the following results, regardless of whether the argument is missing from the USING phrase or explicitly specified as OMITTED:

- The **address** base attribute for an omitted argument supplies the value NULL.
- The **digits** base attribute for an omitted argument supplies zero.
- The **length** base attribute for an omitted argument supplies zero.
- The **scale** base attribute for an omitted argument supplies zero.
- The **type** base attribute for an omitted argument supplies the value RM_OMITTED, which has the value 32 as shown in Table E-1 on page E-19.

If an argument is omitted for a parameter described with the **optional out** attributes, the parameter result value is ignored. However, CodeBridge Builder does not currently allow this combination of attributes. That is, output arguments are required in the current implementation of CodeBridge.

Returning C Error Values

Two base attributes, called error base attributes, support returning C error values to the COBOL program. The **errno** error base attribute returns the value of the external variable errno, which is set by many C library functions. The **get_last_error** error base attribute returns the value returned by the Windows API function GetLastError. The error base attributes are necessary because the RM/COBOL runtime system uses C library functions, and, on Windows, Windows API functions, during the return to the COBOL program that modify the error values. Thus, any error values set by the

CodeBridge called C function are modified before the COBOL program has a chance to obtain them. The error base attributes solve this problem by causing the CodeBridge Builder to generate code to preserve the error value set by the C function specified in the CodeBridge template. The preserved value is returned in an associated COBOL argument for access by the calling COBOL program. Complete details regarding the error base attributes are found in the section "Error Base Attributes" on page E-20. In addition, some general concepts and examples of error base attributes are provided in the sections that follow.

Consistent Return Values

For those C library functions that set the external variable errno, it is considered correct behavior not to modify the value of errno if no error occurs. In other words, if no error is detected, the external variable errno will have the same value that it had before the C function was called. The code sequence that is generated by CodeBridge Builder guarantees the value of errno is zero just prior to the C function call. The generated code sequence is as follows:

```
errno = 0;
__RETURN__open = open(filename, oflag);
__save_errno = errno;
```

Similarly, for those Windows API functions that set a value to be returned by the function GetLastError, it is also considered correct behavior not to modify the last error value if no error occurs. In other words, if no error is detected by the C function, the call to GetLastError will return the same value it would have if it were called just prior to the C function. The code sequence that is generated by CodeBridge Builder guarantees that the value returned by GetLastError will be zero if no error is detected by the C function call. The generated code sequence is as follows:

```
SetLastError(0);
__RETURN__CreateDirectory = CreateDirectory(DirName,SecAttr);
__save_lastError = GetLastError();
```

Specifying Both errno and get_last_error

It is possible to use the error base attributes **errno** and **get_last_error** in the same function description. Functions that return an error code in the external variable errno have a function return value of -1. Functions that return an error through GetLastError have a function return value of FALSE (zero). On the surface, this seems meaningless (and in most cases, it probably is); however, there is no reason to disallow this behavior. It is possible for a Windows API function to call a C library function that could set a

value in the external variable errno. It may be of value to the COBOL program to interrogate both error conditions.

The generated code sequence when both attributes are specified is as follows:

```
SetLastError(0);
errno = 0;
__RETURN__CreateDirectory = CreateDirectory(DirName, SecAttr);
__save_errno = errno;
__save_lastError = GetLastError();
```

Function Return Value (Status) Versus Error Values

In many cases, the return value from a C library function or a Windows API function is merely a simple binary indication of success or failure.

C library functions that set the external variable errno generally return –1 as the function return value. If the return value is not –1, the value may or may not indicate anything of significance. For example, the C library function, **mkdir,** always returns 0 (for success) or –1 (for failure). On the other hand, the C library function, **open,** returns a file handle if the operation succeeded or –1 if the operation failed. Windows API functions normally return non-zero to indicate success and zero to indicate an error.

For those C library and Windows API functions where the return value is a simple indication of success or failure, it may be inefficient to have the COBOL program examine both the return value and the value of the argument associated with the **errno** or **get_last_error** attribute.

If you are certain that the C function return value is not needed—except to show success or failure—you need not access this parameter from COBOL. The following template illustrates how to obtain the _mkdir function return value and the value of the external variable errno:

```
[[integer out]]                  int _mkdir(
  [[string in trailing_spaces]]    const char *DirName
  [[errno]]);
```

This function could be called from COBOL with this statement:

```
CALL "_mkdir" USING File-Name Err-No
    GIVING Return-Status.
```

There is no real need to examine `Return-Status` in the COBOL program, since examining `Err-No` is sufficient (it is guaranteed that `Err-No` will be zero if no error occurred). You may alter the template so that `Err-No` becomes the return value with a template similar to the following:

```
                                    int _mkdir(
[[string in trailing_spaces arg_num(1)]]
                                      const char *DirName
[[errno ret_val]]);
```

The COBOL calling sequence could then be simplified as follows:

```
CALL "_mkdir" USING File-Name
     GIVING Err-No.
```

Besides making the COBOL calling sequence simpler, this technique also simplifies the C source code that is generated by CodeBridge Builder.

Associating C Parameters with COBOL Arguments

Using CodeBridge, a single C parameter or return value may be associated with multiple COBOL arguments by the use of more than one attribute list, but each attribute list associates a parameter with, at most, one argument from the COBOL CALL statement. Also, multiple C parameters may be associated with a single COBOL argument. That is, the CodeBridge association of C parameters with COBOL arguments allows a many-to-many relationship.

CodeBridge has two methods of associating C parameters with COBOL arguments: explicit association and automatic association. You can explicitly specify the association of a C parameter with a COBOL argument, or you can have CodeBridge automatically associate C parameters with COBOL arguments for you. If you do not use the explicit association method, CodeBridge will use the automatic association method by default. If the attribute list for any parameter of a function specifies explicit association of the C parameter to a COBOL argument, the attribute lists for all parameters for that function—except those attribute lists containing a base attribute that does not refer to an argument in the COBOL CALL statement—must specify explicit association. Different functions within a single template file may use different association methods.

Explicit Association

CodeBridge is designed to handle most C-parameter-to-COBOL-argument association situations without requiring you to explicitly specify the associations in the attribute lists of your template file. For those situations where the CodeBridge automatic association method does not produce the desired result, you must use the explicit association method. Even when the automatic association method produces the correct result, you may use the explicit association method. For instance, you might elect to use the explicit association method to clearly document the association of parameters with arguments.

To explicitly specify the association of the C function return value or a C parameter to a particular COBOL argument, you include either the **ret_val** or the **arg_num(***value*) argument number attribute in the attribute list for the return value or parameter (for more information, see "Argument Number Attributes" on page E-2). If you explicitly specify an argument number attribute in any attribute list for an individual C function, you must do so for every attribute list for that function—except for those attribute lists containing a base attribute that does not refer to an argument.

Automatic Association

The following material explains automatic association of C parameters with COBOL arguments. Each attribute list refers either to the C function return value or to a single C parameter.

Automatic Association of the C Function Return Value with a COBOL Argument

When there is no attribute list associated with the C function return value, the function return value is ignored.

If there is an attribute list for the C function return value, the return value is associated with the argument specified by the GIVING (RETURNING) phrase of the RM/COBOL CALL statement. In the automatic association method, if there are multiple attribute lists associated with the C function return value, they all associate the return value with the GIVING argument. If the return value is to be stored other than in the GIVING argument, the explicit association method must be used.

Note Only base attributes that allow the **out** direction attribute may be used in the attribute list associated with the function return value. These base attributes include **float**, **general_string**, **integer**, **numeric_string**, **pointer_base**, **pointer_offset**, **pointer_size**, and **string**.

Automatic Association of C Parameters with COBOL Arguments

When there is no attribute list associated with a C parameter, there is no associated COBOL argument. For such a parameter there are no input conversions, so the parameter is passed an uninitialized variable, and there are no output conversions, so the final value of the parameter is ignored.

If there are one or more attribute lists associated with a C parameter, CodeBridge uses the required base attribute of each attribute list to determine the association with a COBOL argument. For each attribute list, CodeBridge associates the parameter with a COBOL argument in one of three ways. The parameter may associate with one of the following:

- An implied argument
- The next argument
- The current argument

Automatic Association with an Implied Argument

The **arg_count**, **initial_state**, and **windows_handle** base attributes do not refer to a COBOL argument specified in the CALL statement. The CodeBridge Library supplies the value for the C parameter from an implied argument provided by the runtime environment at the time the CALL statement is executed.

Automatic Association with the Next Argument

The **address**, **float**, **general_string**, **integer**, **numeric_string**, **pointer_address**, **pointer_base**, and **string** base attributes refer to the next COBOL argument not yet associated with a C parameter. The first parameter attribute list (ignoring any attribute lists specified for the function return value) that contains one of these base attributes will associate the described C parameter with the first argument in the USING phrase of the COBOL CALL statement. The second such parameter attribute list will associate the described C parameter with the second argument in the USING phrase, and so forth.

A single C parameter may be associated with multiple COBOL arguments by the use of multiple attribute lists for that parameter.

Automatic Association with the Current Argument

The **buffer_length**, **digits**, **effective_length**, **length**, **pointer_length**, **pointer_offset**, **pointer_size**, **scale**, and **type** base attributes associate the described C parameter with the current COBOL argument. This behavior makes it possible to have a single COBOL

argument supply values for several contiguous C parameters. The current COBOL argument is the one last used by the automatic association method for the next argument as described in the previous topic, "Automatic Association with the Next Argument." If an attribute list containing a base attribute that associates with the next argument has not yet been specified, the current COBOL argument is the argument in the GIVING (RETURNING) phrase.

Examples of Associating Parameters with Arguments

Example 1: Automatic Versus Explicit Association

The following set of examples illustrates methods of associating parameters with arguments.

Example 1a: Automatic Association

In the following example, the C function moves the value of the parameter named FloatIn to the parameter named FloatOut after checking that the value will fit (using the values of the parameters named Digits and Scale). The function return value indicates success or failure.

The template file for the C function contains the following lines:

```
[[integer out]]            int fn(
[[float   out]]                   float *FloatOut,
[[digits]]                        int    Digits,
[[scale]]                         int    Scale,
[[float   in]]                    float  FloatIn);
```

The C function is called using the following COBOL statement:

```
CALL "fn" USING Float-Out, Float-In GIVING Fn-Status.
```

CodeBridge uses the automatic association method to associate the function return value with the GIVING argument named Fn-Status. The first three C parameters associate with the first USING argument named Float-Out, as follows:

- The first **float** base attribute causes the C parameter named FloatOut to be associated with the next (that is, in this case, the first) unassociated COBOL argument named Float-Out.
- The **digits** base attribute associates the C parameter named Digits with the current COBOL argument, which is the first argument named Float-Out.

- Similarly, the **scale** base attribute associates the parameter named Scale with the current argument, which is the first argument named Float-Out.

Finally, the second **float** base attribute associates the C parameter named FloatIn with the next (that is, in this case, the second) COBOL argument named Float-In.

Example 1b: Optional Explicit Association

The following template file accomplishes the same associations as in Example 1a, but by using the explicit association method:

```
[[integer out ret_val]]    int fn(
[[float   out arg_num(1)]]        float *FloatOut,
[[digits      arg_num(1)]]        int    Digits,
[[scale       arg_num(1)]]        int    Scale,
[[float   in  arg_num(2)]]        float  FloatIn);
```

Example 1c: Required Explicit Association

The automatic association method is possible only when the C parameters occur in the same order as the COBOL arguments. When they do not and you cannot change the C function, then the explicit association method is required. If the function in Example 1a were changed by moving the output floating-point parameter from first to last, then there would be no automatic association method that could achieve the desired result. In this case, the following explicit association method template file would be required:

```
[[integer out ret_val]]    int fn(
[[digits      arg_num(1)]]        int    Digits,
[[scale       arg_num(1)]]        int    Scale,
[[float   in  arg_num(2)]]        float  FloatIn,
[[float   out arg_num(1)]]        float *FloatOut);
```

Example 2: Multiple Attribute Lists for a C Parameter

The following group of examples illustrates how to associate multiple attribute lists with a single C parameter.

Example 2a: Associating a Parameter with Multiple Arguments

In the following example, the C function has a single input/output parameter, but the COBOL program wishes to pass the C function one input argument and two output arguments. This would allow one copy of the result to be stored in binary form while the other is stored in numeric edited form.

The template file for the C function contains the following lines:

```
void fn([[float in]]
        [[float out]]
        [[float out]] float *FloatInOut);
```

The C function is called using the following COBOL statement:

```
CALL "fn" USING Float-In, Binary-Out, Numeric-Edited-Out.
```

CodeBridge uses the automatic association method to associate each **float** base attribute with the next unassociated COBOL argument. This results in the C parameter named FloatInOut being associated with the first USING argument, named Float-In, during the input conversion process, and with the second and third arguments, named Binary-Out and Numeric-Edited-Out, respectively, during the output conversion process. The final value of the parameter named FloatInOut is converted by CodeBridge, during the output conversion process after the C function returns, to a COBOL binary number (assuming argument Binary-Out was described as a binary data item) and to a COBOL numeric edited number (assuming argument Numeric-Edited-Out was described as a numeric edited data item).

The following template file shows the equivalent explicit association method for this example:

```
void fn([[float in  arg_num(1)]]
        [[float out arg_num(2)]]
        [[float out arg_num(3)]] float *FloatInOut);
```

Example 2b: In Direction Attribute for Multiple Attribute Lists

Normally, when using multiple attribute lists with a single C parameter, only one of the attribute lists should contain the **in** direction attribute for a given C parameter. Consider the following modified template file:

```
void fn([[float in  arg_num(1)]]
        [[float in  arg_num(2)]]
        [[float out arg_num(3)]] float *FloatInOut);
```

Now there are two input arguments and only one output argument. The C function is called by the following COBOL statement:

```
CALL "fn" USING Float-In-1, Float-In-2, Binary-Out.
```

During the input conversion process, CodeBridge first converts the argument named Float-In-1 and stores the result in the parameter named FloatInOut, and second converts

the argument named Float-In-2 and stores it in the parameter named FloatInOut. The value of argument Float-In-1 previously stored in parameter FloatInOut is lost. This may be useful in a few circumstances where the side effects of the first conversion are desired (for example, checking the data type), but is probably almost never what was intended.

Example 2c: Compatibility between Multiple Attribute Lists

When using multiple attribute lists with a single C parameter, you must make sure that the attribute lists are compatible. Consider the following template file:

```
void fn([[float  in  arg_num(1)]]
        [[float  out arg_num(2)]]
        [[string out arg_num(3)]] float *FloatInOut);
```

The first two attribute lists describe a parameter that must be described with the C type specifiers **float** or **double**. The third attribute list describes a parameter that must be a C string parameter, that is, an array of type **char**. A single C parameter cannot be both types of data at the same time. Because the base attribute also determines the allowed types of COBOL arguments (in this case, a numeric argument is required), an error would occur when trying to convert the floating-point parameter, named FloatInOut, to the non-numeric argument, named String-Out, of the following COBOL statement:

```
CALL "fn" USING Float-In, Binary-Out, String-Out.
```

Example 3: No Attribute List for a C Parameter

In addition to allowing one or more attribute lists for a single C parameter, CodeBridge also allows C parameters without an attribute list. For such a parameter there are no input conversions, so the parameter is passed an uninitialized variable, and there are no output conversions, so the final value of the parameter is ignored.

In the following example, the C function takes a floating-point value as input and returns two output parameters, the integer part and the fractional part of the input parameter. The function return value indicates whether the fractional part is zero. If your COBOL program needs only the integer part, use the following template file:

```
int fn([[float   in]]  float  FloatIn,
       [[integer out]] long  *IntegerPartOut,
                       long  *FractionPartOut);
```

Call the C function using the following COBOL statement:

```
CALL "fn" USING Float-In, Integer-Part-Out.
```

Working with a Variable Number of C Parameters

When using a variable number of parameters in a C function prototype, the last parameter in the parameter list (the parameter that precedes the ellipsis) is used as a model for the additional parameters that may occur. In effect, the last listed parameter is treated as the first element of an array that contains a variable number of elements.

All attributes in the template file that apply to the last listed parameter also apply to the additional parameters. Use the **repeat**(*value*) base modifier attribute (see pages E-9 and E-12) in the attribute list for the last listed parameter to specify that there are additional C parameters. (For an illustration, see "Example 3: Accommodating a Variable Number of Parameters" in Appendix B, *CodeBridge Examples*.)

The following limitations apply when using a variable number of C parameters:

- Neither the last listed parameter nor any of the additional parameters may be arrays.
- All additional parameters must be of the same C data type as the last listed parameter.
- The ANSI C convention for variable number of parameters is supported. The older UNIX convention is not supported.

CodeBridge has limited support for C functions with a variable number of parameters. The following sections describe that support for numeric and string C parameters.

Repeating C Numeric Parameters

For numeric parameters that use the **float** and **integer** base attributes, all additional parameters must be the same type and size as the last listed parameter.

Repeating C String Parameters

numeric_string

For C string parameters that use the **numeric_string** base attribute, the last listed parameter and all additional parameters must be numeric strings. The size of the last listed parameter is used as the size of all additional parameters. For parameters with the **numeric_string** base attribute, the default size is four more than the digit length of the passed COBOL argument. However, the **size**(*value*) base modifier attribute (see page E-10) may be used to modify the default size as necessary.

general_string

For C string parameters that use the **general_string** base attribute, the last listed parameter and all additional parameters must be strings. The **general_string** base attribute allows some of the additional string parameters to be passed as numeric arguments while others are passed as non-numeric arguments. The size of the last listed parameter is used as the size of all additional parameters. For parameters with the **general_string** base attribute, the default size is the greater of one more than the length and four more than the digit length of the passed COBOL argument. The **size**(*value*) base modifier attribute (see page E-13) may be used to modify the default size as necessary.

string

For C string parameters that use the **string** base attribute, the last listed parameter and all additional parameters must be non-numeric strings. The size of the last listed parameter is used as the size of all additional parameters. For parameters with the **string** base attribute, the default size is one more than the length of the passed COBOL argument. The **size**(*value*) base modifier attribute (see page E-13) may be used to modify the default size as necessary.

Modifying COBOL Data Areas

CodeBridge allows two ways of modifying COBOL data areas. You can use the **out** direction attribute to tell CodeBridge to convert a C output (or input/output) parameter and store the results in the COBOL argument. Alternatively, you can pass the address of the COBOL data area to a C pointer.

The preferred method is using the **out** direction attribute to have CodeBridge store the result in the COBOL argument data item. The alternative method of passing the address requires the C function to know the details of COBOL data formats, thus negating one of the major benefits of using CodeBridge. Passing the address of the COBOL argument data item to your C function allows the C function to directly modify the value of the COBOL argument, even for input parameters.

Using the out Direction Attribute

Using the **out** direction attribute, possibly in conjunction with the **in** direction attribute, is the preferred method of modifying COBOL data areas. It provides all the flexibility of CodeBridge data conversion as well as the safety afforded by CodeBridge error checking and data validation. There are, however, several ways where you may not get the results you were expecting.

By way of review, the CodeBridge-generated code performs the following steps when a COBOL program calls a C function:

1. If requested, the code performs input argument validation.
2. For parameters with the **in** direction attribute specified or assumed, CodeBridge converts input arguments from COBOL to C data formats (performing error checks in the process) and stores the result in a temporary C data item.
3. CodeBridge calls the C function, passing to each parameter either the value or address of its temporary C data item.
4. If requested, the code performs output parameter validation.
5. For parameters with the **out** direction attribute specified, CodeBridge converts the final value for the temporary C data item from C to COBOL data format (performing error checks in the process) and stores the result in the COBOL argument.

There are several ways that the C function will fail to change the value of the COBOL argument:

- The first is that if step 3 passes the temporary C data item “by value” to the C function, the function cannot change the value of the temporary C data item, which will, therefore, be unchanged even if it is stored in step 5.
- The second is that if the parameter does not have the **out** direction attribute specified, step 5 is skipped and any change to the temporary C data item is discarded.
- The third is that if the COBOL program passed the COBOL argument using the BY CONTENT phrase (analogous to a C call “by value”), then step 5 will modify the contents of the temporary COBOL data area for the argument, which will then be discarded, leaving the original COBOL argument value unchanged.
- The fourth is that if the CALL statement omits the argument (either by specifying the OMITTED reserved word or specifying fewer arguments than expected) or if the COBOL argument is a null-valued pointer passed to a numeric or string parameter, step 5 has no place to store the modified value. (However, CodeBridge Builder does

not currently allow the **optional** base modifier attribute with the **out** direction attribute.)

In summary, you must do all of the following to modify a COBOL argument with the C function:

1. In the COBOL CALL statement, pass the COBOL argument BY REFERENCE rather than BY CONTENT. Since the BY REFERENCE phrase is the default for RM/COBOL, it does not have to be explicitly specified unless a preceding BY CONTENT phrase has overridden the default. RM/COBOL always passes the argument in the GIVING (RETURNING) phrase BY REFERENCE. Also, do not pass a null-valued pointer (see page 2-13) or omit the argument (see page 2-17).
2. In the CodeBridge template file, specify the **out** direction attribute for the C parameter. For the function return value, **out** is assumed.
3. In the C function, specify the parameter as call "by reference" so that the address of the temporary C data item is passed in step 3. In the following example, the first parameter is passed "by value" (as the value of an integer), while the second is passed "by reference" (as a pointer to an integer):

```
fn(int byValue, int *byReference);
```

Passing the Address of COBOL Data

There are times when you may choose to pass the address of the argument or the address of memory that is accessible by the COBOL run unit through a pointer data item. CodeBridge provides three base attributes that may be used for this purpose.

- Using the **address** base attribute passes the address of a COBOL argument to the C function as the parameter value and allows the C function to modify the COBOL data area directly. In the case of a pointer argument, the **address** base attribute returns the address of the pointer data item, which is not the address referred to by the pointer data item. The **length** base attribute may be used to determine the size of the COBOL argument.
- Using the **pointer_address** base attribute passes the effective address (base address plus offset) of a COBOL pointer argument to the C function as the parameter value and allows it to manipulate the contents of the block of memory directly. However, using the **pointer_address** base attribute prevents the C function from changing the value of the COBOL pointer. The **pointer_length** base attribute may be used to determine the effective length (size minus offset) of the memory block.
- Using the **pointer_base** base attribute passes the base address component value of a COBOL pointer argument to the C function as the parameter value and allows the C function to change the value of the pointer base address component as well as the

contents of the block of memory. The **pointer_offset** and **pointer_size** base attributes may be used to manipulate the offset and size components of the COBOL pointer argument.

Note The C function may save in static storage the address obtained by using any of the three base attributes described above. The saved address may then be used in subsequent calls. It is the developer's responsibility to avoid use of a saved address that points to a data item in a COBOL program that has been canceled or to a dynamically allocated memory block that the COBOL program has subsequently deallocated.

Passing Buffer Addresses

In some existing APIs, it is necessary to pass a buffer address to a C function. Later, that buffer address is used by another C function in the API to store a result value as a C data item. In such cases, the preferred method of using the **out** direction attribute cannot be used and the address of the buffer must be passed instead. CodeBridge may still be used in such cases to convert the C data item to a COBOL data format after the result has been stored in the buffer. See "Example 6: Converting Buffered C Data" in Appendix B, *CodeBridge Examples*, for details on the CodeBridge solution to this problem for a C string result in the buffer.

Using P-Scaling

In COBOL, P-scaling is used when working with large integers that have several trailing zero digits before the decimal point or with small fractions that have several leading zero digits after the decimal point. It is commonly used to store values representing thousands, millions, or billions. For example, the PICTURE clause "PIC 9(4)P(3)" is used to represent all integers from 0 to 9,999,000 in units of 1000. The value 1,234,000 would be stored as 1,234, but would continue to mean 1,234,000.

For input conversions of P-scaled numbers, CodeBridge supplies the missing zero digits. For output conversions, the extra digits are eliminated by truncation or rounding. Continuing with the example in the preceding paragraph and using the attribute list `[[float in out rounded]]` for the input conversion, CodeBridge would convert the stored value (1,234) and pass the floating-point representation of 1,234,000 to the C function. If the C function added 999 to its parameter, then the output conversion would round 1,234,999 to 1,235,000 and store 1,235 in the COBOL argument. Adding any number up to 499 would leave the COBOL argument unchanged. When the **rounded** base modifier attribute is not present, CodeBridge truncates the result on output, converting 1,234,999 to 1,234,000 and storing 1,234 in the COBOL argument.

P-scaling also affects the **scale** base attribute. Because of P-scaling, the scale of the COBOL argument in our example is minus three (-3). As another example, the PICTURE character-string "VP(3)9(3)" has a scale of six (6), even though the digit count is only three (3).

Any P-scaling specified in the PICTURE character-string is counted in the digit length used by CodeBridge when allocating a conversion string buffer for a parameter described with the **general_string** or **numeric_string** base attribute. That is, the digit length used by CodeBridge is the sum of the number of 9 and P symbols specified in the PICTURE character-string used to describe the argument data item.

Working with Arrays

Data items having numeric or string base attributes may be one-dimensional arrays. Data items with string base attributes may be arrays of **char ***, which are similar to two-dimensional arrays.

Numeric Arrays

For simple numeric types, such as integer or floating-point, the implementation is straightforward. Examples of valid C numeric array parameters are as follows:

```
fn(char   P1[10],
   char  *P2,
   int    P3[40],
   float *P4,
   float  P5[]);
```

The first two parameters, which use the **char** data type, are normally used to represent character strings. However, you can have a numeric array of characters. The difference is how the called function interprets the data.

To specify the template file for the preceding C function prototype, you might start with the following, for example:

```
fn([[integer in]]           char   P1[10],
   [[integer in]]           char  *P2,
   [[integer in]]           int    P3[40],
   [[float   in]]           float *P4,
   [[float   in]]           float  P5[]);
```

Although the attribute lists for the variables P2, P4, and P5 are valid C code, CodeBridge needs to know the size of the array. You could modify the template file by changing the following:

```
char *P2      to    char P2[20]
float *P4     to    float P4[20]
float P5[]    to    float P5[10]
```

However, the template file would no longer match the C function prototype.

An alternate method is to specify an occurs count in the attribute list by modifying the template file as follows:

```
fn([[integer in occurs(10)]] char   P1[10],
   [[integer in occurs(20)]] char  *P2,
   [[integer in occurs(20)]] int    P3[40],
   [[float   in occurs(20)]] float *P4,
   [[float   in occurs(10)]] float  P5[]);
```

The attribute lists for variables P2, P3, and P4 now have an array size of 20 elements. For variables P1 and P3, the **occurs**(*value*) base modifier attribute overrides the value specified in the function prototype. For variables P2, P4, and P5, the **occurs**(*value*) base modifier attribute provides a value that was missing in the function prototype. Note that the attribute list for variable P1 did not change the size of the array, while the attribute list for variable P3 reduced the size of the array. Reducing the size of the array is required if the COBOL program passes a smaller array since CodeBridge will convert the number of array elements indicated by the template.

String Arrays

The implementation of these types of arrays is more complex because strings are already arrays of characters. One-dimensional arrays of C parameters with a **string** base attribute are allowed (this means that, as a special case, two-dimensional arrays of characters are allowed). Examples of valid C string array parameters are as follows:

```
fn(char  *P1[10],
   char  *P2[],
   char **P3);
```

To specify the template file for the preceding C function prototype, you might start with the following, for example:

```
fn([[string         in]]            char  *P1[10],
   [[numeric_string in]]            char  *P2[],
   [[general_string in]]            char **P3);
```

Note that a difference between **string** and **numeric_string** attribute lists is how the data is interpreted by the called function. However, both provide null-terminated arrays of characters. A **general_string** base attribute may be used to allow numeric and non-numeric arguments to be converted to null-terminated arrays of characters. A **general_string** base attribute applies the rules for the **numeric_string** base attribute when the argument is numeric and the rules for the **string** base attribute when the argument is non-numeric.

You must modify the attribute list for the variables P2 and P3 because CodeBridge must know how many string pointers to allocate. Add an **occurs**(*value*) base modifier attribute for variables P2 and P3 and then modify the C function prototype to make it work correctly (note that you only need to make these changes in the template file, not in the actual C header file). For example, modify the template file as follows:

```
fn([[string         in           ]] char  *P1[10],
   [[numeric_string in occurs(10)]] char  *P2[],
   [[general_string in occurs(10)]] char  *P3[]);
```

For variables P2 and P3, the **occurs**(*value*) base modifier attribute provides information needed to allocate the string pointer arrays. The definition of parameter P3 was changed from “char **P3” to the equivalent form “char *P3[]”.

CodeBridge allocates memory for strings (or arrays of strings) with a single memory allocation call. The generated code contains declarations in the form:

```
char *P1[10];
char *P2[10];
char *P3[10];
```

Each element of the array is initialized to point to the correct offset within the allocated block.

The number of elements in the array and the size of each element determine the size of the allocated block. For a **numeric_string**, the size of each element is equal to four more than the digit length of the COBOL argument. For a **string**, the size of each element is equal to one more than the length of the COBOL argument. For a **general_string**, the size of each element is equal to the greater of four more than the digit length and one more than the length of the COBOL argument.

You may override these default element sizes by using the **size**(*value*) base modifier attribute as follows:

```
fn([[string         in            size(30)]] char *P1[10],
   [[numeric_string in occurs(10) size(35)]] char *P2[],
   [[general_string in occurs(10) size(20)]] char *P3[]);
```

COBOL Array References

When passing an array reference from COBOL to C, you must pass the first item of the COBOL array. For example:

```
CALL "fn" USING Data-Item (1).
```

The OCCURS information for a COBOL data item is not passed to a non-COBOL subprogram. This means that CodeBridge cannot determine the number of elements in a COBOL array from the COBOL descriptor for that item. This is true for both the maximum occurs value and the depending value. If desired, the COBOL program could pass either of these values as separate parameters. The COBOL special registers COUNT, COUNT-MAX, and COUNT-MIN may be used to obtain the current number of occurrences, the maximum number of occurrences specified in the COBOL OCCURS clause, and the minimum number of occurrences specified in the COBOL OCCURS clause.

CodeBridge converts the number of COBOL occurrences specified in the template file regardless of the number of actual occurrences in the COBOL program or any occurrence count parameter. Therefore, the COBOL program that calls the function described by the template must always pass an array that has at least as many occurrences as specified by the template. If the COBOL program defines fewer occurrences than specified in the template, CodeBridge will convert data following the array argument in the COBOL data area. In such cases, output conversion will overwrite data following the array argument, possibly destroying the integrity of the COBOL program.

CodeBridge only handles COBOL table references that are not SYNCHRONIZED. That is, Data-Item in the preceding example must be described with the OCCURS clause and must not be described with the SYNCHRONIZED (SYNC) clause. CodeBridge supports only singly dimensioned tables of COBOL arguments. A multidimensional table may be passed, but only the last subscript will be varied by CodeBridge. Further, the table must contain contiguous elementary items. That is, the last subscript must be for an OCCURS clause in the argument item description rather than a group item that contains the argument item.

CodeBridge Builder

This section describes the CodeBridge Builder, which reads a template file as input and generates C source as output. This generated source provides the interface between the COBOL program and the C function by calling functions in the CodeBridge Library to convert between COBOL arguments and C parameters, as needed, before and after calling the target C function.

For each C function prototype in the template file, a corresponding function is generated in the DLL interface code. Each function contains all of the logic needed to do the following:

- Produce an exportable DLL function
- Optionally perform input argument validation
- Convert input arguments from COBOL to C
- Call the C function
- Optionally perform output parameter validation
- Convert output parameters from C to COBOL

Using the CodeBridge Builder

The CodeBridge Builder is a command line program (for Windows, a console application). The application program file is named **cbridge.exe**.

To start the CodeBridge Builder from the command line, enter:

```
cbridge <input file> [<output file>] [-f (-F)]
```

where:

<*input file*> is the pathname of the template file. This parameter is required. If you do not supply an extension, the CodeBridge Builder will add the extension **.tpl**.

<*output file*> is the pathname for the generated source file. This parameter is optional. If it is not specified, the value of <*input file*> will be used with the extension changed to **.c**.

-f (or -F) is a command line option that may be used to force CodeBridge Builder to generate C source code, even if errors are encountered. This parameter is optional. If it is specified, any error messages will be concatenated to the end of the generated

source in addition to appearing in the error file. The error file is always generated, regardless of whether the -f option is specified.

Note The generated C source contains a **#include** C preprocessor directive that refers to the additional header files: **rmc85cal.h**, **rmport.h**, **rtarg.h**, **rtcallbk.h**, and **standdef.h**. All of these files are installed with CodeBridge.

If errors are encountered, an error file is generated (see "CodeBridge Builder Error Messages" in Appendix A, *CodeBridge Errors*). The error file uses the same pathname as <*output file*> with the extension changed to **.err**.

For example, the command:

```
cbridge src\myfile.tpl
```

reads **src\myfile.tpl**, writes the generated source to **src\myfile.c**, and writes any error messages to **src\myfile.err**.

The command:

```
cbridge tpl\myfile src\myfile.src
```

reads **tpl\myfile.tpl**, writes the generated source to **src\myfile.src**, and writes any error messages to **src\myfile.err**.

The CodeBridge Builder checks for errors in the template file and if any errors are present, it produces a file that contains diagnostic information. If there are errors in the template file, however, no output file will be generated. When there are errors in the template, the resultant source file should be considered unusable even though a C compiler might compile it without errors.

Note The CodeBridge Builder exit codes are also described in Appendix A, *CodeBridge Errors*.

Appendix A: CodeBridge Errors

This appendix lists and describes the messages that can be generated during the use of either the CodeBridge Builder or the CodeBridge Library. These messages also include the CodeBridge Builder exit codes.

CodeBridge Builder Error Messages

CodeBridge Builder error messages have the following form:

```
<file>(<line>) <severity> - <message number>: <message text>
```

where *severity* can be either “inform” or “error”.

Table A-1 lists the error messages produced by the CodeBridge Builder.

Table A-1: CodeBridge Builder Error Messages

Message Number	Message Text
100010	The template element is not correctly formed.
100020	The **#include** directive is not correctly formed.
100030	The user function is not correctly formed.
100040	The attribute is not correctly formed.
100045	The attributes are not correctly formed.
100050	The attribute expression’s element is not correctly formed.
100060	The attribute value clause is not correctly formed.
100070	The attribute clause is not correctly formed.
100080	The C function’s header is not correctly formed.
100090	The name declaration is not correctly formed.
100100	The array declaration is not correctly formed.
100110	The argument list is not correctly formed.
100120	The argument is not correctly formed.
100130	There is no such attribute [[*attribute_name*]].
100140	The attribute [[*attribute_name*]] can’t have a value.

Table A-1: CodeBridge Builder Error Messages *(Cont.)*

Message Number	Message Text
100150	The attributes [[*attribute_name*]] and [[*attribute_name*]] are incompatible.
100160	One of the minimal attribute combinations must be present: [[*attribute combinations*]].
100180	Either the [[arg_num]] or [[ret_val]] attribute must not be used, since it wasn't used on a previous parameter.
100190	Either the [[arg_num]] or [[ret_val]] attribute must be used, since it was used on a previous parameter.
100210	The global attributes are not correctly formed.
100220	The global attribute is not correctly formed.
100230	There is no such global attribute [[*attribute_name*]].
100240	The attribute [[*attribute_name*]] must have *number* value(s).
100250	The global attribute's convention value clause is not correctly formed.
100260	The global attribute's replace value clause is not correctly formed.
100270	The global attribute's normal value clause is not correctly formed.
100280	The global name declaration is not correctly formed.
100285	Duplicate global attribute: [# *attribute_name* #]
100290	There is no such diagnostic value: (*value*).
100300	The number of the argument with [[repeat]] attribute is not the highest.

The CodeBridge Builder uses the following data files: **dllgen.in**, **dllgen.out**, **dllgen.p01**, and **dllgen.sym**. Occasionally, if these files are write-protected, the CodeBridge Builder may not be able to open them, and an error message similar to the following will be displayed:

```
C:\TOOLS\SCANNER.EXE: FAILURE
- Unable to open file 'C:\TOOLS\DLLGEN.xxx'.
```

If this occurs, modify the attributes of these four files so that they are not write-protected.

CodeBridge Builder Exit Codes

The CodeBridge Builder will return a completion status (or exit code). This status can be interrogated by the batch stream or shell script. Table A-2 lists the CodeBridge Builder exit codes.

Table A-2: CodeBridge Builder Exit Codes

Code	Description
0	Normal program termination with no diagnostic messages produced.
1	Normal program termination with some diagnostic messages produced.
253	Abnormal program termination—error creating temporary file.
254	Abnormal program termination—error executing program.
255	Abnormal program termination—an internal error occurred.

CodeBridge Library Error Messages

An execution error in the CodeBridge Library causes the called C subprogram to exit and the COBOL run unit to terminate.

When a CodeBridge Library function detects an error during conversion or validation, it displays an error message before returning to the calling program.

Note The errors displayed by the CodeBridge Library are in addition to errors that may subsequently be displayed by the RM/COBOL runtime system. See Appendix A, *Runtime Messages,* in the *RM/COBOL User's Guide*.

A CodeBridge Library error message contains the following information:

Function: <*calling function name*>
Argument Number: <*number*> (or Argument: Return Value)
Operation: <*library function name*>
Error: <*error number*> - <*message text*>

<*calling function name*> is the *Name* parameter from the last call to ConversionStartup (see page F-42).

<*number*> is the one-based argument number of the argument in the USING phrase. When the alternative, Return Value, is shown, it indicates the argument in the GIVING (RETURNING) phrase.

<*library function name*> is the conversion or validation operation specified as one of the names listed in the “Function Name” column of Table F-1 on page F-2. For example, CobolToInteger, which is described beginning on page F-29, would be specified if the error occurred during conversion of a COBOL numeric argument to a C integer parameter.

<*Error number*> is the “Error Code” and <*message text*> is the “Error Text” listed in Table A-3 on page A-5.

For Windows platforms, a message box with the error message is displayed. The following shows an example of a CodeBridge Library error message on Windows:

For UNIX platforms, the message is written to **stderr**. The following shows an example of a CodeBridge Library error message on UNIX:

```
CodeBridge Library Error
 Function: CINT2INTEGER
 Argument Number: 2
 Operation: CobolToString
 Error: 515 - Non-numeric data expected
```

Table A-3: CodeBridge Library Errors

Error Code	Error Text	Description
501	Digits count too large	One of the base modifier attributes (**assert_digits**, **assert_digits_left**, or **assert_digits_right**) was specified and the corresponding number of digits in the passed COBOL argument was greater than the indicated maximum.
502	Digits count too small	One of the base modifier attributes (**assert_digits**, **assert_digits_left**, or **assert_digits_right**) was specified and the corresponding number of digits in the passed COBOL argument was less than the indicated minimum.
503	Initialization needed	A call was made to a CodeBridge Library function prior to calling the ConversionStartup function. This error should never occur when using the CodeBridge Builder.
504	Integer data expected	The **integer_only** base modifier attribute was specified and the COBOL argument contains digits to the right of the decimal point.
505	Internal logic – Argument setup	This indicates an incompatibility between the RM/COBOL compiler and runtime. The descriptor of the COBOL argument contained unexpected values.
506	Internal logic – Data type	This indicates an incompatibility between the RM/COBOL compiler and runtime. The type of the COBOL argument contained an unexpected value.
507	Internal logic – Parameter setup	This indicates a logic error in the CodeBridge Library. While setting up a description of the C parameter, an unexpected condition was encountered.
508	Invalid argument number	The argument number supplied was not valid. This could indicate an internal error with the CodeBridge Builder or that the developer used a bad value when calling a CodeBridge Library function directly.
509	Invalid C numeric string	`[[numeric_string out]]` was specified and the C string is not numeric.

Table A-3: CodeBridge Library Errors *(Cont.)*

Error Code	Error Text	Description
510	Invalid data type	The COBOL argument contains an unsupported data type.
511	Invalid sign specification	The COBOL argument contains an invalid sign.
512	Length too large	The **assert_length** base modifier attribute was specified and the corresponding length of the passed COBOL argument was greater than the indicated maximum.
513	Length too small	The **assert_length** base modifier attribute was specified and the corresponding length of the passed COBOL argument was less than the indicated minimum.
514	Memory allocation error	The CodeBridge Library attempted to allocate memory and encountered an error.
515	Non-numeric data expected	A numeric COBOL argument was used with the **string** base attribute.
516	Null pointer not allowed	The COBOL program passed a null pointer when the **no_null_pointer** base modifier attribute was used.
517	Numeric data expected	A non-numeric COBOL argument was used with one of the following numeric base attributes: **float**, **integer**, or **numeric_string**.
518	Omitted argument not allowed	The COBOL argument was omitted for an argument that was not optional.
519	Pointer data expected	The COBOL argument was not a POINTER when a pointer base attribute was used.
520	Signed argument expected	An unsigned numeric COBOL argument was used when the **signed** base modifier attribute was set.
521	Size error	A size error occurred during numeric data conversion and the **no_size_error** base modifier attribute was not set.
522	Size not supported	The size of the C parameter does not conform to one of the supported C numeric data types (such as **int** or **float**).
523	Unsigned argument expected	A signed numeric COBOL argument was used when the **unsigned** base modifier attribute was set.

Table A-3: CodeBridge Library Errors *(Cont.)*

Error Code	Error Text	Description
524	Version level mismatch	This version of the CodeBridge Library does not support the minimum level of conversion and validation features indicated by the *Version* parameter of the ConversionStartup call.
525	Effective_length occurs too large	The occurs count for an **effective_length** base attribute is larger than the occurs count for the C parameter associated with the same argument number. The occurs count for the **effective_length** base attribute must be less than or equal to the occurs count for the associated C parameter.

Appendix B: CodeBridge Examples

This appendix contains examples that use the typical CodeBridge development process outlined in Chapter 1, *Introduction*. The examples build from simple to complex, as a means of introducing CodeBridge concepts, which are discussed in Chapter 2, *Concepts*.

In addition to these examples, there are several CodeBridge sample programs that are included with the development system in the CodeBridge samples subdirectory (**cbridge** on Windows and **cbsample** on UNIX). See the appropriate README file (and the **samples.txt** file on Windows) for additional information about the CodeBridge sample programs that are included.

Note 1 In the following example template files, bold type is used to indicate the first instance of a CodeBridge attribute that is being introduced. Detailed information about attributes and attribute lists is provided in Appendix D, *Global Attributes*, and Appendix E, *Parameter Attributes*.

Note 2 Unlike COBOL, C is a case-sensitive programming language. Thus, the case is significant for words in these example template files.

Example 1: Calling a Standard C Library Function

This example demonstrates calling a standard C library function without writing any C code. Parameter attribute lists are also presented. See the details of this example on page 1-9 in the "Typical Development Process Example" section.

Example 2: Calling a Windows API Function

This example demonstrates calling a Windows API function to display a message box. Both global attribute lists and parameter attribute lists are used.

Note Since this example deals with a Windows API function, it is fully elaborated only for Windows, where the ODBC API is readily available from Microsoft. However, the CodeBridge techniques illustrated are general in nature and may be instructive to developers creating templates for C subprograms on UNIX systems.

1. Start with the function prototype for the Windows API function, MessageBox:

```
WINUSERAPI int WINAPI MessageBox(HWND hWnd,
  LPCSTR lpText, LPCSTR lpCaption, UINT uType);
```

2. Create a template file named **mbox.tpl** in the **src** directory that consists of the following lines:

```
#include <windows.h>
#include <winuser.h>

[# replace_type(LPCSTR; char *)
   convention(WINUSERAPI)
   convention(WINAPI) #]

[[integer out]] WINUSERAPI
            int WINAPI MessageBox(
[[windows_handle]]                HWND    hWnd,
[[string  in trailing_spaces]]    LPCSTR lpText,
[[string  in trailing_spaces]]    LPCSTR lpCaption,
[[integer in unsigned]]           UINT    uType);
```

The template file needs **#include** C preprocessor directives for files that contain any required defined data types (using macros defined with the **#define** C preprocessor directives and C data types defined with **typedef** statements). In this example, the **windows.h** and **winuser.h** header files are included.

Global attribute lists (for example, `[# replace_type(LPCSTR; char *) #]`) are constructed by placing the attributes between the characters [# and #]. The two global attributes used in this example are **replace_type** and **convention**.

The **replace_type** global attribute causes CodeBridge to replace a defined C type with the specified value. In this example, the type LPCSTR is replaced with the value **char ***, which is required whenever the definition of a pointer is hidden within

a defined type. The number of levels of indirection (indicated by asterisks) in a C data type tells CodeBridge Builder how to correctly build calls to the C function.

The **convention** global attribute informs the CodeBridge Builder that a particular text string represents a calling convention to a C function. CodeBridge must preserve the calling convention in the constructed external reference to the C function while removing it from the definition of the generated variable used to hold the function return value.

Several new parameter attributes are introduced. The **integer** base attribute is used when the type of the C parameter is an integer (such as **char**, **short**, **int**, **unsigned**, or **long**). The **string** base attribute is used when the type of the C parameter is a string (an array of characters) and the type of the COBOL argument is non-numeric.

Some parameter base attributes do not obtain information directly from a COBOL argument. One of these is the **windows_handle** base attribute, which obtains its value from the Windows handle associated with the calling program (in this case, the Windows handle of the RM/COBOL runtime system).

There are two input strings in this example. The attribute list `[[string in trailing_spaces]]` is used for both of them. When an input string is encountered, a conversion buffer is allocated to contain the string. The data is copied from the COBOL argument and a trailing null is appended. The **trailing_spaces** base modifier attribute causes trailing spaces to be removed before the null character is added for input conversions (for output conversions, the null character is removed and trailing spaces are appended).

One of the C parameters is of type UINT, which has a value of unsigned integer. The **unsigned** base modifier attribute ensures that the CodeBridge Library treats the data as unsigned.

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\mbox.tpl
```

This command reads the input file from **src\mbox.tpl** and writes its output file to **src\mbox.c**. Any errors would be written to file **src\mbox.err**

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice, using commands similar to the following:

```
cl -c –MD -Zp1 src\mbox.c

link -nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:mbox.dll
      mbox.obj kernel32.lib user32.lib
```

5. Create a COBOL program in a file named **mbox.cbl** that contains the following source fragments:

```
77 NUMBER-1 PIC 99.
77 NUMBER-2 PIC 99.
77 NUMBER-3 PIC 99.
77 NUMBER-4 PIC 99.
77 NUMBER-5 PIC 99.
77 NUMBER-6 PIC 99.
77 TEXT-1   PIC X(256).
77 RESULT   PIC 99.

78 CR-LF         Value X"0D0A".
78 MB-OK-BUTTON  Value  0.
78 MB-INFO-ICON  Value 64.
78 MB-STYLE      Value MB-OK-BUTTON + MB-INFO-ICON.
78 MB-CAPTION    Value "LOTTERY".

STRING "Today's winning lottery numbers" CR_LF
    NUMBER-1 " – " NUMBER-2 " – " NUMBER-3 " – "
    NUMBER-4 " – " NUMBER-5 " – " NUMBER-6
    DELIMITED BY SIZE INTO TEXT-1.
CALL "MessageBox" USING  TEXT-1 MB-CAPTION MB-STYLE
                  GIVING RESULT.
```

   The COBOL code creates a message box containing the text, “Today’s winning lottery numbers *xx – xx – xx – xx – xx – xx*”, where *xx* represents one of the six lottery numbers. (The code for setting NUMBER-1 through NUMBER-6 is not shown.)

   Note The value of the Windows handle parameter, named hWnd, is supplied by the RM/COBOL runtime system. It does not have an associated COBOL argument.

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol mbox
```

7. Run the application with the following command line:

```
runcobol mbox -l mbox.dll
```

Example 3: Accommodating a Variable Number of Parameters

This example uses an alternate method to create the same message box that was presented in Example 2. It also demonstrates calling a C function that accepts a variable number of parameters.

Note Since this example deals with a Windows API function, it is fully elaborated only for Windows. However, the CodeBridge techniques illustrated are general in nature and may be instructive to developers creating templates for C subprograms on UNIX systems.

1. Create a C function, **message_box** (which calls the Windows API function, MessageBox), in a file named **mbox2fn.c** in the **src** directory that consists of the following lines:

```
#include <windows.h>
#include <winuser.h>

int message_box(HWND hWnd, int ArgCount,
     int Options, char *Title, char *Text, ...)
{
  int i;
  char MessageText[512];
  va_list Marker;

  strcpy(MessageText, Text);
  va_start(Marker, Text);
  for (i = 4; i <= ArgCount; i++)
    strcat(MessageText, va_arg(Marker, char*));
  va_end(Marker);

  return(MessageBox(hWnd, MessageText, Title, Options));
}
```

Note 1 The function has a variable number of string parameters (represented on the function prototype by the ellipsis "…"), which are concatenated to form a single text string. This allows the calling COBOL program to pass these strings separately instead of using a STRING statement to concatenate them as was done in Example 2.

Note 2 Although it would seem logical to name the file that contains the **message_box** function **mbox2.c** and the file that contains the template **mbox2.tpl**, the CodeBridge Builder names its output file **mbox2.c** and thus would overwrite the file containing **message_box** were it also named **mbox2.c**.

2. Create a template file named **mbox2.tpl** in the **src** directory that consists of the following lines:

```
[[integer out]] int message_box(
[[windows_handle]]              HWND  hWnd,
[[arg_count]]                   int   ArgCount,
[[integer in unsigned]]         int   Options,
[[string in trailing_spaces]]   char *Title,
[[general_string in
  trailing_spaces
  leading_minus   repeat(20)]]  char *Text, ...);
```

The **arg_count** base attribute (like the **windows_handle** base attribute introduced in Example 2) is not associated with a COBOL argument. It is used to pass the actual number of COBOL arguments to the C function. This allows the **message_box** function to determine, for each call, how many strings have been passed.

CodeBridge offers several ways to pass a string to a C function:

- The **string** base attribute is used when the COBOL argument is non-numeric.
- The **numeric_string** base attribute is used when the COBOL argument is numeric.
- The **general_string** base attribute is used in those cases when it is desirable to allow a C string parameter to accept either a numeric COBOL argument or a non-numeric COBOL argument. When a numeric argument is passed to a parameter described with the **general_string** base attribute, the argument is converted as if the parameter were described with the **numeric_string** base attribute; otherwise, the argument is converted as if the parameter were described with the **string** base attribute. An attribute list containing the **general_string** base attribute allows any additional attributes that may be used with either a **string** base attribute or a **numeric_string** base attribute. For each call and for each argument passed to a parameter within a set of a variable number of parameters, attributes that do not apply to the COBOL argument actually passed are ignored for the conversion of that argument. That is, for a numeric argument, base modifier attributes not applicable to the **numeric_string** base attribute are ignored and for a non-numeric argument, base modifier attributes not applicable to the **string** base attribute are ignored.

In this example, when a non-numeric argument is passed to the parameter named Text, the **trailing_spaces** base modifier attribute will be acted upon and the **leading_minus** base modifier attribute will be ignored. When a numeric argument is passed, the opposite will occur.

The **leading_minus** base modifier attribute is used in **numeric_string** and **general_string** parameter attribute lists to specify that a minus sign character should be placed before the digits of the parameter value when the COBOL argument is a negative number. For more information, see the discussion of the **leading_minus** base modifier attribute in the "Base Modifiers that Apply to Numeric Base Attributes" section on page E-7.

The **repeat**(*value*) base modifier attribute provides partial support for C functions with a variable number of parameters. The **message_box** function uses the ellipsis (…) to indicate that it can accept any number of parameters following the parameter named Text. While CodeBridge Builder does not allow an unspecified number of trailing parameters, it does support a fixed number of extra parameters (in this example, repeat(20) specifies up to 20 extra string parameters, which may be associated with numeric or non-numeric arguments because of the **general_string** base attribute).

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\mbox2.tpl
```

   This command reads the input file from **src\mbox2.tpl** and writes its output file to **src\mbox2.c**. Any errors would be written to file **src\mbox2.err**.

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice. There are now two C files to compile:

   - The **message_box** function (created in step 1) in the file named **mbox2fn.c**.
   - The file named **mbox2.c** (created in step 3 by CodeBridge Builder when it processed the file named **mbox2.tpl**, created in step 2).

   Use commands similar to the following:

```
cl -c –MD -Zp1 src\mbox2fn.c
cl -c –MD -Zp1 src\mbox2.c

link -nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:mbox.dll
     mbox2.obj mbox2fn.obj kernel32.lib user32.lib
```

5. Create a COBOL program in a file named **mbox2.cbl** that contains the following source fragments:

```
77 NUMBER-1 PIC 99.
77 NUMBER-2 PIC 99.
77 NUMBER-3 PIC 99.
77 NUMBER-4 PIC 99.
77 NUMBER-5 PIC 99.
77 NUMBER-6 PIC 99.
77 TEXT-1   PIC X(256).
77 RESULT   PIC 99.

78 CR-LF         Value X"0D0A".
78 MB-OK-BUTTON  Value  0.
78 MB-INFO-ICON  Value 64.
78 MB-STYLE      Value MB-OK-BUTTON + MB-INFO-ICON.
78 MB-CAPTION    Value "LOTTERY".

CALL "message_box" USING MB-STYLE MB-CAPTION
    "Today's winning lottery numbers" CR-LF
    NUMBER-1 " – " NUMBER-2 " – " NUMBER-3 " – "
    NUMBER-4 " – " NUMBER-5 " – " NUMBER-6
    GIVING RESULT.
```

   The COBOL code creates a message box containing the text, "Today's winning lottery numbers *xx – xx – xx – xx – xx – xx*", where *xx* represents one of the six lottery numbers. (The code for setting NUMBER-1 through NUMBER-6 is not shown.)

   Note The parameters to **message_box** have been reordered so that the variable parameters occur at the end. For this reason, the arguments of the COBOL CALL have been similarly reordered. Values for the Windows handle and argument count parameters, named hWnd and ArgCount, respectively, are supplied by the RM/COBOL runtime system.

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol mbox2
```

7. Run the application with the following command line:

```
runcobol mbox2 -l mbox2.dll
```

Example 4: Accessing COBOL Pointer Arguments

This example shows how to access data described by pointer data items and demonstrates how dynamic memory management can be implemented. It also illustrates that a COBOL pointer argument can be used with both the C function return value and a C parameter. Finally, it shows the use of more than one attribute list for a single C parameter.

Note While the C functions illustrated in this example could be used for providing dynamic memory allocation, RM/COBOL supplies the subprograms C$MemoryAllocate and C$MemoryDeallocate in its subprogram library as described in Appendix F, *Subprogram Library*, of the *RM/COBOL User's Guide*. Those subprograms, in most circumstances, should be used to provide dynamic memory allocation in RM/COBOL.

A COBOL pointer data item describes a block of memory. It contains three components: base address, offset, and size. When a pointer data item is initialized, the base address contains the starting address of the block, the offset is set to 0, and the size contains the total length of the block.

CodeBridge pointer base attributes are used when COBOL pointer arguments are being passed to the C function. CodeBridge provides two approaches for accessing data described by a pointer data item. The first approach is used when the C function only needs to access the data referenced by the pointer. The second approach is used when the C function also needs to access the components of the pointer argument itself. The following example demonstrates the second approach.

1. Start with the function prototypes for the standard C library memory allocation functions, **free**, **malloc**, and **realloc**:

```
void     free(void *memblock);
void  *malloc(size_t size);
void *realloc(void *memblock, size_t size);
```

2. Create a template file named **mem.tpl** in the **src** directory that consists of the following lines:

```
#include <stdlib.h>
#include <malloc.h>

                                  void     free(
[[pointer_base in]]                             void *memblock);

[[pointer_base out
  pointer_reset_offset
                     ret_val]] void  *malloc(
[[integer      in arg_num(1)]]
[[pointer_size out   ret_val]]                  size_t size);

[[pointer_base out   ret_val]] void *realloc(
[[pointer_base in arg_num(1)]]                  void *memblock,
[[integer      in arg_num(2)]]
[[pointer_size out   ret_val]]                  size_t size);
```

The **arg_num** and **ret_val** argument number attributes are used to refer to COBOL arguments when they are passed by the calling program in an order that differs from the parameter order of the C function. For more information on associating C parameters with COBOL arguments, see “Associating C Parameters with COBOL Arguments” on page 2-21.

Note When the **arg_num** or **ret_val** argument number attributes are used for any attribute list, they must be used for every attribute list of that function.

The **pointer_base** and **pointer_size** base attributes refer to the base address component and size component, respectively, of a COBOL pointer argument. The **pointer_reset_offset** base modifier attribute is used with **pointer_base** base attribute to set the offset component to zero.

The **free** function, which deallocates memory, uses the **pointer_base** base attribute to describe an input parameter that provides the base address of the memory block that will be freed.

The **malloc** function, which allocates memory, uses the **pointer_base** base attribute to describe an output parameter that receives the base address of the allocated memory using the function return value. The **pointer_reset_offset** base modifier attribute sets the offset component to zero. The **malloc** function also uses the **pointer_size** base attribute to describe an output parameter that sets the pointer size component from the input parameter named size.

The **realloc** function, which changes the size and possibly the address of the block of memory, differs from the **malloc** function in three ways. It does not reset the pointer offset component to zero (the old value is retained). It also expects the address of the current memory block as an input parameter (in this case, the **pointer_base** base attribute is used with argument 1 to satisfy this expectation).

Finally, the parameter named size has two attribute lists. The first attribute list supplies the new block size from the second COBOL argument in the USING phrase to the size parameter. The second attribute list sets the size component of the argument in the GIVING (RETURNING) phrase from the size parameter.

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\mem.tpl
```

This command reads the input file from **src\mem.tpl** and writes its output file to **src\mem.c**. Any errors would be written to file **src\mem.err**.

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice, using commands similar to the following:

For Windows

```
cl -c –MD -Zp1 src\mem.c

link -nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:mem.dll
     mem.obj kernel32.lib user32.lib
```

For UNIX

A makefile is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object to be used as a support module with the RM/COBOL runtime system. For additional information, see “Preparing C Subprograms” on page H-10.

To compile:

```
cc -c src/mem.c
```

Note Some compilers may require that the ELF (Executable and Linking Format) be specified, as follows:

```
cc -b elf -c src/mem.c
```

To link:

```
cc -G -o mem.so mem.o
```

5. Create a COBOL program in a file named **mem.cbl** that contains the following source fragments:

```
01  Pointer-1 USAGE POINTER.
01  Pointer-2 USAGE POINTER.

CALL "malloc"  USING      4096 GIVING Pointer-1.
CALL "realloc" USING Pointer-1 8192 GIVING Pointer-2.
IF Pointer-2 NOT = NULL
   SET Pointer-1 TO Pointer-2
END-IF.
CALL "free"    USING Pointer-1.
```

   The COBOL code allocates a block of memory that is 4096 bytes long. After the **malloc** call, the base address component of Pointer-1 contains the address of the allocated memory block (or NULL if **malloc** was unable to allocate the memory). The offset component of Pointer-1 is zero and its size component is 4096. Next, the **realloc** call increases the size of the memory block to 8192 bytes (or possibly allocates a new block, copies the data, and frees the original block; also, a NULL may be returned if the request cannot be satisfied). Finally, the **free** call deallocates the 8192-byte block of memory (or the original 4096-byte block if the call to **realloc** fails).

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol mem
```

7. Run the application, specifying the name of the COBOL program and the name of the non-COBOL subprogram library.

   You may specify the name of the non-COBOL subprogram with the appropriate file extension. The following two commands illustrate how to specify a Windows DLL or a UNIX shared object (generally referred to as optional support modules). Since the COBOL program and the non-COBOL subprogram have the same root name (mem), it is necessary to specify the correct file extension.

   For Windows

```
runcobol mem -l mem.dll
```

   For UNIX

```
runcobol mem -1 mem.so
```

If the preceding examples had used different root names for the COBOL program and the non-COBOL subprogram, it would not be necessary to specify the file extension. For example, if the COBOL program were named “myprog”, then the following command could be used for either Windows or UNIX:

```
runcobol myprog -l mem
```

This example assumes that both the COBOL program and the non-COBOL subprogram are located in the current directory.

Example 5: Packing and Unpacking Structures

When a C function uses structures or unions as parameters, you must use an intermediate function that packs scalars into structure and union parameters. This example illustrates that process. No new attributes or attribute lists are presented.

1. Start with the function prototypes for the two standard C library functions, time and localtime:

```
time_t          time(time_t *timer);
struct tm *localtime(const time_t *timer);
```

   The return value for **localtime** is a C structure named **tm**, which is defined as:

```
struct tm      {int tm_sec;     //seconds           [0,59]
                int tm_min;     //minutes           [0,59]
                int tm_hour;    //hours             [0,23]
                int tm_mday;    //day of month      [1,31]
                int tm_mon;     //month             [0,11]
                int tm_year;    //years since 1900!
                int tm_wday;    //day of week       [0,6]
                int tm_yday;    //day of year       [0,365]
                int tm_isdst;   //daylight savings flag};
```

   Create a C function, **time_function**, in a file named **timefn.c** in the **src** directory that consists of the following lines:

```
#include <time.h>

time_function(short *sec, short *min, short *hour)
{
    time_t     time_of_day;
    struct tm *tmbuf;

    time_of_day = time(NULL);
    tmbuf = localtime(&time_of_day);

    *sec  = tmbuf->tm_sec;
    *min  = tmbuf->tm_min;
    *hour = tmbuf->tm_hour;
}
```

   This function calls **time** and **localtime** and extracts the structure members named tm_sec, tm_min, and tm_hour, into scalar output parameters named sec, min, and hour.

2. Create a template file named **mytime.tpl** in the **src** directory that consists of the following lines:

```
                time_function(
[[integer out]]             short *sec,
[[integer out]]             short *min,
[[integer out]]             short *hour);
```

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\mytime.tpl
```

   This command reads the input file from **src\mytime.tpl** and writes its output file to **src\mytime.c**. Any errors would be written to file **src\mytime.err**.

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice. There are two C files to compile:

   - The **time_function** function (created in step 1) in the file named **timefn.c**.
   - The file named **mytime.c** (created in step 3 by CodeBridge Builder when it processed the file named **mytime.tpl**, created in step 2).

   Use commands similar to the following:

   For Windows

```
cl -c –MD -Zp1 src\timefn.c
cl -c –MD -Zp1 src\mytime.c

link -nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:mytime.dll
     mytime.obj timefn.obj kernel32.lib user32.lib
```

For UNIX

A makefile is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object to be used as a support module with the RM/COBOL runtime system. For additional information, see "Preparing C Subprograms" on page H-10.

To compile:

```
cc -c src/mytime.c
cc -c src/timefn.c
```

Note Some compilers may require that the ELF (Executable and Linking Format) be specified, as follows:

```
cc -b elf -c src/mytime.c
cc -b elf -c src/timefn.c
```

To link:

```
cc -G -o mytime.so mytime.o timefn.o
```

5. Create a COBOL program in a file named **mytime.cbl** that contains the following source fragments:

```
01  GROUP-1.
    02  TM-SEC   PIC 9(2).
    02  TM-MIN   PIC 9(2).
    02  TM-HOUR  PIC 9(2).

CALL "time_function" USING TM-SEC TM-MIN TM-HOUR.
```

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol mytime
```

7. Run the application, specifying the name of the COBOL program and the name of the non-COBOL subprogram library.

   You may specify the name of the non-COBOL subprogram with the appropriate file extension. The following two commands illustrate how to specify a Windows DLL or a UNIX shared object (generally referred to as optional support modules). Since the COBOL program and the non-COBOL subprogram have the same root name (mytime), it is necessary to specify the correct file extension.

For Windows

```
runcobol mytime -l mytime.dll
```

For UNIX

```
runcobol mytime -1 mytime.so
```

If the preceding examples had used different root names for the COBOL program and the non-COBOL subprogram, it would not be necessary to specify the file extension. For example, if the COBOL program were named “myprog”, then the following command could be used for either Windows or UNIX:

```
runcobol myprog -l mytime
```

This example assumes that both the COBOL program and the non-COBOL subprogram are located in the current directory.

Example 6: Converting Buffered C Data

When an existing C API uses one C function to establish a buffer address and another C function to store data into the buffer, the preferred method of using the **out** direction attribute to modify COBOL data areas cannot be used (see "Modifying COBOL Data Areas" on page 2-29).

Note This example is fully elaborated only for Windows, where the ODBC API is readily available from Microsoft. However, the CodeBridge techniques illustrated are general in nature and may be instructive to developers creating templates for C subprograms on UNIX, including use of the ODBC API provided by other companies for some UNIX systems.

An example of this situation occurs in the Microsoft ODBC API. A buffer location is established with the function SQLBindCol, which binds a result set column to a storage location. Later, a call to the function SQLFetch obtains data from the result set and returns the data for each column previously bound to a storage location with the function SQLBindCol. The data obtained by the function SQLFetch is stored as C format data, not COBOL format data. For example, a string would be stored as a null-terminated C string. If a COBOL program is using CodeBridge to make the calls to the functions, SQLBindCol and SQLFetch, a method is needed to convert the C format data to COBOL format data. Such a conversion function can be written using CodeBridge and a minimal C function supplied by the developer.

This example illustrates a conversion routine that converts a C null-terminated string into a space-filled COBOL alphanumeric data item.

1. Start by writing a simple C function that copies one C string to another:

```
#include <string.h>

void cstring2text(char *pInput, char *pOutput)
{   (void)strcpy(pOutput, pInput);
}
```

2. Create a template file named **strcvt.tpl** in the **src** directory that consists of the following lines:

```
void cstring2text(
[[address]]                    char *pInput,
[[string out trailing_spaces]] char *pOutput);
```

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\strcvt.tpl
```

   This command reads the input file from **src\strcvt.tpl** and writes its output file to **src\strcvt.c**. Any errors would be written to file **src\strcvt.err**.

   CodeBridge Builder generates a C function from the template file. The generated C function will add trailing space characters to the output string argument because of the **trailing_spaces** base modifier attribute specified in the template file. All the work of the conversion is performed in the call to StringToCobol in the generated function (see page F-59 for a description of StringToCobol).

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice, using commands similar to the following:

```
cl -c –MD -Zp1 src\strcvt.c

link -nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:strcvt.dll
     strcvt.obj kernel32.lib user32.lib
```

5. Create a COBOL program in a file named **strcvt.cbl** that contains the following source fragments:

```
01 IN-STRING             PIC X(257).
01 OUT-STRING            PIC X(256).

CALL "cstring2text" USING IN-STRING OUT-STRING.
```

   In this example, it is assumed that the address of IN-STRING was passed to a C function, for example, the function SQLBindCol, and then subsequently a C function was called that used this address to store a string, for example, the function SQLFetch. See “Passing the Address of COBOL Data” on page 2-31 for an explanation of how the address of a data item is passed using CodeBridge. These fragments of the COBOL program are not illustrated here. In this example the data item named IN-STRING would contain a null-terminated C string and thus should not be used by the COBOL program other than in the call to the function that uses it as a buffer address and to the conversion function, cstring2text.

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol strcvt
```

7. Run the application with the following command line:

```
runcobol strcvt -l strcvt.dll
```

Example 7: Calling C++ Libraries from CodeBridge

The following example demonstrates how to resolve external references between the ways that C external names and C++ external names are represented.

The special techniques described in this example are necessary because the external function and variable names generated by C and C++ compilers do not match. C++ embeds type information in the external name that C cannot use. This type information is present even in C++ code that does not use C++ features. The linker, therefore, cannot resolve a call from C into C++ unless the C++ function or variable declaration explicitly specifies that the function or variable be made compatible with C.

To correct this situation, the C++ function definition in the C++ library must include the notation `extern "C"` in the definition. For example, modifying

```
int FunctionName(...)
```

to

```
extern "C" int FunctionName(...)
```

instructs the C++ compiler to generate a function name that is compatible with both C and C++.

In many instances, the CodeBridge developer will not have access to the source for libraries that are written in C++. In such cases, it is necessary to create intermediate or mapping functions that include the `extern "C"` notation.

Within this example, a naming convention is used. Entities that are a part of the C++ library have names that begin with libfunc or LibFunction, while entities that are related to the C++ intermediate functions that you write have names that begin with maplib or MapFunction. The normal C/C++ file extension name convention is followed throughout this example (that is, **.cpp** indicates a C++ file; **.c** indicates a C file).

This example, while very simplified, illustrates how you can use CodeBridge to call programs that are written in C++. Since the C++ programming language is not the same as C, some expertise in C++ on the developer's part will be required. In practice, the intermediate or mapping functions that you write will be "driver" functions that perform several steps. When dealing with C++ class libraries or methods, the intermediate program will have to deal with these C++ language constructs.

1. In this example, the following C++ source files represent the C++ library. The files named **libfunc.cpp** and **libfunc.h** represent components of the C++ library. The C++ library contains functions named LibFunction1 and LibFunction2.

   The file **libfunc.cpp** represents the source code that is used to build a C++ library and contains the following lines:

```
int LibFunction1()
{
  return(1);
}
int LibFunction2()
{
  return(2);
}
```

   The file **libfunc.h** makes function definitions available externally and contains the following lines:

```
int LibFunction1();
int LibFunction2();
```

   Create a C++ source file that will map the function from C++ names to C names. The file **maplib.cpp** contains the following lines:

```
#include "libfunc.h"
extern "C" int MapFunction1()
{
  return(LibFunction1());
}
extern "C" int MapFunction2()
{
 return(LibFunction2());
}
```

2. Create a template file named **maplibcb.tpl** that consists of the following lines:

```
[[integer out]] int MapFunction1();
[[integer out]] int MapFunction2();
```

3. Additionally, create a COBOL program in a file named **myprog.cbl** that calls the functions “MapFunction1” and MapFunction2”. This file would include the following lines:

```
CALL "MapFunction1" GIVING Result
CALL "MapFunction2" GIVING Result
```

4. Invoke the CodeBridge Builder by using the following command line:

```
cbridge maplibcb.tpl
```

5. Compile and link the non-COBOL subprogram library with the C and C++ compilers, using commands similar to the following:

For Windows

```
cl –c –MD –Zp1 src\maplibcb.c
cl –c –MD –Zp1 src\maplib.cpp
cl –c –MD –Zp1 src\libfunc.cpp

link –nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:MapLib.dll
     maplib.obj libfunc.obj maplibcb.obj
```

For UNIX

A makefile is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object to be used as a support module with the RM/COBOL runtime system. For additional information, see "Preparing C Subprograms" on page H-10.

To compile:

```
cc -c src/maplibcb.c
CC -c src/maplib.cpp
CC -c src/libfunc.cpp
```

Note Some compilers may require that the ELF (Executable and Linking Format) be specified, as follows:

```
cc -b elf -c src/maplibcb.c
CC -b elf -c src/maplib.cpp
CC -b elf -c src/libfunc.cpp
```

To link:

```
cc -G -o maplib.so maplibcb.o maplib.o libfunc.o
```

Note Uppercase CC is used to represent the name of the C++ compiler. On some systems, it may be CC (uppercase) while on others it may be cc (lowercase). For Gnu C++, the name is g++. Be sure to check your system documentation for the name used on your system.

6. Compile the COBOL program **myprog.cbl** that calls “MapFunction1” and “MapFunction2” by using the following command line:

   ```
   rmcobol myprog
   ```

7. Run the application, specifying the name of the COBOL program and the name of the non-COBOL subprogram library, with the following command line:

   ```
   runcobol myprog -l maplib
   ```

   This example assumes that both the COBOL program and the non-COBOL subprogram are located in the current directory.

Example 8: Using errno

This example demonstrates how to use the error base attribute, **errno**. The **errno** attribute supports obtaining the value of the external variable errno that was set by a call to a C library function. It allows return of the error information by editing the CodeBridge template instead of the generated code.

1. Start with the function prototype for the C standard library function, mkdir.

   For Windows

   ```
   int _mkdir( const char *dirname );
   ```

   For UNIX

   ```
   int mkdir (const char *filename, mode_t mode);
   ```

2. Create a template file named **mkdir.tpl** in the **src** directory that consists of the following lines:

   For Windows

   ```
   [[integer out]]                  int _mkdir(
     [[string in trailing_spaces]]    const char *DirName
     [[errno]]);
   ```

   For UNIX

   ```
   [[integer out]]                  int _mkdir(
     [[string in trailing_spaces]]    const char *DirName,
     [[integer in]]                   mode_t Mode
     [[errno]]);
   ```

   The **errno** error base attribute associates a COBOL argument with the value associated with the external C global variable errno. There is no corresponding parameter in the underlying C function parameter list.

   Note In this example, the **errno** error base attribute is placed after the last C parameter. This is a legal operation. The attribute could also have been placed anywhere any other attribute could have been placed.

3. Invoke the CodeBridge Builder by using the following command line:

   ```
   cbridge src\mkdir.tpl
   ```

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice, using commands similar to the following:

   For Windows

```
cl –c –MD –Zp1 src\mkdir.c

link   –nologo –machine:IX86 –section:.edata,RD –dll
       -subsystem:windows –out:mkdir.dll
       mkdir.obj kernel32.lib user32.lib
```

   For UNIX

```
cc –c src/mkdir.c

cc –G –o mkdir.so makdir.o
```

5. Create a COBOL program in a file named **mkdir.cbl** that contains the following source fragments:

   For Windows

```
01  Err-No          PIC S9(9).
01  File-Name       PIC X(64) Value "TempFile".
01  Return-Status   PIC S9(9).

CALL "_mkdir"
    USING File-Name Err-No
    GIVING Return-Status.
```

   For UNIX

```
01  Err-No          PIC S9(9).
01  File-Name       PIC X(64) Value "TempFile".
01  Mode            PIC S9(9) Value 1638.
01  Return-Status   PIC S9(9).

CALL "mkdir"
    USING File-Name Mode Err-No
    GIVING Return-Status.
```

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol src\mkdir
```

7. Run the application, specifying the name of the COBOL program and the name of the non-COBOL subprogram library:

   You may specify the name of the non-COBOL subprogram with the appropriate file extension. The following two commands illustrate how to specify a Windows DLL or a UNIX shared object (generally referred to as optional support modules). Since the COBOL program and the non-COBOL subprogram have the same root name (mkdir), it is necessary to specify the correct file extension.

   For Windows

```
runcobol src\mkdir.cob –l mkdir.dll
```

   For UNIX

```
runcobol src/mkdir.cob –l mkdir.so
```

Example 9: Using get_last_error

This example demonstrates how to use the **get_last_error** error base attribute. The **get_last_error** attribute supports obtaining the value returned by the Windows API function GetLastError called immediately after another Windows API function has been called.

Note The following discussion applies to using this attribute on the Windows platform only. Some Windows APIs have been ported to UNIX. In such cases, it may be appropriate to use the **get_last_error** attribute on UNIX. (CodeBridge Builder does support the get_last_error attribute on UNIX.) However, if the SetLastError and GetLastError functions are not available, the generated program will probably not compile and would certainly not link without errors.

1. Start with the function prototype for the Windows API function, CreateDirectory.

```
WINBASEAPI BOOL WINAPI CreateDirectory(LPCTSTR DirName,
    LPSECURITY_ATTRIBUTES SecAttr);
```

2. Create a template file named **Dir.tpl** in the **src** directory that consists of the following lines:

```
#include <windows.h>

[# replace_type(LPCTSTR; char *)
   replace_type(LPSECURITY_ATTRIBUTES; void *)
   convention(WINBASEAPI)
   convention(WINAPI) #]

[[integer out]] WINBASEAPI BOOL WINAPI   CreateDirectory(
[[string in trailing_spaces]]     LPCTSTR DirName,
[[string in trailing_spaces value_if_omitted(NULL)]]
                                  LPSECURITY_ATTRIBUTES SecAttr
[[get_last_error]]);
```

   The **get_last_error** error descriptor attribute associates a COBOL argument with the value associated with the GetLastError Windows function. There is no corresponding parameter in the underlying C function parameter list.

   Note In this example, the **get_last_error** attribute is placed after the last C parameter. This is a legal operation. The attribute could also have been placed anywhere any other attribute could have been placed.

3. Invoke the CodeBridge Builder by using the following command line:

```
cbridge src\Dir.tpl
```

4. Compile and link the non-COBOL subprogram library with the C compiler of your choice, using commands similar to the following:

```
cl –c –MD –Zp1 src\Dir.c

link    –nologo –machine:IX86 –section:.edata,RD –dll
        -subsystem:windows –out:Dir.dll
        Dir.obj kernel32.lib user32.lib
```

5. Create a COBOL program in a file named **Dir.cbl** in the **src** directory that contains the following source fragments:

```
01  Last-Error      PIC 9(9).
01  File-Name       PIC X(64) Value "TempFile".
01  Return-Status   PIC S9(9).

CALL "CreateDirectory"
    USING File-Name Last-Error
    GIVING Return-Status.
```

6. Compile the COBOL program with the RM/COBOL compiler by using the following command line:

```
rmcobol src\Dir
```

7. Run the application with the following command line:

```
runcobol src\Dir.cob –l Dir.dll
```

Appendix C: Useful C Information

To develop applications using CodeBridge, it is necessary to have a fundamental understanding of certain C concepts as well as the ability to use a C compiler and linker. The information provided in this appendix is intended to serve as a starting point for those developers who may not be proficient with C programming and who wish to call existing C function libraries without writing any additional C code. This material should not be viewed as a formal or complete definition of the language. The ideas and concepts presented here are in an informal format. The developer is encouraged to acquire additional C reference information, as necessary.

The topics presented here include:

- Understanding C language concepts (see the following topic)
- Compiling and linking C functions (see page C-5)

Understanding C Language Concepts

In order to construct a template file, you must understand the concept of a C function prototype. The template file is based on a "marked-up" C function prototype. Conceptually, a C function prototype is similar to a COBOL LINKAGE SECTION. While the LINKAGE SECTION describes the interface to a COBOL subprogram, a function prototype describes the interface to a C function.

When using C, it is the preferred practice to use header files to contain the function prototypes (along with other information that is needed to describe the interface to a function). Header files are similar to copy files in COBOL. Providers of C function libraries will normally provide one or more header files to describe the interface to their libraries. Typically, a header filename will have a suffix of **.h**. For example, a provider of a statistics package may provide a header file named **statistics.h**. Header files are included in the source to be compiled with the **#include** C preprocessor directive and are thus sometimes referred to as include files.

Before discussing function prototypes in more detail (see page C-4), let's explain several concepts that are integral to the construction of function prototypes. These topics include case sensitivity, data types, data declarations, type definitions and macros, and calling conventions.

Case Sensitivity

The COBOL programming language is mostly case-insensitive. With a few exceptions (such as non-numeric literals), the uppercase and lowercase representations of a given letter are treated as equivalent. On the other hand, the C programming language is predominately case-sensitive. The attribute keywords used in the template file are also case-sensitive. This means that the uppercase and lowercase representations of a given letter are not equivalent.

For example, the following names are treated as separate entities by C, but treated as the same entity by COBOL: name, Name, and NAME.

Data Types

C includes predefined data types that may be categorized as integer, floating-point, pointer, and void.

Integer data types include **char**, **short**, **int** and **long**. These data types may be prefixed with the keywords **signed** or **unsigned**. Normally, integer types default to **signed**. As a shorthand notation, when **signed** or **unsigned** appear without the corresponding integer data type, then **int** is implied (that is, **unsigned** is the same as **unsigned int**).

C also includes the floating-point data types **float** and **double**. Floating-point is the computer representation of scientific notation. It allows numbers with a large scale or small scale to be represented with an approximate value. For the IEEE representation of floating-point, the **float** type is normally limited to about 6 or 7 digits of precision with an exponent (scale) of –38 to +38. Also, the **double** type is normally limited to about 15 or 16 digits of precision with an exponent (scale) of –308 to +308.

A pointer data type contains the address of a typed data item and is represented by the asterisk character (*****) in the data declaration or type definition (these terms are described in the following sections).

The void data type, **void**, is used to represent untyped or sometimes omitted data.

Note that other keywords, such as **far** and **near** also exist, although their meaning is mostly historical. Depending on the compiler, one or two underscore characters may precede some keywords (**_far** or **__far** instead of **far**).

Data Declarations

A data declaration associates data type information with the name of a variable. For example:

```
int P1
```

declares a variable named P1 with a type of **int**. Additional examples are shown in the following table:

Declaration	Variable Name	Type
`unsigned short P2;`	P2	**unsigned short**
`float P3;`	P3	**float**
`int * P4;`	P4	pointer to an **int**
`char P5[30];`	P5	array of **char** (30 elements in the array)
`void * P6;`	P6	pointer to a **void** (that is, a generic pointer)

When an array is passed to a C function, the address of (pointer to) the array is used. In a C function prototype, a pointer reference and an array reference are equivalent. That is, char P5[30] is treated the same as char *P5 (with the exception that the compiler can do some compile time range checking if the number of elements in the array is explicitly declared).

Type Definitions and Macros

In addition to the standard data types described previously, you can define additional types that are based on combinations of existing types. Two techniques are used: type definitions (**typedef**) and macros.

A **typedef** defines a new data type. Consider the following examples:

```
typedef int INT;
typedef unsigned char UCHAR;
typedef char * CHARPTR;
```

The first definition defines INT to be equivalent to **int**. That is, the two definitions of INT and **int** are identical. The second definition defines UCHAR to be equivalent to **unsigned char**. The third definition defines CHARPTR to be equivalent to **char *** (a pointer to a char).

A **typedef** “hides” the underlying data type so that programs may be paramaterized against portability problems. Type definitions also provide better documentation for a program.

Although a macro is similar to a **typedef**, there are some important, yet subtle, differences. The first two previous examples may be defined as macros with the **#define** C preprocessor directive, as follows:

```
#define INT int
#define UCHAR unsigned char
```

Macros are implemented as part of the C compiler preprocessor. If INT is defined in a macro, the compiler will never see INT as a data type; it will already have been replaced with int.

Additionally, macros provide a powerful text replacement feature that can be used for more than type redefinition. Macros may contain parameters and can be used to implement inline functions. For example:

```
#define MAX(A,B) (A+B)/2 + abs(A-B)/2
```

Macros are presented here to familiarize you with concepts that might occur in a header file. Since complex macros tend to be fragile, it is recommended that the modification of these macros be done with care.

Calling Conventions

A calling convention defines additional type information. It directs how the compiler generates function-calling sequences and is an optional part of a function prototype. Examples include __cdecl (or RM_CDECL when writing code for both Windows and UNIX), __stdcall, and __pascal. Often a calling convention is hidden with a type definition or a macro. For example, the following macro definition defines the macro, SQL_API to be the __stdcall calling convention:

```
#define SQL_API  __stdcall
```

Function Prototypes

A function prototype may contain or refer to any of the concepts that have been previously presented (data types, data declarations, type definitions and macros, and calling conventions).

A function prototype consists of the function name and a list of parameter names. The name of the function and the name of each parameter are prefaced with type information to form a data declaration. For example:

```
double RM_CDECL pow(double X, double Y);
```

In this example, the type of the function is double, which indicates that the function returns a value of type double. The parameters are also of type double. Notice that the calling convention RM_CDECL is included with the function type information.

An older style of prototype may be encountered. In this case, the function prototype omits the parameter names since they are only placeholders. The prototype for the function presented above may appear as follows (depending on platform and compiler):

```
double pow(double, double);
```

Placeholder names must be provided in the template file that is based on one of these older style prototypes. Any unique (to the function) names will do. For example:

```
[[float out rounded]] double pow(
[[float in rounded]]             double X,
[[float in rounded]]             double Y);
```

Compiling and Linking C Functions

Throughout the CodeBridge manual, examples of compiling and linking are presented. The syntax of the Windows examples uses Microsoft's compiler and linker conventions to generate 32-bit Windows dynamic link libraries (DLLs).. The syntax of the UNIX examples uses conventions that are common to many compilers and linkers on UNIX to generate shared objects.

Note A makefile is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object to be used as a support module with the RM/COBOL runtime system. For additional information, see "Preparing C Subprograms" on page H-10.

This section also includes an example of how to generate multiple template files.

Compiling on Windows

The following illustrates an example of invoking Microsoft's Visual C++ compiler to generate Windows object files:

```
cl -c –MD -Zp1 src\trig.c
```

where:

cl indicates the name of the compiler.

-c suppresses the implicit call to LINK that normally occurs.

-MD selects the Multithread and DLL options. The developer may choose -MDd in order to select the debugging option also.

-Zp1 specifies structure member alignment of 1 byte.

Note A structure is the C equivalent of a COBOL group. The -Zp1 option is recommended because the ARGUMENT_ENTRY structure passed from the RM/COBOL runtime system is built using the -Zp1 option.

src\trig.c indicates the name of the C source program to be compiled.

Note This example uses the hyphen (-) character to denote compiler options. Microsoft's Visual C++ compiler also allows a forward slash (/) character to be used (for example, /c instead of –c).

Compiling on UNIX

The following illustrates an example of producing object files on UNIX:

```
cc -c src/trig.c
```

where:

cc indicates the name of the compiler/linker.

-c suppresses the linking stage and does not produce an executable file.

src/trig.c indicates the name of the C source program to be compiled.

Linking on Windows

The following shows an example involving the Microsoft linker to generate a Windows DLL:

```
link -nologo -machine:IX86 -section:.edata,RD –dll
     -subsystem:windows -out:trig.dll trig.obj
```

where:

link indicates the name of the linker.

-nologo suppresses the startup banner.

-machine:IX86 specifies the target platform. While this option is not required, it is good practice to include it. (It also eliminates a warning message.)

-section:.edata,RD specifies Section Attributes, which force the linker to include the edata section in the generated DLL. The RM/COBOL runtime system uses this information to load the DLL.

-dll builds a DLL as the main output file.

-subsystem:windows specifies the subsystem being supported.

-out:trig.dll names the output file.

trig.obj specifies the name of the object file that is to be included in the link.

Note In addition to naming the object file(s) that are to be included, the necessary libraries also should be included. The names of the libraries are normally provided by the provider of the library functions or by the C compiler. The default link libraries for Win32 DLLs include:

- kernel32.lib
- user32.lib
- gdi32.lib
- winspool.lib
- comdlg32.lib
- advapi32.lib
- shell32.lib
- ole32.lib
- oleaut32.lib
- uuid.lib
- odbc32.lib
- odbccp32.lib

Note This example uses the hyphen (-)character to denote compiler options. Microsoft's linker also allows a forward slash (/) character to be used.

Linking on UNIX

The following illustrates an example of linking a shared object on UNIX:

```
cc -G -o trig.so trig.o -lm
```

where:

cc indicates the name of the compiler/linker.

-G produces a shared object.

-o trig.so names the output file.

trig.o specifies the name of the object file that is to be included in the link.

-**lm** indicates that the math library is to be included in the link.

Multiple Template Files

The normal practice is to generate only one template file for each non-COBOL subprogram library that is being constructed. However, some developers may choose to generate more than one template file.

For Windows platforms, the source generated by CodeBridge Builder contains a definition for DllMain. If CodeBridge Builder generates multiple files, then errors in linking the DLL will occur because of multiple definitions. This can be resolved by defining the symbol RM_NO_DLL_MAIN for all but one of the compilations of generated source files.

For example:

```
cl -l -MD -Zp1                  src\cbfunc1.c
cl -l -MD -Zp1 -DRM_NO_DLL_MAIN src\cbfunc2.c
cl -l -MD -Zp1 -DRM_NO_DLL_MAIN src\cbfunc3.c
```

Appendix D: Global Attributes

This appendix provides detailed descriptions of the attributes used in a global attribute list in a template file. See Chapter 2, *Concepts*, for more information about the basic components of a template file. The attributes used in a parameter attribute list are discussed in Appendix E, *Parameter Attributes*. More information about C language concepts and terms may be found in Appendix C, *Useful C Information*.

Note As you read through this manual, keep in mind that the term "parameter attribute" is a shorthand notation for an attribute that occurs in a parameter attribute list. Likewise, "global attribute" indicates that the attribute can be found in a global attribute list.

Overview

A global attribute list provides information about one or more C function prototypes that is not specific to any given parameter. This information also could be used to modify the default behavior of CodeBridge Builder.

A global attribute takes effect from the point at which it occurs in a template file and remains in effect until another global attribute in that template file alters those settings. There are five global attributes: **banner**, **convention**, **diagnostic**, **load_message**, and **replace_type**.

Attributes are case-sensitive and must be entered as shown.

Note The discussions and examples of the global attributes, **replace_type** and **convention**, use SQL_API and SQLPOINTER, which are a macro and data type, respectively, defined in the Microsoft Visual C++ header file, **sqltypes.h**. Their definitions are:

```
#define SQL_API __stdcall

typedef void * SQLPOINTER
```

SQL_API is a calling convention macro defined by the C preprocessor directive, **#define**. SQLPOINTER is a data type defined by a C type definition (that is, a **typedef** statement).

banner Attribute

Use the **banner** global attribute to display a text string when a non-COBOL subprogram built with CodeBridge is loaded by the RM/COBOL runtime system.

The format of the **banner** global attribute is as follows:

```
[# banner(value) #]
```

where *value* is a character string. For example:

```
[# banner("Copyright (c) 2000, by me.") #]
```

Such banners are displayed only on UNIX systems when the K Option of the RM/COBOL Runtime Command (**runcobol**) is not specified or configured. For example:

```
runcobol myprog -l ./mylib.so
```

This causes a message similar to the following to be displayed:

```
Copyright (c) 2000, by me.
```

No banner message is produced by the RM/COBOL for Windows runtime.

convention Attribute

Use the **convention** attribute to declare C calling conventions (for example, SQL_API). Calling conventions cannot be placed in the CodeBridge-generated declarations of variables; however, they must be preserved in the external function prototype that is used to call the C function.

The format of the **convention** global attribute is as follows:

```
[# convention(name) #]
```

where *name* is the name of a call convention.

SQL_API can be resolved as follows:

```
[# convention(SQL_API) #]
```

SQL_API is removed from variable declarations, but is preserved as part of the external function prototype.

diagnostic Attribute

Use the **diagnostic** attribute to control error reporting.

The format of the **diagnostic** global attribute is as follows:

```
[# diagnostic(value) #]
```

where *value* may be one of the following:

- **silent.** Use the silent value to instruct CodeBridge not to display diagnostic messages.
- **verbose.** Use the verbose value to instruct CodeBridge to display diagnostic messages even if the **silent** base modifier attribute (described on page E-4) is set for an individual parameter attribute list.
- **normal.** Use the normal value to instruct CodeBridge to display diagnostic messages unless the **silent** base modifier attribute (described on page E-4) is specified for an individual parameter attribute list.

load_message Attribute

Use the **load_message** attribute to display a text string when a non-COBOL subprogram built with CodeBridge is loaded by the RM/COBOL runtime system.

The format of the **load_message** global attribute is as follows:

```
[# load_message(value) #]
```

where *value* is a character string. For example:

```
[# load_message("My math package - Version 1.13") #]
```

Load messages are displayed only on UNIX systems when the V Option of the RM/COBOL Runtime Command (**runcobol**) is specified or configured. For example:

```
runcobol myprog -v -l ./mylib.so
```

This causes a message similar to the following to be displayed:

```
RM/COBOL: Dynamic library loaded - ./mylib.so - My math package - Version 1.13
```

No load message is produced by the RM/COBOL for Windows runtime.

replace_type Attribute

The CodeBridge Builder program does not resolve C data types. Frequently, necessary data type information may be hidden in a macro or a type definition construct (as shown in the definitions above). Specifically, CodeBridge must know whether a data item is a pointer data type. It is necessary, therefore, for the template file to resolve some type definitions for CodeBridge.

Use the **replace_type** global attribute to allow CodeBridge to resolve pointer data declarations that hide the C unary pointer operator (*) within the data type name (for example, SQLPOINTER).

You may choose to use the **replace_type** attribute as a form of self-documentation to expand any defined data type, even if the expansion does not reveal any levels of indirection.

The format of the **replace_type** global attribute is as follows:

```
[# replace_type(name;value) #]
```

where *value* is the character string that replaces the data type specified by *name*.

The SQLPOINTER data type can be resolved as follows:

```
[# replace_type(SQLPOINTER; void *) #]
```

The user-supplied entry for *name* must be a single token. The user-supplied entry for *value* may be any string of characters. The following are all equivalent:

```
[# replace_type(SQLPOINTER;void*) #]

[# replace_type(SQLPOINTER;void *) #]

[# replace_type(SQLPOINTER; void * ) #]
```

Appendix E: Parameter Attributes

This appendix provides detailed descriptions of the attributes used in a parameter attribute list in a template file. See Chapter 2, *Concepts*, for more information about the basic components of a template file. The attributes used in a global attribute list are discussed in Appendix D, *Global Attributes*. More information about C language concepts and terms may be found in Appendix C, *Useful C Information*.

Note As you read through this manual, keep in mind that the term "parameter attribute" is a shorthand notation for an attribute that occurs in a parameter attribute list. Likewise, "global attribute" indicates that the attribute can be found in a global attribute list.

Overview

The parameter attributes are organized into the following three groups:

- Argument number
- Direction
- Base and base modifier

Each group is described in the following sections. An alphabetical summary of all available parameter attributes is shown in Table E-2 beginning on page E-24.

Attributes are case-sensitive and must be entered as shown.

Argument Number Attributes

The two argument number parameter attributes, **arg_num***(value)* and **ret_val**, specify explicitly the COBOL argument number. This invokes the explicit method of associating C parameters with COBOL arguments rather than using the default automatic association method.

In the **arg_num**(*value*) argument number attribute, *value* specifies the argument number as 1 for the first argument in the USING phrase, 2 for the second argument in the USING phrase, and so forth. The value must be specified as an integer constant; a macro or constant expression may not be specified here.

The **ret_val** argument number attribute specifies the argument in the GIVING (RETURNING) phrase.

For more information, see "Associating C Parameters with COBOL Arguments" on page 2-21.

Direction Attributes

The direction attributes are **in** and **out**. The **in** direction attribute specifies an input parameter to the C function. The **out** direction attribute specifies an output parameter from the C function.

Both the **in** and **out** direction attributes may be specified in a parameter attribute list. Within a parameter attribute list, you may present the attributes in any order. For example, `[[integer in]]` is the same as `[[in integer]]`. When a parameter is used for both input and output, both the **in** and **out** direction attributes are specified in either order.

The direction attributes may be used to protect the calling COBOL program from unintended modification of data. For example, when the **out** direction attribute is not used, then the data in the C parameter is not converted to COBOL format, and the data is not placed in the address space of the COBOL program.

For a given parameter, if none of its attribute lists contain the **in** direction attribute, an uninitialized value may be passed to the function. No more than one attribute list (for any given parameter) should be used for input; however, several output attribute lists may be assigned to the same parameter.

Some base attributes imply a direction and thus do not allow either of the direction attributes. The error base attributes, **errno** and **get_last_error** (see page E-20), imply the **out** direction attribute. The descriptor base attributes (see page E-17), two of the pointer base attributes, **pointer_address** and **pointer_length** (see page E-15), and the string length base attributes (see page E-14) imply the **in** direction attribute.

Base and Base Modifier Attributes

Base attributes may be categorized as follows:

- **Numeric.** Numeric base attributes (see page E-5) are used when passing COBOL numeric arguments to the C function.
- **String.** The s**tring** base attribute (see page E-11) is used when passing COBOL non-numeric arguments to the C function.
- **String Length**. String length base attributes (see page E-14) are used when passing the length of a string or numeric string parameter as a separate C parameter.
- **Pointer.** Pointer base attributes (see page E-15) are used when passing COBOL pointer data items to the C function.
- **Descriptor.** Descriptor base attributes (see page E-17) are used when passing a component of a COBOL data descriptor, the argument count, the COBOL initial state flag, or the Windows handle to the C function.
- **Error.** Error base attributes (see page E-20) are used to retrieve error information from a C library or Windows API function that is returned separately from the calling C function.

Note 1 The **numeric_string** base attribute (see page E-6) is unique because it associates a C string parameter, rather than a C numeric parameter, with a COBOL numeric argument. This base attribute refers to a COBOL numeric argument (whose USAGE clause specifies DISPLAY, PACKED-DECIMAL, BINARY, and so forth) and is, therefore, a numeric base attribute. However, the argument value is represented as an ASCII character string in the C function.

Note 2 The **general_string** base attribute (see page E-13) converts numeric and non-numeric arguments to null-terminated arrays of characters. If the COBOL argument is numeric, the conversion behaves as if **numeric_string** had been specified as the base attribute. If the COBOL argument is non-numeric, the conversion behaves as if **string** had been specified as the base attribute.

Base attributes can be supplemented with additional information by specifying base modifier attributes. While some base modifier attributes are common to several categories of base attributes, as discussed in the following section, others are specific to a base attribute category. The latter are described in each base attribute category section to which they apply.

Base Modifiers Common to Base Attributes

Two base modifier attributes, **silent** and **alias**(*name*), are common to several categories of base attributes:

- **silent.** The **silent** base modifier is used with any base attribute to prevent CodeBridge from displaying diagnostic messages during CodeBridge Library calls generated for that attribute list. The global attribute, **diagnostic**(*value*), may be used to alter default behavior for every CodeBridge Library call (see page D-3).
- **alias**(*name*)**.** The **alias**(*name*) base modifier is used in any parameter attribute list that refers to the function return value (that is, it should not be used with function parameters). The **alias**(*name*) base modifier may be used in a parameter attribute list with other attributes, or it may be the only attribute in an attribute list.

  If it is the only attribute in a parameter attribute list, no value will be returned to the calling COBOL program.

  Normally, the CodeBridge Builder generates its interface function name from the C function name. The **alias**(*name*) base modifier attribute makes it possible for the COBOL program to call the C function using a different name. The following example shows how to implement two functions, INTEGER_PART and FRACTION_PART, from the standard C library function, modf.

  Use the following template file to construct an interface to the standard C library function, modf. This function returns the integer part of A in IntPart and the fraction part of A as the return value.

```
[[float out]]             double modf(
[[float in]]                         double  A,
[[float in out]]                     double *IntPart);
```

  Use the following template file to return only the integer part:

```
                          double modf(
[[float in arg_num(1)]]              double  A,
[[float out   ret_val]]              double *IntPart);
```

A problem with this example is that the COBOL program must call **modf** instead of **integer_part**. To resolve this problem, use the **alias**(*name*) base modifier attribute as follows:

```
[[alias(integer_part)]]   double modf(
[[float in arg_num(1)]]              double  A,
[[float out   ret_val]]              double *IntPart);
```

A similar function, called **fraction_part**, uses the return value of the **modf** function, as follows:

```
[[alias(fraction_part)
  float out]]             double modf(
[[float in]]                         double  A,
                                     double *IntPart);
```

Numeric Base Attributes

Three numeric base attributes are used to convert between COBOL numeric data items and C data items:

- **integer.** Use the **integer** base attribute with C integer data types (such as **char**, **short**, **int**, and **long**).

  On input, the COBOL numeric argument is converted to an integer C parameter. If the argument value contains a fractional component after application of the **scaled**(*value*) base modifier attribute, if specified, it will be truncated (or rounded, if the **rounded** base modifier is used). On output, the C parameter is converted to a COBOL numeric argument. If the argument is described using P-scaling (see page 2-32), truncation may occur (or rounding, if the **rounded** base modifier is used).

- **float.** Use the **float** base attribute with C floating-point data types (**float** and **double**).

  On input, the COBOL numeric argument is converted to a floating-point C parameter. If the argument contains more trailing digits than are supported by the floating-point representation, it is truncated (or rounded if the **rounded** base modifier is used). On output, the C parameter is converted to a COBOL numeric argument. Truncation may occur (or rounding, if the **rounded** base modifier is used).

- **numeric_string**. Use the **numeric_string** base attribute to pass COBOL numeric arguments to null-terminated C string parameters, called a numeric string in this document.

  A numeric string is created in a dynamically allocated buffer. By default, the buffer length is four more than the digit length of the COBOL argument. This ensures enough room in the buffer to contain the numeric value, the decimal point character, one or two sign characters, and a trailing null character. This default length may be overridden using the **size**(*value*) base modifier attribute.

Note Numeric base attributes may be used with arrays. For more information, see "Numeric Arrays" on page 2-33.

Numeric String Formatting and Conversion Rules

A numeric string parameter is a parameter for which either the **numeric_string** or the **general_string** base attribute has been specified and for which the COBOL argument is numeric. For use with a C function, a numeric string is formatted according to the following rules:

1. The string is composed of two parts: an optional sign and a numeric value.
2. The sign may be a leading sign (occurring before the numeric value) or a trailing sign (occurring after the numeric value). A leading sign may be a single character (either "+" or "-"). A trailing sign may be either one character (either "+" or "-") or two characters (the debit symbol "DB" or the credit symbol "CR").

   Note On input conversion before calling the C function, the sign representation will be placed in the string according to the leading or trailing sign base modifiers that are selected. On output conversion (after returning from the C function), any supported sign representation is allowed. See "leading or trailing signs" on page E-10 for the supported sign representations.
3. The numeric value is represented as a string of numeric characters ('0' through '9') with an embedded decimal point character point, as needed.

   Note On input conversion, if the data item contains an integer value, the resultant numeric string does not contain a decimal point character or trailing zero characters. Also, on input conversion, if the data item contains only a fraction value (the absolute value of the data item is non-zero and less than 1), the resultant numeric string will contain a leading zero character followed by a decimal point character.
4. Space characters may occur before and after both the numeric value and the sign. They are ignored.

Note On input conversion to the C function, CodeBridge will not place any space characters in a numeric string. On output conversion from the C function, CodeBridge will tolerate embedded spaces.

Some examples of numeric strings are:

```
" 1 "

" - 1 "

" 2.34 CR"

"0"
```

Base Modifiers that Apply to Numeric Base Attributes

Numeric base attributes can be supplemented with additional information by the base modifier attributes that are listed below. Some of the base modifier attributes apply to all numeric base attributes, while others apply only to a particular numeric base attribute.

The following base modifier attributes may be used with any numeric base attributes:

- **alias**(*name*)**.** For a description of this base modifier, see page E-4. Note that the **alias** base modifier attribute is only allowed when the parameter attribute list precedes the function name.
- **assert_digits**(*min;max*). Use this base modifier attribute to verify that the digit length of the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[integer out assert_digits(5;5)]]` indicates that the COBOL data item must contain exactly five digits.

  The use of P-scaling in the COBOL program will increase the digit length by the number of P symbols specified in the PICTURE character-string. For example, all of the PICTURE character-strings 9(8), 9(5)P(3), and VP(3)9(5) describe a data item with a digit length of eight for CodeBridge.
- **assert_digits_left**(*min;max*)**.** Use this base modifier attribute to verify that the number of digits to the left of the decimal point in the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[float assert_digits_left(5;~0)]]` indicates that the COBOL data item must contain five or more digits to the left of the decimal point, or equivalently, no less than five digits before the decimal point.

Note The C construct, **~0**, denotes a pattern of all ones and represents the largest positive value that can be stored in a data item. This usage is preferable to other choices such as 0xffff (which requires knowing the number of f's to write) and –1 (which is not allowed C for unsigned data types).

The use of P-scaling in the COBOL program will increase the number of digits to the left of the decimal point by the number of P symbols specified in the PICTURE character-string that occur to the left of the decimal point. For example, both of the PICTURE character-strings 9(8) and 9(5)P(3) describe a data item with eight digits to the left of the decimal point for CodeBridge.

- **assert_digits_right**(*min;max*)**.** Use this base modifier attribute to verify that the number of digits to the right of the decimal point in the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[float assert_digits_right(0;2)]]` indicates that the COBOL data item must contain no more than two digits after the decimal point.

  The use of P-scaling in the COBOL program will increase the number of digits to the right of the decimal point by the number of P symbols specified in the PICTURE character-string that occur to the right of the decimal point. For example, both of the PICTURE character-strings V9(8) and VP(3)9(5) describe a data item with eight digits to the right of the decimal point for CodeBridge.

- **assert_length**(*min;max*)**.** Use this base modifier attribute to verify that the actual length of the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[integer out assert_length(10;~0)]]` indicates that the COBOL data item must contain at least ten characters.

  Note The C construct, **~0**, denotes a pattern of all ones and represents the largest positive value that can be stored in a data item. This usage is preferable to other choices such as 0xffff (which requires knowing the number of f's to write) and –1 (which is not allowed C for unsigned data types).

- **assert_signed.** Use this base modifier attribute to verify that the passed COBOL argument contains a sign.

- **assert_unsigned.** Use this base modifier attribute to verify that the passed COBOL argument does not contain a sign.

- **no_null_pointer.** The calling COBOL program may pass a pointer with a null value as an argument either by specifying the figurative constant NULL (NULLS) or by specifying a COBOL pointer argument that has been set to NULL (NULLS). In this case, CodeBridge would normally pass a null pointer as a parameter to the C function. If the **no_null_pointer** base modifier attribute is used, an error condition will be generated instead.

- **no_size_error.** During conversion (either COBOL to C or C to COBOL), it is possible that leading digits will be lost. If this occurs, the normal behavior is to generate an error condition. If the **no_size_error** base modifier attribute is used, the error condition will be ignored.
- **occurs**(*value*)**.** Arrays of COBOL numeric arguments may be passed to a C function. Use the **occurs**(*value*) base modifier attribute to specify the array size. If the C function prototype specifies the array size, it is not necessary to use the **occurs**(*value*) base modifier attribute unless you need to override the value specified in the function prototype.
- **optional.** The calling COBOL program may omit the associated argument (see "Managing Omitted Arguments" beginning on page 2-17 for more information on omitted arguments and this attribute), in which case CodeBridge would normally generate an error condition. If the **optional** base modifier attribute is used, then a default value is generated and passed to the C function. The default value associated with an **integer** or **float** base attribute is a numeric zero. The default value associated with a **general_string** or **numeric_string** base attribute is an empty string (the first character of the string is a null character). If a value other than the CodeBridge supplied default value is desired, see the **value_if_omitted**(*value*) base modifier attribute description.

  Note The current implementation of CodeBridge Builder only allows input optional parameters. Output parameters are required by default.
- **repeat**(*value*)**.** Use this base modifier attribute with the C parameter before the ellipsis when a variable number of C parameters is used. *value* indicates the maximum number of additional C parameters.
- **rounded.** Use this base modifier attribute to cause rounding in those cases where truncation would normally occur (on either input or output). Rounding is performed using COBOL rounding rules.
- **silent**. For a description of this base modifier, see page E-4.
- **value_if_omitted**(*value*)**.** Use this base modifier attribute to specify a value to be used when the calling COBOL program omits the associated argument (see "Managing Omitted Arguments" beginning on page 2-17 for more information on omitted arguments and this attribute). When this attribute is used, it is not necessary to also use the **optional** base modifier attribute. An **integer** attribute list must specify an integer value (for example, `value_if_omitted(3)`); a **float** attribute list must specify a floating-point value (for example, `value_if_omitted(3.0)`); and a **numeric_string** attribute list must specify a string value (for example, `value_if_omitted("3.0")`).

In addition to the base modifier attributes that apply to all numeric base attributes, the following modifiers are specific to the **integer** base attribute:

- **integer_only.** Use this base modifier attribute to verify that the passed COBOL argument represents an integer value (that is, no digits are allowed to the right of the decimal point). This attribute is equivalent to the **assert_digits_right(0;0)** base modifier attribute specification.
- **scaled**(*value*)**.** Use this base modifier attribute to scale integer values during the conversion process. On input, the COBOL argument is multiplied by 10^{value}. On output, the C parameter is divided by 10^{value}.

  For example, if the attribute list is `[[integer in out scaled(2)]]` and the COBOL program supplied a value of 1.53, the C function would receive a value of 153. If the C function changed the value to 4, the COBOL program would receive .04 back.
- **unsigned.** Use this base modifier attribute to force CodeBridge to treat the C parameter as unsigned. The default is to treat C parameters as signed.

In addition to the modifiers that apply to all numeric base attributes, the following modifiers are specific to the **numeric_string** base attribute:

- **size**(*value*)**.** Use this base modifier attribute with the **numeric_string** base attribute to specify a *value* that overrides the default length when the conversion string buffer is dynamically allocated.
- **leading or trailing signs.** One of the following leading or trailing sign base modifier attributes may be used with for the **numeric_string** base attribute. The default base modifier attribute is **leading_sign**.

Attribute	Sign if positive	Sign if negative
leading_sign	"+"	"-"
leading_minus	none	"-"
trailing_sign	"+"	"-"
trailing_minus	none	"-"
trailing_credit	none	"CR"
trailing_debit	none	"DB"

string Base Attribute

C strings are a null-terminated array of characters. Although there are many standard C library functions that deal with C strings, there is no corresponding COBOL data type. The **string** base attribute is used to convert between COBOL non-numeric arguments and null-terminated C string parameters.

On input, data is copied to a dynamically allocated buffer and a trailing null character is added. On output, data is copied from the buffer and the trailing null character is removed. By default, the data buffer is one byte larger that the length of the COBOL argument so that there is room for the trailing null character. This default may be overridden using the **size**(*value*) base modifier attribute.

Note 1 On Windows platforms, CodeBridge allocates the intermediate buffer using the SysAllocStringByteLen function. This places additional overhead information before the start of the string. The SysStringByteLen function may be used to obtain the length of the buffer. Use the standard C library function, strlen, to retrieve the length of the string in the buffer.

Note 2 A **string** base attribute may be used with arrays. For more information, see "String Arrays" on page 2-34.

Base Modifiers that Apply to the String Base Attribute

One leading character and one trailing character base modifier attribute may be specified for each parameter. On input, leading and/or trailing characters are removed as specified. On output, trailing characters (if selected) are added to left-justified data items, while leading characters (if selected) are added to right-justified data items.

The **string** base attribute can be supplemented with additional information by the base modifier attributes that are listed below.

The following base modifier attributes may be used with the **string** base attribute:

- **alias**(*name*)**.** For a description of this base modifier, see page E-4. Note that the **alias** base modifier attribute is only allowed when the parameter attribute list precedes the function name.
- **assert_length**(*min;max*)**.** Use this base modifier attribute to verify that the actual length of the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[string out assert_length(10;~0)]]` indicates that the COBOL data item must contain at least ten characters.

Note The C construct, **~0**, denotes a pattern of all ones and represents the largest positive value that can be stored in a data item. This usage is preferable to other choices such as 0xffff (which requires knowing the number of f's to write) and –1 (which is not allowed C for unsigned data types).

- **leading_spaces.** Use this base modifier attribute to instruct CodeBridge to remove leading space characters on input, and for right-justified arguments, add leading space characters on output.
- **leading**(*value*)**.** This base modifier attribute is the same as the **leading_spaces** base modifier, except that the character represented by *value* is used instead of a space character.
- **no_null_pointer.** The calling COBOL program may pass a pointer with a null value as an argument either by specifying the figurative constant NULL (NULLS) or by specifying a COBOL pointer argument that has been set to NULL (NULLS). In this case, CodeBridge would normally pass a null pointer as a parameter to the C function. If the **no_null_pointer** base modifier attribute is used, an error condition will be generated instead.
- **occurs**(*value*)**.** Arrays of COBOL non-numeric arguments may be passed to a C function. Use the **occurs**(*value*) base modifier attribute to specify the array size. If the C function prototype specifies the array size, it is not necessary to use the **occurs**(*value*) base modifier attribute unless you need to override the value specified in the function prototype.
- **optional.** The calling COBOL program may omit the associated argument (see "Managing Omitted Arguments" beginning on page 2-17 for more information on omitted arguments and this attribute), in which case CodeBridge would normally generate an error condition. If the **optional** base modifier attribute is used, then a default value is generated and passed to the C function. The default value associated with a **general_string** or **string** base attribute is an empty string (the first character of the string is a null character). If a value other than the CodeBridge supplied default value is desired, see the **value_if_omitted**(*value*) base modifier attribute description.

  Note The current implementation of CodeBridge Builder only allows input optional parameters. Output parameters are required by default.
- **repeat**(*value*)**.** Use this base modifier attribute with the C parameter before the ellipsis when a variable number of C parameters is used. *value* indicates the maximum number of additional C parameters.
- **silent**. For a description of this base modifier, see page E-4.

- **size**(*value*)**.** Use this base modifier attribute with the **string** base attribute to specify a *value* that overrides the default length when the conversion string buffer is dynamically allocated.
- **trailing_spaces.** Use this base modifier attribute to instruct CodeBridge to remove trailing space characters on input and, for left-justified arguments, add trailing space characters on output.
- **trailing**(*value*)**.** This base modifier attribute is the same as the **trailing_spaces** modifier, except that the character represented by *value* is used instead of a space character.
- **value_if_omitted**(*value*)**.** Use this base modifier attribute to specify a value to be used when the calling COBOL program omits the associated argument (see "Managing Omitted Arguments" beginning on page 2-17 for more information on omitted arguments and this attribute). When this base modifier attribute is used, it is not necessary to also use the **optional** base modifier attribute. A **string** attribute list must specify a string value (for example, `value_if_omitted("Default")`).

general_string Base Attribute

The **general_string** base attribute is used in those cases when it is desirable to allow a C string parameter to accept either a numeric COBOL argument or a non-numeric COBOL argument. When a numeric argument is passed to a parameter described with the **general_string** base attribute, the argument is converted as if the parameter were described with the **numeric_string** base attribute; otherwise, the argument is converted as if the parameter were described with the **string** base attribute. An attribute list containing the **general_string** base attribute allows any additional attributes that may be used with either a **string** base attribute or a **numeric_string** base attribute. For each call and for each argument passed to a parameter within a set of a variable number of parameters, attributes that do not apply to the COBOL argument actually passed are ignored for the conversion of that argument. That is, for a numeric argument, base modifier attributes not applicable to the **numeric_string** base attribute are ignored and for a non-numeric argument, base modifier attributes not applicable to the **string** base attribute are ignored. Refer to "Numeric Base Attributes" on page E-5 and "string Base Attribute" on page E-11 for further information.

In **general_string** attribute lists, base modifier attributes that apply to a **numeric_string** or **string** base attribute may be used together. Those base modifier attributes that do not apply for a given passed argument are ignored (for example, **trailing_sign** for a non-numeric COBOL argument).

String Length Base Attributes

The string length base attributes, **buffer_length** and **effective_length**, are used to pass length information about a string parameter as a separate parameter to a C function. Attribute lists formed with these base attributes are used with the attribute lists formed with the **general_string**, **numeric_string**, and **string** base attributes. By default, these length attributes refer to the same COBOL argument number as the base attribute in the preceding attribute list. If the length attribute list does not immediately follow the associated attribute list, then the **arg_num**(*value*) argument number attribute must be used, where *value* must be the same as used in an **arg_num**(*value*) attribute of the associated **general_string**, **numeric_string**, or **string** base attribute.

The string length base attributes include the following:

- **buffer_length.** The **buffer_length** base attribute describes a C numeric parameter and instructs CodeBridge to pass the length of the conversion buffer to the C function as the value of the parameter. The length of the buffer is determined by the base attribute that is used to describe the string parameter associated with the same argument, as follows:
  - For the **string** base attribute, the buffer length defaults to one more than the length of the passed COBOL argument, which allows space for the characters of the argument value and a null-termination character.
  - For the **numeric_string** base attribute, the buffer length defaults to four more than the digit length of the passed COBOL argument, which allows space for the digits of the argument value and the sign, decimal-point, and null-termination characters.
  - For the **general_string** base attribute, the buffer length defaults to the greater of one more than the length of the passed COBOL argument and four more than the digit length of the passed COBOL argument, which allows space for either a non-numeric or numeric argument conversion.

  The default values for **buffer_length** may be overridden by using the **size**(*value*) base modifier attribute in the attribute list that contains the **string**, **numeric_string**, or **general_string** base attribute that is associated with the same argument as **buffer_length**.
- **effective_length.** The **effective_length** base attribute returns the actual number of characters stored in the conversion string buffer after the input conversion process is complete. (This is similar to the standard C library function, strlen.) This base attribute is used for obtaining the length of input string parameters denoted by **general_string**, **numeric_string**, or **string** base attributes.

Note To obtain the length of the COBOL argument, use the **length** base attribute described on page E-18.

Base Modifiers that Apply to String Length Base Attributes

The following base modifier attributes may be used with the string length base attributes:

- **occurs**(*value*)**.** Arrays of COBOL non-numeric arguments (or numeric arguments converted by numeric_string) may be passed to a C string parameter. Use the **occurs**(*value*) base modifier attribute to specify the array size. If the C function prototype specifies the array size, it is not necessary to use the **occurs**(*value*) base modifier attribute unless you need to override the value specified in the function prototype.

  Note The array size for the string length base attributes must be less than or equal to the array size of the C string parameter associated with the same argument number.

- **silent**. For a description of this base modifier, see page E-4.

Pointer Base Attributes

Pointer base attributes are used when passing a component of a COBOL pointer argument to the C function. These attributes are associated with the RM/COBOL POINTER data type, a new feature introduced in RM/COBOL version 7.0. A COBOL pointer describes a block of memory and has three components: base address, offset, and size. When a pointer data item is initialized, the base address contains the starting address of the block of memory, the offset is set to zero, and the size contains the total length of the block. The offset may be modified in an RM/COBOL program by using the Format 6 SET statement (see the *RM/COBOL Language Reference Manual*).

CodeBridge provides two approaches for accessing data described by a COBOL pointer data item. The first method is useful when the C function wishes to access or modify memory referenced by the pointer. This approach uses the following two pointer base attributes, both of which are defined for input to the C function but not for output:

- **pointer_address.** Use the **pointer_address** base attribute to pass the effective address (base address plus offset) of a passed COBOL pointer argument to the C function.
- **pointer_length.** Use the **pointer_length** base attribute to pass the effective length (size minus offset) of a passed COBOL pointer argument to the C function. This is the amount of data between the current value of the pointer and the end of the block of memory described by the pointer.

The second approach is useful if the C function wishes to access the components of the COBOL pointer data item directly. This method is useful when the C function wishes to change one of the components of a COBOL pointer.

Note Although CodeBridge provides the ability to change the value of COBOL data areas or COBOL pointers, caution should be used due to the potential risk of corrupting the COBOL program.

The second approach uses the following three pointer base attributes, all of which may be used for both input and output:

- **pointer_base.** Use the **pointer_base** base attribute to pass the base address component of a passed COBOL pointer argument to and from the C function.
- **pointer_offset.** Use the **pointer_offset** base attribute to pass the offset component of a passed COBOL pointer argument to and from the C function.
- **pointer_size.** Use the **pointer_size** base attribute to pass the size component of a passed COBOL pointer argument to and from the C function.

Note A COBOL pointer data item with a zero base address component is always a null pointer, regardless of the offset and size values. If the base address of a pointer is set to a zero value or remains a zero value, the pointer offset and size components cannot be set to non-zero values. When a COBOL pointer data item with a zero base address component is stored, the pointer offset and size components will be set to zero.

Base Modifiers that Apply to Pointer Base Attributes

In addition to the **alias**(*name*) and **silent** base modifier attributes (see page E-4), two other base modifier attributes are available for the second approach described above:

- **pointer_max_size**. Use this base modifier attribute when either the **pointer_base** or **pointer_offset** base attribute is used for output to force the pointer size component to a value of all ones.
- **pointer_reset_offset**. Use this base modifier attribute when either the **pointer_base** or **pointer_size** base attribute is used for output to force the **pointer_offset** component to a value of zero. For an example of using **pointer_reset_offset**, see "Example 4: Accessing COBOL Pointer Arguments", which begins on page B-9.

Descriptor Base Attributes

Sometimes it may be necessary to pass individual data descriptor components for a COBOL argument, as well as the argument count, the COBOL initial state flag, or the Windows handle, directly as C parameters. (See "Passing COBOL Descriptor Data" on page 2-15 and "Passing Miscellaneous Information" on page 2-17.)

The following lists the descriptor base attributes:

- **address.** Use the **address** base attribute when passing the address of a passed COBOL argument to the C function. By using this attribute, the C function may modify the COBOL data area directly. When the address of a COBOL data item is passed in this way, the C function is responsible for any parameter conversion that is required. The address may be saved by the C function and used by this or other functions in the non-COBOL subprogram later in the run unit. However, if the address refers to a data item in a COBOL program that is later canceled, the saved address may no longer be valid. It is the programmer's responsibility to prevent such situations.

- **arg_count.** Use the **arg_count** base attribute to pass the actual number of COBOL arguments to the C function. The **arg_count** base attribute does not refer to a COBOL argument.

  The argument count is the number of actual arguments specified in the USING phrase of the CALL statement, including any arguments explicitly specified by the OMITTED keyword. The count does not include the argument specified in the GIVING (RETURNING) phrase.

  Note When using the explicit argument association method, it is an error to specify the argument number attribute, **arg_num**(*value*), with the **arg_count** base attribute since this base attribute does not refer to a COBOL argument.

- **digits.** Use the **digits** base attribute when passing the digit count, that is, the number of 9's in the PICTURE character-string, of a passed COBOL numeric argument to the C function. If the item is not numeric, the results are undefined.

- **initial_state.** Use the **initial_state** base attribute to pass the COBOL initial state flag to the C function. The **initial_state** base attribute does not refer to a COBOL argument. It returns information about the state of the called program within the run unit.

  When the COBOL initial state flag is zero, the C function may choose to reinitialize any "state" variables it contains. When it is non-zero, the C function uses the current values of any "state" variables. For more information, see item number 4 on page G-6.

  Note 1 A "state" variable is one whose contents are normally preserved between function calls.

  Note 2 When using the explicit argument association method, it is an error to specify the argument number attribute, **arg_num**(*value*), with the **initial_state** base attribute since this base attribute does not refer to a COBOL argument.

- **length.** Use the **length** base attribute when passing the length (in bytes) of a passed COBOL argument to the C function. The **length** attribute may be used for the same argument as the **address** base attribute to allow a C function to modify the COBOL data area directly. Other uses also exist; for example, the **length** base attribute may be used for the same argument as the **string** base attribute to pass the maximum size that a string may occupy (it does not include space for the trailing null character).

- **scale.** Use the **scale** base attribute when passing the digit count of the number of digits to the right of the decimal point in a passed COBOL numeric argument to the C function. If the item is not numeric, the results are undefined. The scale value is the arithmetic complement of the scale value in the COBOL argument descriptor.

  Note If the COBOL data item uses P-scaling, the scaling factor may be negative. For example, for a PIC 9(7)P(3) data item, using this attribute will pass -3 to the C function; for a PIC P(3)9(7) data item, using this attribute will pass 10 to the C function.

- **type.** Use the **type** base attribute when passing the type code of a passed COBOL argument to the C function. Type codes, which are defined in the header file **rmc85cal.h**, are included in Table E-1 for easy reference. Note that some values are classified as "reserved" in the "Classification" column. They either refer to internal formats that are not used by CodeBridge or to values that are reserved for future use.

Table E-1: Type Attribute Codes

Name	Value	Classification	Description
RM_NSE	0	Numeric	Numeric String Edited
RM_NSU	1	Numeric	Display String Unsigned
RM_NTS	2	Numeric	Display Trailing Separate
RM_NTC	3	Numeric	Display Trailing Combined
RM_NLS	4	Numeric	Display Leading Separate
RM_NLC	5	Numeric	Display Leading Combined
RM_NCS	6	Numeric	Comp (unpacked) Signed
RM_NCU	7	Numeric	Comp (unpacked) Unsigned
RM_NPP	8	Numeric	Packed Positive
RM_NPS	9	Numeric	Packed Signed
RM_NPU	10	Numeric	Packed Unsigned
RM_NBS	11	Numeric	Binary Signed
RM_NBU	12	Numeric	Binary Unsigned or Index
	13 – 15	Reserved	
RM_ANS	16	Non-numeric	Alphanumeric String
RM_ANSR	17	Non-numeric	Alphanumeric (Right Justified)
RM_ABS	18	Non-numeric	Alphabetic String
RM_ABSR	19	Non-numeric	Alphabetic (Right Justified)
RM_ANSE	20	Non-numeric	Alphanumeric String Edited
RM_ABSE	21	Non-numeric	Alphabetic String Edited
RM_GRPF	22	Non-numeric	Group
	23 – 24	Reserved	
RM_PTR	25	Pointer	COBOL Pointer
RM_NBSN	26	Numeric	Binary Signed Native
RM_NBUN	27	Numeric	Binary Unsigned Native
	28 – 31	Reserved	
RM_OMITTED	32	Omitted	Omitted argument

- **windows_handle.** Use the **windows_handle** base attribute to pass the Windows handle associated with the run unit to the C function. This attribute, which is available only for Windows systems, is useful when calling some Windows APIs. For example, when opening a new window, it may be necessary to supply the handle of the parent's window. The **windows_handle** base attribute does not refer to a COBOL argument.

  Note 1 The **windows_handle** base attribute is not available on UNIX platforms as it can cause compilation errors.

  Note 2 When using the explicit argument association method, it is an error to specify the argument number attribute, **arg_num**(*value*), with the **windows_handle** base attribute since this base attribute does not refer to a COBOL argument.

Base Modifier that Applies to Descriptor Base Attributes

Only one base modifier attribute, **silent**, is used with descriptor base attributes. For a description of this base modifier, see page E-4.

Error Base Attributes

Occasionally, either the C library or one of the Windows API functions will return error information that must be retrieved separately from the C function that is called.

The C library often places error information in the external variable, errno. If the called function returns a value of –1, then in the calling program value of the external variable errno is the error code. In releases prior to version 7.1, CodeBridge had no means of accessing this variable.

Some Windows APIs return error information that must be retrieved by calling the C function, GetLastError. If the called function returns a status of FALSE (numeric zero), then the calling program must call the function GetLastError to obtain the error number. In many cases, however, the value that would have been returned by GetLastError likely will be modified by the RM/COBOL runtime between successive calls from the COBOL program, making it impossible to call GetLastError as a separate function.

Error base attributes associate with a COBOL argument for which there is no corresponding C function return or parameter. Two error base attributes have been added to CodeBridge that deal with these situations:

- **errno.** Use the **errno** base attribute to retrieve the contents of the external variable, errno. Specifying the base attribute **errno** is similar to specifying **integer out**, except that it does not associate with a C function return or parameter. While this attribute does not associate with the C function return or any parameter, the position of the attribute list within the C function prototype in which it appears is significant for determining the COBOL argument number when automatic argument association is used, as described on page 2-22. The external variable errno, which is the source item for the attribute **errno**, has the C type of **int**, which is signed. The assumed direction attribute is **out**; a direction attribute is not allowed with the attribute **errno**.
- **get_last_error.** Use the **get_last_error** base attribute to retrieve the contents returned by the C function, GetLastError. Specifying **get_last_error** is similar to specifying **integer out unsigned**, except that it does not associate with a C function return or parameter. While this attribute does not associate with the C function return or any parameter, the position of the attribute list within the C function prototype in which it appears is significant for determining the COBOL argument number when automatic argument association is used, as described on page 2-22. The return value of GetLastError, which is the source item for the attribute **get_last_error**, has the Windows type of **DWORD**, which is unsigned. The assumed direction attribute is **out**; a direction attribute is not allowed with the attribute **get_last_error**.

Error base attributes refer to an argument in the COBOL CALL statement, but do not refer to any C function return value or parameter. These attributes cause the CodeBridge Builder to generate separate code sequences to return the value of the external variable errno or the return value of the Windows GetLastError function. For additional information, see “Returning C Error Values” on page 2-18.

Error base attributes are, in a certain sense, the opposite of descriptor base attributes (these include **arg_count**, **initial_state**, and **windows_handle**). The error base attributes describe a COBOL argument for which there is no corresponding C parameter, because the source item for these attributes is not described in the C function prototype, and are output (to the COBOL argument) only. The descriptor base attributes are used to develop input values for C parameters from a source other than a COBOL argument or from the description of a COBOL argument.

Base Modifiers that Apply to Error Base Attributes

The error base attributes may be used in an attribute list with the same base modifier attributes as for the base attribute **integer** with the following exception:

- The **unsigned** attribute is not allowed. It would be incorrect for **errno** and is implied for **get_last_error**.

The error base attributes can be supplemented with additional information by the base modifier attributes listed below:

- **alias**(*name*)**.** For a description of this base modifier, see page E-4. Note that the **alias** base modifier attribute is only allowed when the parameter attribute list precedes the function name.
- **assert_digits**(*min;max*)**.** Use this base modifier attribute to verify that the digit length of the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[errno assert_digits(9;18)]]` indicates that the COBOL data item must contain from 9 to 18 digits.

  The use of P-scaling in the COBOL program will increase the digit length by the number of P symbols specified in the PICTURE character-string. For example, all of the PICTURE character-strings 9(8), 9(5)P(3), and VP(3)9(5) describe a data item with a digit length of eight for CodeBridge.
- **assert_digits_left**(*min;max*)**.** Use this base modifier attribute to verify that the number of digits to the left of the decimal point in the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[get_last_error assert_digits_left(5;~0)]]` indicates that the COBOL data item must contain five or more digits to the left of the decimal point, or equivalently, no less than five digits before the decimal point.

  Note The C construct, **~0**, denotes a pattern of all ones and represents the largest positive value that can be stored in a data item. This usage is preferable to other choices such as 0xffff (which requires knowing the number of f's to write) and –1 (which is not allowed C for unsigned data types).

  The use of P-scaling in the COBOL program will increase the number of digits to the left of the decimal point by the number of P symbols specified in the PICTURE character-string that occur to the left of the decimal point. For example, both of the PICTURE character-strings 9(8) and 9(5)P(3) describe a data item with eight digits to the left of the decimal point for CodeBridge.

- **assert_digits_right**(*min;max*)**.** Use this base modifier attribute to verify that the number of digits to the right of the decimal point in the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[errno assert_digits_right(0;0)]]` indicates that the COBOL data item must contain no digits after the decimal point.

  The use of P-scaling in the COBOL program will increase the number of digits to the right of the decimal point by the number of P symbols specified in the PICTURE character-string that occur to the right of the decimal point. For example, both of the PICTURE character-strings V9(8) and VP(3)9(5) describe a data item with eight digits to the right of the decimal point for CodeBridge.

- **assert_length**(*min;max*)**.** Use this base modifier attribute to verify that the actual length of the passed COBOL argument is within the range specified by *min* and *max*. For example, `[[get_last_error assert_length(10;~0)]]` indicates that the COBOL data item must contain at least ten characters.

  Note The C construct, **~0**, denotes a pattern of all ones and represents the largest positive value that can be stored in a data item. This usage is preferable to other choices such as 0xffff (which requires knowing the number of f's to write) and –1 (which is not allowed C for unsigned data types).

- **assert_signed.** Use this base modifier attribute to verify that the passed COBOL argument contains a sign.

- **assert_unsigned.** Use this base modifier attribute to verify that the passed COBOL argument does not contain a sign.

- **no_size_error.** During conversion (either COBOL to C or C to COBOL), it is possible that leading digits will be lost. If this occurs, the normal behavior is to generate an error condition. If the **no_size_error** base modifier attribute is used, the error condition will be ignored.

- **rounded.** Use this base modifier attribute to cause rounding in those cases where truncation would normally occur (on either input or output). Rounding is performed using COBOL rounding rules.

- **scaled**(*value*)**.** Use this base modifier attribute to scale integer values during the conversion process. On output, the C value is divided by 10^{value}.

  For example, if the attribute list is `[[errno scaled(2)]]` and the C function changed the value of the external variable errno to 123, the COBOL program would receive 1.23 back.

- **silent.** For a description of this base modifier, see page E-4.

Parameter Attributes Summary

Table E-2 lists all available parameter attributes in alphabetical order. The "Attribute Category" column contains the category of the parameter attribute as one of the categories: Argument Number, Direction, Base or Base Modifier, as discussed in earlier sections. The "Modifier Usage" column indicates whether base modifier attributes affect the COBOL argument, the C data item, or the C function name. The "Description" column presents a brief overview of the function of the parameter attribute.

Table E-2: Parameter Attributes Summary

Parameter Attribute	Attribute Category	Modifier Usage	Description
address	Base (Descriptor)		Passes the address of a passed COBOL argument to the C function. See page E-17.
alias(*name*)	Base Modifier	C Function Name	Changes the generated function name to be the name specified by *name*. See page E-4.
arg_count	Base (Descriptor)		Passes the actual number of COBOL arguments to the C function. See page E-17.
arg_num(*value*)	Argument Number		Explicitly specifies the COBOL argument number of an argument in the USING phrase rather than accepting the default argument association. See page E-2.
assert_digits (*min;max*)	Base Modifier	COBOL Argument	Insures that the number of digits in the passed COBOL argument is within the range specified by *min* and *max*. This modifier is used with numeric base attributes. See page E-7.
assert_digits_left (*min;max*)	Base Modifier	COBOL Argument	Insures that the number of digits to the left of the decimal point in the passed COBOL argument is within the range specified by *min* and *max*. This modifier is used with numeric base attributes. See page E-7.

Table E-2: Parameter Attribute Summary (*Cont.*)

Parameter Attribute	Attribute Category	Modifier Usage	Description
assert_digits_right (*min;max*)	Base Modifier	COBOL Argument	Insures that the number of digits to the right of the decimal point in the passed COBOL argument is within the range specified by *min* and *max*. This modifier is used with numeric base attributes. See page E-8.
assert_length (*min;max*)	Base Modifier	COBOL Argument	Insures that the length of the passed COBOL argument is within the range specified by *min* and *max*. This modifier is used with numeric or string base attributes. See pages E-8 and E-11.
assert_signed	Base Modifier	COBOL Argument	Insures that the passed COBOL argument is signed. This modifier is used with numeric base attributes. See page E-8.
assert_unsigned	Base Modifier	COBOL Argument	Insures that the passed COBOL argument is unsigned. This modifier is used with numeric base attributes. See page E-8.
buffer_length	Base (String Length)		Passes the size (in bytes) of the string buffer to the C function. **buffer_length** is one greater than the length of a non-numeric COBOL argument or four greater than the digit length of a numeric COBOL argument. See page E-14.
digits	Base (Descriptor)		Passes the number of digits in a passed COBOL numeric argument to the C function. See page E-17.
effective_length	Base (String Length)		Passes the effective size (in bytes) of the string buffer to the C function. This is similar to the standard C library function, strlen. See page E-14.
errno	Base (Error)		Causes the external variable errno to be set to zero before the function call and the value of the external variable errno after the function call to be returned to a COBOL numeric argument. See page E-20.
float	Base (Numeric)		Converts COBOL numeric arguments to C floating-point parameters (such as float or double). See page E-5.

Table E-2: Parameter Attribute Summary (*Cont.*)

Parameter Attribute	Attribute Category	Modifier Usage	Description
general_string	Base (Numeric or String)		Converts numeric and non-numeric COBOL arguments to null-terminated C strings. Numeric COBOL arguments are treated as if the **numeric_string** base attribute were specified. Non-numeric COBOL arguments are treated as if the **string** base attribute were specified. See page E-13.
get_last_error	Base (Error)		Causes the Windows error code to be set to zero by a call to SetLastError before the function call and the value returned from a call to GetLastError after the function call to be returned to a COBOL numeric argument. See page E-20.
in	Direction		Specifies an input parameter to the C function. See page E-2.
initial_state	Base (Descriptor)		Passes the COBOL initial state flag to the C function. See page E-18.
integer	Base (Numeric)		Converts COBOL numeric arguments to C integer parameters (such as char, short, int, or long). See page E-5.
integer_only	Base Modifier	COBOL Argument	Insures that the passed COBOL argument is an integer (no digits are allowed to the right of the decimal point). This modifier is used with the **integer** base attribute. See page E-10.
leading(*value*)	Base Modifier	C Parameter	Specifies the use of leading strip/fill characters indicated by *value*. This modifier is used with the **string** base attribute. See page E-12.
leading_minus	Base Modifier	C Parameter	Forces a minus sign character (“-”) to be placed before the numeric value when the value is negative. Positive values do not contain a sign character. This modifier is used with the **numeric_string** base attribute. See page E-10.

Table E-2: Parameter Attribute Summary (*Cont.*)

Parameter Attribute	Attribute Category	Modifier Usage	Description
leading_sign	Base Modifier	C Parameter	Forces a sign character, either a plus (“+”) or a minus (“-”), depending on the sign of the value, to be placed before the numeric value. This modifier is used with the **numeric_string** base attribute. See page E-10.
leading_spaces	Base Modifier	C Parameter	Specifies the use of leading strip/fill space characters. This modifier is used with the **string** base attribute. See page E-12.
length	Base (Descriptor)		Passes the size (in bytes) of a passed COBOL argument to the C function. See page E-18.
no_null_pointer	Base Modifier	COBOL Argument	Returns an error if the COBOL program passes a pointer with a null value as an argument. This modifier is used with numeric or string base attributes. See pages E-8 and E-12.
no_size_error	Base Modifier	COBOL Argument	Causes numeric conversion errors to be ignored. This modifier is used with numeric base attributes. See page E-9.
numeric_string	Base (Numeric)		Converts COBOL numeric arguments to null-terminated C strings. See page E-6.
occurs(*value*)	Base Modifier	C Parameter	Specifies that the parameter is an array containing *value* elements. This modifier is used with numeric or string base attributes. It is also used with the **buffer_length** and **effective_length** base attributes. See pages E-9 and E-12.
optional	Base Modifier	COBOL Argument	Allows the COBOL program to omit an input argument even though a C parameter is associated with that argument. This modifier is used with numeric or string base attributes. See pages E-9 and E-12.
out	Direction		Specifies an output parameter from the C function and causes an output conversion into the associated COBOL argument. See page E-2.

Table E-2: Parameter Attribute Summary (*Cont.*)

Parameter Attribute	Attribute Category	Modifier Usage	Description
pointer_address	Base (Pointer)		Passes the effective address (base address component plus offset component) of a passed COBOL pointer argument to the C function. See page E-15.
pointer_base	Base (Pointer)		Passes the base address component of a passed COBOL pointer argument to the C function. See page E-16.
pointer_length	Base (Pointer)		Passes the effective length (size component minus offset component) of a passed COBOL pointer argument to the C function. See page E-15.
pointer_max_size	Base Modifier	COBOL Argument	Sets the size component of a passed COBOL pointer argument to the maximum value (all ones) on output. This modifier is used with the **pointer_base** or **pointer_offset** base attributes. See page E-16.
pointer_offset	Base (Pointer)		Passes the offset component of a passed COBOL pointer argument to the C function. See page E-16.
pointer_reset_offset	Base Modifier	COBOL Argument	Sets the offset component of a passed COBOL pointer argument to zero on output. This modifier is used with the **pointer_base** or **pointer_size** base attributes. See page E-16.
pointer_size	Base (Pointer)		Passes the size component of a passed COBOL pointer argument to the C function. See page E-16.
repeat(*value*)	Base Modifier	C Parameter	Used when the C function expects a variable number of parameters. This modifier is used for numeric or string base attributes. See pages E-9 and E-12.
ret_val	Argument Number		Explicitly specifies the COBOL argument in the GIVING (RETURNING) phrase rather than accepting the default argument association. See page E-2.

Table E-2: Parameter Attribute Summary (*Cont.*)

Parameter Attribute	Attribute Category	Modifier Usage	Description
rounded	Base Modifier	COBOL Argument	Causes rounding (instead of truncation) to occur during parameter conversion when trailing digits must be removed. This modifier is used with numeric base attributes. See page E-9.
scale	Base (Descriptor)		Passes the scale of a passed COBOL numeric argument to the C function. If a COBOL argument had a picture of 999V99, the scale used by COBOL is –2. This value is negated and passed as +2 to the C function. If the picture contains "P" characters, this value may appear unusual. See page E-18.
scaled(*value*)	Base Modifier	C Parameter	On input, multiplies the passed COBOL argument by a 10^{value}. On output, divides the C parameter by a 10^{value}. This modifier is used with the **integer** base attribute. See page E-10.
silent	Base Modifier	C Parameter	Suppresses display of errors detected during conversion or validation. See page E-4.
size(*value*)	Base Modifier	C Parameter	Used with numeric_string and string base attributes to override the default length (its size or precision) of the passed COBOL argument. See pages E-10 and E-13.
string	Base (String)		Converts COBOL non-numeric arguments to null-terminated C strings. See page E-11.
trailing(*value*)	Base Modifier	C Parameter	Specifies the use of trailing strip/fill characters indicated by *value*. This modifier is used with the **string** base attribute. See page E-13.
trailing_credit	Base Modifier	C Parameter	Forces a credit symbol ("CR") to be placed after the numeric value when the value is negative. Positive values do not contain a sign representation. This modifier is used with the **numeric_string** base attribute. See page E-10.

Table E-2: Parameter Attribute Summary (*Cont.*)

Parameter Attribute	Attribute Category	Modifier Usage	Description
trailing_debit	Base Modifier	C Parameter	Forces a debit symbol (“DB”) to be placed after the numeric value when the value is negative. Positive values do not contain a sign representation. This modifier is used with the **numeric_string** base attribute. See page E-10.
trailing_minus	Base Modifier	C Parameter	Forces a minus sign character (“-”) to be placed after the numeric value when the value is negative. Positive values do not contain a sign character. This modifier is used with the **numeric_string** base attribute. See page E-10.
trailing_sign	Base Modifier	C Parameter	Forces sign character, either a plus (“+”) or a minus (“-”) sign character, depending on the sign of the value, to be placed after the numeric value. This modifier is used with the **numeric_string** base attribute. See page E-10.
trailing_spaces	Base Modifier	C Parameter	Specifies the use of trailing strip/fill space characters. This modifier is used with the **string** base attribute. See page E-13.
type	Base (Descriptor)		Passes the type-code of a passed COBOL argument to the C function. See page E-18.
unsigned	Base Modifier	C Parameter	Indicates that the C parameter is unsigned. If this attribute is not used, all integer C parameters are treated as signed. This modifier is used with the **integer** base attribute. See page E-10.
value_if_omitted (*value*)	Base Modifier	COBOL Argument	Assigns a default value when the COBOL program omits the associated argument. This modifier is used with the numeric or string base attributes. See pages E-9 and E-13.
windows_handle	Base (Descriptor)		Passes the Windows handle of the current COBOL CALL to the C function. This attribute is available only for Windows systems. See page E-20.

Parameter Attribute Combinations

CodeBridge Builder recognizes various parameter attribute combinations. Table E-3 is a quick reference that lists the allowed combinations. For instance, some base modifier attributes make sense only for input or output. In those cases, there are separate rows for "in only" and "out only".

Note When the "Direction" column contains "in (assumed)", the direction is always assumed to be "in", but the **in** direction attribute is not allowed.

Table E-3: Parameter Attribute Combinations

Base	Direction	Argument Number	Modifiers	
address	in (assumed)	arg_num	silent	
arg_count	in (assumed)	none	silent	
buffer_length	in (assumed)	arg_num	occurs	silent
digits	in (assumed)	arg_num	silent	
effective_length	in (assumed)	arg_num	occurs	silent
errno	out (assumed)	ret_val	alias assert_digits assert_digits_left assert_digits_right assert_length assert_signed	assert_unsigned no_size_error rounded scaled silent
	out (assumed)	arg_num	assert_digits assert_digits_left assert_digits_right assert_length assert_signed	assert_unsigned no_size_error rounded scaled silent
float	in only		optional	value_if_omitted
	out only	ret_val	alias	
	either	arg_num	assert_digits assert_digits_left assert_digits_right assert_length assert_signed assert_unsigned	no_null_pointer no_size_error occurs repeat rounded silent

Table E-3: Parameter Attribute Combinations (*Cont.*)

Base	Direction	Argument Number	Modifiers	
general_string	in only		leading_minus leading_sign optional trailing_credit	trailing_debit trailing_minus trailing_sign value_if_omitted
	out only	ret_val	alias	
	either	arg_num	assert_digits assert_digits_left assert_digits_right assert_length assert_signed assert_unsigned leading leading_spaces no_null_pointer	no_size_error occurs repeat rounded silent size trailing trailing_spaces
get_last_error	out (assumed)	ret_val	alias assert_digits assert_digits_left assert_digits_right assert_length assert_signed	assert_unsigned no_size_error rounded scaled silent
	out (assumed)	arg_num	assert_digits assert_digits_left assert_digits_right assert_length assert_signed	assert_unsigned no_size_error rounded scaled silent
initial_state	in (assumed)	none	silent	
integer	in only		integer_only optional	value_if_omitted
	out only	ret_val	alias	
	either	arg_num	assert_digits assert_digits_left assert_digits_right assert_length assert_signed assert_unsigned no_null_pointer	no_size_error occurs repeat rounded scaled silent unsigned

Table E-3: Parameter Attribute Combinations (*Cont.*)

Base	Direction	Argument Number	Modifiers
length	in (assumed)	arg_num	silent
numeric_string	in only		leading_minus leading_sign optional trailing_credit trailing_debit trailing_minus trailing_sign value_if_omitted
	out only	ret_val	alias
	either	arg_num	assert_digits assert_digits_left assert_digits_right assert_length assert_signed assert_unsigned no_null_pointer no_size_error occurs repeat rounded silent size
pointer_address	in (assumed)	arg_num	silent
pointer_base	in only		
	out only	ret_val	alias pointer_max_size pointer_reset_offset
	either	arg_num	silent
pointer_length	in (assumed)	arg_num	silent
pointer_offset	in only		
	out only	ret_val	alias pointer_max_size
	either	arg_num	silent
pointer_size	in only		
	out only	ret_val	alias pointer_reset_offset
	either	arg_num	silent
scale	in (assumed)	arg_num	silent

Table E-3: Parameter Attribute Combinations (*Cont.*)

Base	Direction	Argument Number	Modifiers	
string	in only		optional	value_if_omitted
	out only	ret_val	alias	
	either	arg_num	assert_length leading leading_spaces no_null_pointer occurs	repeat silent size trailing trailing_spaces
type	in (assumed)	arg_num	silent	
windows_handle (This attribute is available only for Windows systems. See page E-20.)	in (assumed)	none	silent	

Appendix F: CodeBridge Library Functions

The CodeBridge Library is a collection of functions that are included in the RM/COBOL runtime system. These functions are used to convert input data from COBOL arguments to C parameters on entry and from C parameters to COBOL arguments just prior to exit. The CodeBridge Library also contains functions that perform data range and integrity checks.

This appendix describes each function in the CodeBridge Library. These descriptions will help you understand the C code generated by the CodeBridge Builder. Information on specifying the *Flags* parameter is also covered. The information in this appendix will also prove useful if you are debugging applications developed using CodeBridge.

Note The information presented here assumes a working knowledge of the C programming language. The material in Appendix C, *Useful C Information*, is not comprehensive enough to provide this necessary background.

Overview

The CodeBridge Library consists of the conversion and validation functions shown in Table F-1. (These functions are described in detail beginning on page F-6.) Input functions are called before the C function is called. Output functions are called after the C function is called but before returning to the calling COBOL program.

Note Each of these routines returns FALSE if an error condition occurs. Logic in the C source code file (generated by the CodeBridge Builder) will terminate the DLL and return an error to the RM/COBOL runtime system, which will terminate the calling COBOL program. See Appendix A, *CodeBridge Errors*, for a list of these errors.

Table F-1: CodeBridge Library Functions

Function Name	Input or Output	Used For
AssertDigits	Either	[[*numeric* assert_digits]]
AssertDigitsLeft	Either	[[*numeric* assert_digits_left]]
AssertDigitsRight	Either	[[*numeric* assert_digits_right]]
AssertLength	Either	[[*numeric* assert_length]] or [[*string* assert_length]]
AssertSigned	Either	[[*numeric* assert_signed]]
AssertUnsigned	Either	[[*numeric* assert_unsigned]]
BufferLength	Input	[[buffer_length]]
CobolArgCount	Input	[[arg_count]]
CobolDescriptorAddress	Input	[[address]]
CobolDescriptorDigits	Input	[[digits]]
CobolDescriptorLength	Input	[[length]]
CobolDescriptorScale	Input	[[scale]]
CobolDescriptorType	Input	[[type]]
CobolInitialState	Input	[[initial_state]]
CobolToFloat	Input	[[float]]
CobolToGeneralString	Input	[[general_string]]
CobolToInteger	Input	[[integer]]
CobolToNumericString	Input	[[numeric_string]]
CobolToPointerAddress	Input	[[pointer_address]]
CobolToPointerBase	Input	[[pointer_base in]]
CobolToPointerLength	Input	[[pointer_length]]
CobolToPointerOffset	Input	[[pointer_offset in]]
CobolToPointerSize	Input	[[pointer_size in]]
CobolToString	Input	[[string]]
CobolWindowsHandle	Input	[[windows_handle]]
ConversionCleanup	Neither	Cleanup during conversion exit.
ConversionStartup	Neither	Initialization of conversion process.
DiagnosticMode	Global	[# diagnostic(*flag*) #]

Table F-1: CodeBridge Library Functions (*Cont.*)

Function Name	Input or Output	Used For
EffectiveLength	Input	[[effective_length]]
FloatToCobol	Output	[[float out]]
GeneralStringToCobol	Output	[[general_string out]]
GetCallerInfo	Neither	Obtaining information about the calling COBOL program.
IntegerToCobol	Output	[[integer out]]
NumericStringToCobol	Output	[[numeric_string out]]
PointerBaseToCobol	Output	[[pointer_base out]]
PointerOffsetToCobol	Output	[[pointer_offset out]]
PointerSizeToCobol	Output	[[pointer_size out]]
StringToCobol	Output	[[string out]]

The series of functions that begin with “Assert” are designated as “Either” in the Input or Output column. It is recommended that these functions be called prior to the execution of the C function.

The ConversionStartup, ConversionCleanup, and GetCallerInfo functions are designated as “Neither” in the Input or Output column. The ConversionStartup function should be called once just after entry from COBOL. The ConversionCleanup function should be called once just prior to returning to COBOL. The GetCallerInfo function may be called at any time; it is usually called after an error is detected in order to add calling program information to an error message.

The DiagnosticMode function is designated as “Global” in the Input or Output column. This function may be called at any time, including multiple times, after the call to ConversionStartup and prior to the call to ConversionCleanup.

Specifying the *Flags* Parameter

The behavior of the CodeBridge Library conversion and validation functions is determined by flag settings in the *Flags* parameter. In some cases, the behavior requested by a flag requires that additional information be passed in another parameter. For example, when passing an array, you must set both the PF_OCCURS flag and pass the array size in the *Occurs* parameter.

Values for the *Flags* parameter, which is used with most of the CodeBridge Library functions, are defined in **cbridge.h**. These values correspond to the base modifier attributes that can be specified in template files. See Table F-2 on page F-5 for a list of flag definitions.

Normally, the PF_IN flag is used only for documentation purposes. However, when a Numeric or String output conversion function (FloatToCobol, GeneralStringToCobol, IntegerToCobol, NumericStringToCobol, and StringToCobol) is used, the corresponding Numeric or String input conversion function (CobolToFloat, CobolToGeneralString, CobolToInteger, CobolToNumericString, and CobolToString) must also be called. This is true even when the COBOL argument is not used as an input to the C function. For these reasons, the setting of the PF_IN flag is critical for Numeric and String input conversions. When the PF_IN flag is not set, initialization of the C data item is not performed, but the initialization necessary for the output conversion is performed.

The PF_OCCURS, PF_OUT, and PF_RETURN_VALUE flags are not used in the current implementation of the CodeBridge Library and, therefore, are used only for documentation purposes. However, because of possible changes to future versions of the CodeBridge Library, we recommend that these flags be set whenever appropriate. That is, calls to the CodeBridge Library output functions (FloatToCobol, GeneralStringToCobol, IntegerToCobol, NumericStringToCobol, PointerBaseToCobol, PointerOffsetToCobol, PointerSizeToCobol, and StringToCobol) should set the PF_OUT flag. When associated with the C function return value, calls to these same output functions should set the PF_RETURN_VALUE flag in addition to the PF_OUT flag. The PF_OCCURS flag should be set whenever an array is specified.

Although the following masks are neither used nor required in any CodeBridge Library call, they are provided for convenience and completeness:

- PF_LEADING. This mask is a combination of the PF_LEADING_SPACES flag and the PF_LEADING_VALUE flag.
- PF_TRAILING. This mask is a combination of the PF_TRAILING_SPACES flag and the PF_TRAILING_VALUE flag.
- PF_NUMERIC_STRING_MASK. This mask may be used to isolate the following flags: PF_LEADING_MINUS, PF_LEADING_SIGN, PF_TRAILING_CREDIT, PF_TRAILING_DEBIT, PF_TRAILING_MINUS, and PF_TRAILING_SIGN.

Table F-2: CodeBridge Library Flag Definitions

Name	Value	Description
PF_ASSERT_SIGNED	0x00000008	COBOL argument must be signed.
PF_ASSERT_UNSIGNED	0x00000010	COBOL argument must be unsigned.
PF_IN	0x00000020	Input argument for C function.
PF_INTEGER_ONLY	0x00000040	COBOL argument must be an integer.
PF_LEADING	0x00000180	Mask for leading strip/fill.
PF_LEADING_MINUS	0x00000001	Place "-" before negative value.
PF_LEADING_SIGN	0x00000000	Place "+" or "-" before value.
PF_LEADING_SPACES	0x00000080	Strip/fill leading spaces.
PF_LEADING_VALUE	0x00000100	Strip/fill leading *value*.
PF_NO_NULL_POINTER	0x00000200	Disallow NULL value for pointer.
PF_NO_SIZE_ERROR	0x00000400	Ignore numeric size errors.
PF_NUMERIC_STRING_MASK	0x00000007	numeric_string sign handling mask.
PF_OCCURS	0x00000800	Parameter is an array.
PF_OPTIONAL	0x00001000	Parameter is optional.
PF_OUT	0x00002000	Output parameter from C function.
PF_POINTER_MAX_SIZE	0x00004000	Maximize pointer size (all ones).
PF_POINTER_RESET_OFFSET	0x00008000	Clear pointer offset.
PF_REPEAT	0x00010000	Parameter repeated multiple times.
PF_RETURN_VALUE	0x00020000	Return value of the C function.
PF_ROUNDED	0x00040000	Round last digit if lost precision.
PF_SCALED	0x00080000	On input, multiply by 10^{value}; on output, divide by 10^{value}.
PF_SILENT	0x00100000	Suppress error message display.
PF_SIZE	0x00200000	Override default size of string.
PF_TRAILING	0x00C00000	Mask for trailing strip/fill.
PF_TRAILING_CREDIT	0x00000006	Place "CR" after negative value.
PF_TRAILING_DEBIT	0x00000007	Place "DB" after negative value.
PF_TRAILING_MINUS	0x00000005	Place "-"after negative value.
PF_TRAILING_SIGN	0x00000004	Place "+" or "-" after value
PF_TRAILING_SPACES	0x00400000	Strip/fill trailing spaces.
PF_TRAILING_VALUE	0x00800000	Strip/fill trailing *value*.
PF_UNSIGNED	0x01000000	C parameter is unsigned.
PF_VALUE_IF_OMITTED	0x02000000	Override *value* for omitted argument.

AssertDigits

AssertDigits returns TRUE if the number of digits for the COBOL argument is in the range specified by *MinValue* and *MaxValue*; otherwise, the function returns FALSE. This function also returns FALSE if the argument is not numeric.

If the COBOL CALL statement omits an argument (see "Managing Omitted Arguments" on page 2-17), the value that is substituted for the omitted argument is not checked by this function.

The use of P-scaling in the COBOL program will increase the digit length by the number of P symbols specified in the PICTURE character-string. For example, all of the PICTURE character-strings 9(8), 9(5)P(3), and VP(3)9(5) describe a data item with a digit length of eight for CodeBridge.

Calling Sequence

```
int _rmdll_RtCall->pAssertDigits
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments,
     int Flags,
     unsigned short MaxValue,
     unsigned short MinValue);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of data validation. Valid flag values (see Table F-2 on page F-5) for AssertDigits are: PF_OPTIONAL, PF_SILENT, and PF_VALUE_IF_OMITTED.

The value of the PF_OPTIONAL and PF_VALUE_IF_OMITTED flags must be the same as the corresponding conversion call (such as CobolToFloat or FloatToCobol, described on pages F-25 and F-46, respectively) for that argument.

MaxValue is the maximum allowed length, in digits.

MinValue is the minimum allowed length, in digits.

Note 1 The C construct, **~0**, may be used to indicate a value of all ones.

Note 2 *MaxValue* and *MinValue* may be specified in either order. The function will reverse their values if necessary.

AssertDigitsLeft

AssertDigitsLeft returns TRUE if the number of digits to the left of the decimal point for the COBOL argument is in the range specified by *MinValue* and *MaxValue*; otherwise, the function returns FALSE. This function also returns FALSE if the argument is not numeric.

If the COBOL CALL statement omits an argument (see "Managing Omitted Arguments" on page 2-17), the value that is substituted for the omitted argument is not checked by this function.

The use of P-scaling in the COBOL program will increase the number of digits to the left of the decimal point by the number of P symbols specified in the PICTURE character-string that occur to the left of the decimal point. For example, both of the PICTURE character-strings 9(8) and 9(5)P(3) describe a data item with eight digits to the left of the decimal point for CodeBridge.

Calling Sequence

```
int _rmdll_RtCall->pAssertDigitsLeft
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments,
     int Flags,
     unsigned short MaxValue,
     unsigned short MinValue);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of data validation. Valid flag values (see Table F-2 on page F-5) for AssertDigitsLeft are: PF_OPTIONAL, PF_SILENT, and PF_VALUE_IF_OMITTED.

The value of the PF_OPTIONAL and PF_VALUE_IF_OMITTED flags must be the same as the corresponding conversion call (such as CobolToFloat or FloatToCobol, described on pages F-25 and F-46, respectively) for that argument.

MaxValue is the maximum allowed digits to the left of the decimal point.

MinValue is the minimum allowed digits to the left of the decimal point.

Note 1 The C construct, **~0**, may be used to indicate a value of all ones.

Note 2 *MaxValue* and *MinValue* may be specified in either order. The function will reverse their values if necessary.

AssertDigitsRight

AssertDigitsRight returns TRUE if the number of digits to the right of the decimal point for the COBOL argument is in the range specified by *MinValue* and *MaxValue*; otherwise, the function returns FALSE. This function also returns FALSE if the argument is not numeric.

If the COBOL CALL statement omits an argument (see "Managing Omitted Arguments" on page 2-17), the value that is substituted for the omitted argument is not checked by this function.

The use of P-scaling in the COBOL program will increase the number of digits to the right of the decimal point by the number of P symbols specified in the PICTURE character-string that occur to the right of the decimal point. For example, both of the PICTURE character-strings V9(8) and VP(3)9(5) describe a data item with eight digits to the right of the decimal point for CodeBridge.

Calling Sequence

```
int _rmdll_RtCall->pAssertDigitsRight
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments,
     int Flags,
     unsigned short MaxValue,
     unsigned short MinValue);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of data validation. Valid flag values (see Table F-2 on page F-5) for AssertDigitsRight are: PF_OPTIONAL, PF_SILENT, and PF_VALUE_IF_OMITTED.

The value of the PF_OPTIONAL and PF_VALUE_IF_OMITTED flags must be the same as the corresponding conversion call (such as CobolToFloat or FloatToCobol, described on pages F-25 and F-46, respectively) for that argument.

MaxValue is the maximum allowed digits to the right of the decimal point.

MinValue is the minimum allowed digits to the right of the decimal point.

Note 1 The C construct, **~0**, may be used to indicate a value of all ones.

Note 2 *MaxValue* and *MinValue* may be specified in either order. The function will reverse their values if necessary.

AssertLength

AssertLength returns TRUE if the length of the COBOL argument (in bytes) is in the range specified by *MinValue* and *MaxValue*; otherwise, the function returns FALSE.

If the COBOL CALL statement omits an argument (see "Managing Omitted Arguments" on page 2-17), the value that is substituted for the omitted argument is not checked by this function.

Calling Sequence

```
int _rmdll_RtCall->pAssertLength
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments,
     int Flags,
     unsigned short MaxValue,
     unsigned short MinValue);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of data validation. Valid flag values (see Table F-2 on page F-5) for AssertLength are: PF_OPTIONAL, PF_SILENT, and PF_VALUE_IF_OMITTED.

The value of the PF_OPTIONAL and PF_VALUE_IF_OMITTED flags must be the same as the corresponding conversion call (such as CobolToFloat or FloatToCobol, described on pages F-25 and F-46, respectively) for that argument.

MaxValue is the maximum allowed length, in bytes.

MinValue is the minimum allowed length, in bytes.

Note 1 The C construct, **~0**, may be used to indicate a value of all ones.

Note 2 *MaxValue* and *MinValue* may be specified in either order. The function will reverse their values if necessary.

AssertSigned

AssertSigned returns TRUE if the COBOL argument is signed; otherwise, the function returns FALSE.

If the COBOL CALL statement omits an argument (see "Managing Omitted Arguments" on page 2-17), the value that is substituted for the omitted argument is not checked by this function.

Calling Sequence

```
int _rmdll_RtCall->pAssertSigned
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments,
     int Flags);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of data validation. Valid flag values (see Table F-2 on page F-5) for AssertSigned are: PF_OPTIONAL, PF_SILENT, and PF_VALUE_IF_OMITTED.

The value of the PF_OPTIONAL and PF_VALUE_IF_OMITTED flags must be the same as the corresponding conversion call (such as CobolToFloat or FloatToCobol, described on pages F-25 and F-46, respectively) for that argument.

AssertUnsigned

AssertUnsigned returns TRUE if the COBOL argument is unsigned; otherwise, the function returns FALSE.

If the COBOL CALL statement omits an argument (see "Managing Omitted Arguments" on page 2-17), the value that is substituted for the omitted argument is not checked by this function.

Calling Sequence

```
int _rmdll_RtCall->pAssertUnsigned
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments,
     int Flags);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of data validation. Valid flag values (see Table F-2 on page F-5) for AssertUnsigned are: PF_OPTIONAL, PF_SILENT, and PF_VALUE_IF_OMITTED.

The value of the PF_OPTIONAL and PF_VALUE_IF_OMITTED flags must be the same as the corresponding conversion call (such as CobolToFloat or FloatToCobol, described on pages F-25 and F-46, respectively) for that argument.

BufferLength

BufferLength obtains the length (in bytes) of the data buffer that has been allocated for conversion to and from the COBOL argument. For COBOL non-numeric arguments, this normally would be one more than the length of the argument. For COBOL numeric arguments, this normally would be four more than the digit length of the argument. This function returns TRUE if it is successful and FALSE if there is an error.

Note The BufferLength function may be used only in combination with one of the input string functions: CobolToGeneralString (see page F-27), CobolToNumericString (see page F-31), or CobolToString (see page F-38). *ArgNumber* must have the same value in the BufferLength function call and the corresponding input string function call. The call to BufferLength may precede or follow the call to the corresponding input string function.

Calling Sequence

```
int _rmdll_RtCall->pBufferLength
    (short ArgCount,
     short ArgNumber,
     CONV_TABLE *ConvTable,
     int Flags,
     int Occurs,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

ConvTable is the internal conversion table allocated by the ConversionStartup function (see page F-42).

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for BufferLength are: PF_OCCURS and PF_SILENT.

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Note For any given argument, the buffer length is a constant regardless of whether the argument is a scalar or an array. Thus, if you are writing you own C routine, there is no reason to have a buffer length parameter that is an array, even when the related C string parameter is an array.

Parameter is the address of the C parameter where the buffer length will be stored.

Size is the size of the C parameter.

CobolArgCount

CobolArgCount obtains that actual number of arguments passed from the calling COBOL program. This function returns TRUE if it is successful and FALSE if there is an error.

Note The CobolArgCount function is one of the trivial conversion functions described on page I-7.

Calling Sequence

```
int _rmdll_RtCall->pCobolArgCount
    (short ArgCount,
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

Flags modify the behavior of the conversion. The only valid flag value for CobolArgCount is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the argument count will be stored.

Size is the size of the C parameter.

CobolDescriptorAddress

CobolDescriptorAddress obtains the address of the COBOL argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolDescriptorAddress
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void **Parameter);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value for CobolDescriptorAddress is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C pointer where the address of the COBOL argument will be stored.

CobolDescriptorDigits

CobolDescriptorDigits obtains the digit count for the COBOL argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolDescriptorDigits
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value CobolDescriptorDigits is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the digit count will be stored.

Size is the size of the C parameter.

CobolDescriptorLength

CobolDescriptorLength obtains the length (in bytes) of the COBOL argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolDescriptorLength
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value for CobolDescriptorLength is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the length will be stored.

Size is the size of the C parameter.

CobolDescriptorScale

CobolDescriptorScale obtains the scale (the number of digits to the right of the decimal point) of the COBOL argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolDescriptorScale
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value CobolDescriptorScale is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the scale will be stored. The scale value returned is the arithmetic complement of the value in the COBOL descriptor.

Size is the size of the C parameter.

CobolDescriptorType

CobolDescriptorType obtains the type of the COBOL argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolDescriptorType
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value for CobolDescriptorType is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the Type value will be stored (see the discussion of "String Arrays" on page 2-34).

Size is the size of the C parameter.

CobolInitialState

CobolInitialState obtains the value of the initial state flag from the current COBOL CALL. This function returns TRUE if it is successful and FALSE if there is an error.

Note The CobolInitialState function is one of the trivial conversion functions described on page I-7.

When *State* is zero, the C function may choose to (re)initialize any "state" variables it contains. When *State* is non-zero, the C function may choose to use the current values of any "state" variables.

Note A "state" variable is one whose contents are normally preserved between function calls.

Calling Sequence

```
int _rmdll_RtCall->pCobolInitialState
    (int Flags,
     void *Parameter,
     int Size),
     short State);
```

Flags modify the behavior of the conversion. The only valid flag value for CobolInitialState is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the initial state flag will be stored. It may also be the address of an array of floating-point values if the PF_OCCURS flag is set.

Size is the size of the C parameter.

State is the initial state flag for the current COBOL CALL.

CobolToFloat

CobolToFloat converts the COBOL numeric argument to a C floating-point value. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, this function should be called prior to the FloatToCobol (see page F-46) function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to this function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pCobolToFloat
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     double Omitted,
     void **Parameter,
     int Repeat,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToFloat are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_IN
- PF_NO_NULL_POINTER
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OPTIONAL
- PF_REPEAT
- PF_ROUNDED
- PF_SILENT
- PF_VALUE_IF_OMITTED

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Omitted is the default value for omitted arguments if either of the PF_OPTIONAL or PF_VALUE_IF_OMITTED flags is set.

Parameter is a pointer to the address of the C parameter where the floating-point value will be stored.

Repeat is the repeat count if PF_REPEAT is set.

Size is the size of the C parameter.

CobolToGeneralString

CobolToGeneralString converts the COBOL argument to a null-terminated C string. For COBOL numeric arguments, this function has the same behavior as CobolToNumericString (see page F-31). For COBOL non-numeric arguments, this function has the same behavior as CobolToString (see page F-38). This function returns TRUE if it is successful and FALSE if there is an error.

By convention, this function should be called prior to the GeneralStringToCobol (see page F-48) function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to this function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pCobolToGeneralString
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     CONV_TABLE *ConvTable,
     int Flags,
     int Occurs,
     char *Omitted,
     void **Parameter,
     int Repeat,
     int Size,
     short Value1,
     short Value2);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

ConvTable is the internal conversion table allocated by ConversionStartup (see page F-42).

Flags modify the behavior of the conversion. The flags available for CobolToGeneralString are the union of the flags for CobolToNumericString and CobolToString. Some flags, such as PF_LEADING_MINUS, are ignored for non-numeric strings. Other flags, such as PF_LEADING_SPACES are ignored for numeric strings. Valid flag values (see Table F-2 on page F-5) for CobolToGeneralString are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_IN
- PF_LEADING_MINUS
- PF_LEADING_SIGN
- PF_LEADING_SPACES
- PF_LEADING_VALUE
- PF_NO_NULL_POINTER
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OPTIONAL
- PF_REPEAT
- PF_ROUNDED
- PF_SILENT
- PF_SIZE
- PF_TRAILING_CREDIT
- PF_TRAILING_DEBIT
- PF_TRAILING_MINUS
- PF_TRAILING_SIGN
- PF_TRAILING_SPACES
- PF_TRAILING_VALUE
- PF_VALUE_IF_OMITTED

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Omitted is the default value for omitted arguments if either of the PF_OPTIONAL or PF_VALUE_IF_OMITTED flags is set.

Parameter is the address of the C pointer where the address of the string will be stored. It may also be the address of an array of string values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Size is the conversion buffer length override when the PF_SIZE flag is set. If the PF_SIZE flag is not set, the default conversion buffer length is the greater of one more than the length of the COBOL argument and four more than the digit length of the COBOL argument. The digit length of a COBOL argument is the sum of the number of 9 and P symbols used in its PICTURE character-string.

Value1 is the strip/fill character value if the PF_LEADING_VALUE flag is set.

Value2 is the strip/fill character value if the PF_TRAILING_VALUE flag is set.

CobolToInteger

CobolToInteger converts the COBOL numeric argument to a C integer value. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, this function should be called prior to the IntegerToCobol (see page F-52) function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to this function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pCobolToInteger
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     long Omitted,
     void **Parameter,
     int Repeat,
     int Scale,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToInteger are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_IN
- PF_INTEGER_ONLY
- PF_NO_NULL_POINTER
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OPTIONAL
- PF_REPEAT
- PF_ROUNDED
- PF_SCALED
- PF_SILENT
- PF_UNSIGNED
- PF_VALUE_IF_OMITTED

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Omitted is the default value for omitted arguments if either of the PF_OPTIONAL or PF_VALUE_IF_OMITTED flags is set.

Parameter is a pointer to the address of the C parameter where the integer value will be stored. It may also be the address of an array of integer values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Scale is the scale value if the PF_SCALED flag is set. It represents the power of ten by which to multiply the COBOL argument.

Size is the size of the C parameter.

CobolToNumericString

CobolToNumericString converts the COBOL numeric argument to a null-terminated C string. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, this function should be called prior to the NumericStringToCobol (see page F-54) function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to this function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pCobolToNumericString
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     CONV_TABLE *ConvTable,
     int Flags,
     int Occurs,
     char *Omitted,
     void **Parameter,
     int Repeat,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

ConvTable is the internal conversion table allocated by ConversionStartup (see page F-42).

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToNumericString are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_IN
- PF_LEADING_MINUS
- PF_LEADING_SIGN
- PF_NO_NULL_POINTER
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OPTIONAL
- PF_REPEAT
- PF_ROUNDED
- PF_SILENT
- PF_SIZE
- PF_TRAILING_CREDIT
- PF_TRAILING_DEBIT
- PF_TRAILING_MINUS
- PF_TRAILING_SIGN
- PF_VALUE_IF_OMITTED

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Omitted is the default value for omitted arguments if either of the PF_OPTIONAL or PF_VALUE_IF_OMITTED flags is set.

Parameter is the address of the C pointer where the address of the string will be stored. It may also be the address of an array of string values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Size is the conversion buffer length override when the PF_SIZE flag is set. If the PF_SIZE flag is not set, the default conversion buffer length is the greater of one more than the length of the COBOL argument and four more than the digit length of the COBOL argument. The digit length of a COBOL argument is the sum of the number of 9 and P symbols used in its PICTURE character-string.

CobolToPointerAddress

CobolToPointerAddress obtains the effective address of the COBOL pointer argument by adding its offset and base address components. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolToPointerAddress
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void **Parameter);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value for CobolToPointerAddress is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C pointer where the effective address of the COBOL pointer argument will be stored.

CobolToPointerBase

CobolToPointerBase obtains the base address component of the COBOL pointer argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolToPointerBase
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void **Parameter);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToPointerBase are: PF_IN and PF_SILENT.

Parameter is the address of the C pointer where the base address component of the COBOL pointer argument will be stored.

CobolToPointerLength

CobolToPointerLength obtains the effective length of the COBOL pointer argument by subtracting its offset component from its size component. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolToPointerLength
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The only valid flag value for CobolToPointerLength is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the effective length of the COBOL pointer argument will be stored.

Size is the size of the C parameter.

CobolToPointerOffset

CobolToPointerOffset obtains the offset component of the COBOL pointer argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolToPointerOffset
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void **Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToPointerOffset are: PF_IN and PF_SILENT.

Parameter is a pointer to the address of the C parameter where the offset component of the COBOL pointer argument will be stored.

Size is the size of the C parameter.

CobolToPointerSize

CobolToPointerSize obtains the size component of the COBOL pointer argument. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pCobolToPointerSize
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void **Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToPointerSize are: PF_IN and PF_SILENT.

Parameter is a pointer to the address of the C parameter where the size component of the COBOL pointer argument will be stored.

Size is the size of the C parameter.

CobolToString

CobolToString converts the COBOL non-numeric argument to a null-terminated C string. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, this function should be called prior to the StringToCobol (see page F-59) function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to this function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pCobolToString
   (short ArgCount,
    short ArgNumber,
    struct ARGUMENT_ENTRY Arguments[],
    CONV_TABLE *ConvTable,
    int Flags,
    int Occurs,
    char *Omitted,
    void **Parameter,
    int Repeat,
    int Size,
    short Value1,
    short Value2);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

ConvTable is the internal conversion table allocated by ConversionStartup (see page F-42).

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for CobolToString are:

- PF_IN
- PF_LEADING_SPACES
- PF_LEADING_VALUE
- PF_NO_NULL_POINTER
- PF_OCCURS
- PF_OPTIONAL
- PF_REPEAT
- PF_SILENT
- PF_SIZE
- PF_TRAILING_SPACES
- PF_TRAILING_VALUE
- PF_VALUE_IF_OMITTED.

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Omitted is the default value for omitted arguments if either of the PF_OPTIONAL or PF_VALUE_IF_OMITTED flags is set.

Parameter is the address of the C pointer where the address of the string will be stored. It may also be the address of an array of string values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Size is the conversion buffer length override when the PF_SIZE flag is set. If the PF_SIZE flag is not set, the default conversion buffer length is one more than the length of the COBOL argument.

Value1 is the strip/fill character value if the PF_LEADING_VALUE flag is set.

Value2 is the strip/fill character value if the PF_TRAILING_VALUE flag is set.

CobolWindowsHandle

CobolWindowsHandle obtains the Windows handle of the current COBOL CALL. This function returns TRUE if it is successful and FALSE if there is an error.

Note The CobolWindowsHandle function is one of the trivial conversion functions described on page I-7.

Calling Sequence

```
int _rmdll_RtCall->pCobolWindowsHandle
    (int Flags,
     void *Parameter,
     int Size,
     HWND WindowsHandle);
```

Flags modify the behavior of the conversion. The only valid flag value for CobolWindowsHandle is PF_SILENT (see Table F-2 on page F-5).

Parameter is the address of the C parameter where the Windows handle will be stored.

Size is the size of the C parameter.

WindowsHandle is the Windows handle for the current COBOL CALL. This attribute is not available on UNIX platforms as it can cause compilation errors.

ConversionCleanup

ConversionCleanup must be called just prior to returning to the calling COBOL program. It releases all memory that has been allocated by other conversion functions.

Note ConversionCleanup must be called for every exit back to the calling COBOL program when the C function has multiple return paths.

Calling Sequence

```
void _rmdll_RtCall->pConversionCleanup
    (short ArgCount,
     CONV_TABLE *ConvTable);
```

ArgCount is the argument count for the current COBOL CALL.

ConvTable is the internal conversion table allocated by ConversionStartup (see page F-42).

ConversionStartup

ConversionStartup must be called once at the beginning of the C function called from COBOL and should precede all calls to other conversion functions. It allocates a block of memory for each COBOL argument (based on the value of *ArgCount*). This block contains information that must be preserved between calls to other conversion functions. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pConversionStartup
    (short ArgCount,
     CONV_TABLE **ConvTable,
     char *Name,
     short Version);
```

ArgCount is the argument count for the current COBOL CALL.

ConvTable is the address of a C pointer where the address of the internal conversion table will be stored.

Name is name of the C function that was called by the COBOL program.

Version is the minimum version of the CodeBridge Library that can provide all the conversion and validation features required by the C function. To specify that the CodeBridge Library for RM/COBOL version 7.0 is required, the value for *Version* should be 0x700.

DiagnosticMode

DiagnosticMode controls the display of error messages during execution. If *Flag* contains the value, DF_SILENT, no error messages will be displayed. If *Flag* contains the value, DF_VERBOSE, error messages will always be displayed. If *Flag* contains the value, DF_NORMAL, the display of error messages is governed by the PF_SILENT flag in each call to the CodeBridge Library.

Note DiagnosticMode has global scope. It affects all conversion and validation calls until another DiagnosticMode call is made. Before the first call to DiagnosticMode, the display of error messages is governed by the PF_SILENT flag in each call to the CodeBridge Library as if DiagnosticMode had been called with the DF_NORMAL flag value.

Calling Sequence

```
void _rmdll_RtCall->pDiagnosticMode
    (short Flag);
```

Flag modifies the display of the error message. Valid flag values for DiagnosticMode are the following:

Name	Value	Description
DF_SILENT	-1	Diagnostic messages are never displayed.
DF_NORMAL	0	Diagnostic messages are displayed unless the PF_SILENT flag is set in the CodeBridge Library function call.
DF_VERBOSE	1	Diagnostic messages are always displayed.

EffectiveLength

EffectiveLength obtains the length of the C string after conversion from the COBOL argument. This includes removal of leading and/or trailing characters. The value is the same as the value that would be returned by the C library function, strlen. This function returns TRUE if it is successful and FALSE if there is an error.

Note The EffectiveLength function may be used only in combination with one of the input string functions: CobolToGeneralString (see page F-27), CobolToNumericString (see page F-31), or CobolToString (see page F-38). *ArgNumber* must have the same value in the EffectiveLength function call and the corresponding input string function call. The call to EffectiveLength may precede or follow the call to the corresponding input string function.

Calling Sequence

```
int _rmdll_RtCall->pEffectiveLength
    (short ArgCount,
     short ArgNumber,
     CONV_TABLE, *ConvTable,
     int Flags,
     int Occurs,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

ConvTable is the internal conversion table allocated by ConversionStartup (see page F-42).

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for EffectiveLength are: PF_OCCURS and PF_SILENT.

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Parameter is the address of the C parameter where the effective length will be stored.

Size is the size of the C parameter.

FloatToCobol

FloatToCobol converts from a C floating-point value to the COBOL numeric argument. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, the CobolToFloat function (see page F-25) should be called prior to this function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to the CobolToFloat function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pFloatToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     void *Parameter,
     int Repeat,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for FloatToCobol are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OUT
- PF_REPEAT
- PF_RETURN_VALUE
- PF_ROUNDED
- PF_SILENT

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Parameter is the address of the C parameter. It may also be the address of an array of floating-point values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Size is the size of the C parameter.

GeneralStringToCobol

GeneralStringToCobol converts a null-terminated C string to the COBOL argument. For COBOL numeric arguments, this function has the same behavior as NumericStringToCobol (see page F-54). For COBOL non-numeric arguments, this function has the same behavior as StringToCobol (see page F-59). This function returns TRUE if it is successful and FALSE if there is an error.

By convention, the CobolToGeneralString function (see page F-27) should be called prior to this function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to the CobolToGeneralString function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pGeneralStringToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     void *Parameter,
     int Repeat,
     int Size,
     short Value1,
     short Value2);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. The flags available for GeneralStringToCobol are the union of the flags for NumericStringToCobol and StringToCobol. Some flags, such as PF_LEADING_MINUS, are ignored for non-numeric strings. Other flags, such as PF_LEADING_SPACES are ignored for numeric strings. Valid flag values (see Table F-2 on page F-5) for GeneralStringToCobol are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_IN
- PF_LEADING_SPACES
- PF_LEADING_VALUE
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OUT
- PF_REPEAT
- PF_RETURN_VALUE
- PF_ROUNDED
- PF_SILENT
- PF_SIZE
- PF_TRAILING_SPACES
- PF_TRAILING_VALUE

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Parameter is the address of the C parameter. It may also be the address of an array of string values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Size is the conversion buffer length override when the PF_SIZE flag is set. If the PF_SIZE flag is not set, the default conversion buffer length is the greater of one more than the length of the COBOL argument and four more than the digit length of the COBOL argument. The digit length of a COBOL argument is the sum of the number of 9 and P symbols used in its PICTURE character-string. The setting of the PF_SIZE flag and the value of the *Size* parameter must be the same as specified in the call to CobolToGeneralString (described on page F-27) for the same argument.

Value1 is the strip/fill character value if the PF_LEADING_VALUE flag is set.

Value2 is the strip/fill character value if the PF_TRAILING_VALUE flag is set.

GetCallerInfo

GetCallerInfo obtains information about the calling COBOL program. Such information is particularly useful in error messages because it helps identify the offending CALL statement. This function returns a pointer to a structure that contains the information about the calling program.

Calling Sequence

```
CALLER_INFO* _rmdll_RtCall->pGetCallerInfo();
```

The function has no arguments.

The structure pointed to by the return value is described by a type definition in the supplied header file **rtcallbk.h**, which is included by the supplied header file **cbridge.h**. For reference, the structure is as follows:

```
typedef struct tagCallerInfo
{
  BIT16  Version;           /* structure version; 0x0001 is first version  */
  BIT16  Flags;             /* flags; see #define CIF_... below         */
  char  *ProgramLocation;   /* line number of CALL or
                               segment/offset of statement after CALL */
  char  *ProgramName;       /* calling program name */
  char  *ProgramFileName;   /* calling program object file name
                               (including pathname) */
  char  *ProgramDateTime;   /* calling program date and time compiled */
}  CALLER_INFO;
```

The Flags fields in the CALLER_INFO structure have the following meanings (as defined in **rtcallbk.h**):

```
#define CIF_LOCATION_ADDRESS  0x8000    /* indicates ProgramLocation
                                           is segment/offset */
#define CIF_NESTED_PROGRAM    0x4000    /* indicates calling program
                                           is a nested program */
```

The CIF_LOCATION_ADDRESS flag is set when the calling program was compiled with the Q Compile Command Option, thus making line numbers unavailable at runtime. In this case, the ProgramLocation entry points to a string giving the segment/offset of the return location for the CALL statement as shown in the DEBUG column of a compilation listing. When the flag is not set, the ProgramLocation entry points to a string giving the source line number of the CALL statement.

Note There is no global or parameter attribute that can be placed in a template file to cause the CodeBridge Builder to produce a call to GetCallerInfo. The CodeBridge Library will automatically call GetCallerInfo when displaying any error messages caused by conversion errors. A user-written function, whether or not it uses other CodeBridge Library calls, may call GetCallerInfo to add this information to its own error messages.

IntegerToCobol

IntegerToCobol converts from a C integer value to the COBOL numeric argument. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, the CobolToInteger function (see page F-29) should be called prior to this function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to the CobolToInteger function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pIntegerToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     void *Parameter,
     int Repeat,
     int Scale,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for IntegerToCobol are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OUT
- PF_REPEAT
- PF_RETURN_VALUE
- PF_ROUNDED
- PF_SCALED
- PF_SILENT
- PF_UNSIGNED

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Parameter is the address of the C parameter. It may also be the address of an array of integer values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Scale is the scale value if the PF_SCALED flag is set. It represents the power of ten by which to divide the C parameter.

Size is the size of the C parameter.

NumericStringToCobol

NumericStringToCobol converts a null-terminated C string to the COBOL numeric argument. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, the CobolToNumericString function (see page F-31) should be called prior to this function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to the CobolToNumericString function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pNumericStringToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     void *Parameter,
     int Repeat,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for NumericStringToCobol are:

- PF_ASSERT_SIGNED
- PF_ASSERT_UNSIGNED
- PF_NO_SIZE_ERROR
- PF_OCCURS
- PF_OPTIONAL
- PF_OUT
- PF_REPEAT
- PF_RETURN_VALUE
- PF_ROUNDED
- PF_SILENT
- PF_SIZE

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Parameter is the address of the C parameter. It may also be the address of an array of string values if the PF_OCCURS flag is set.

Size is the conversion buffer length override when the PF_SIZE flag is set. If the PF_SIZE flag is not set, the default conversion buffer length is the greater of one more than the length of the COBOL argument and four more than the digit length of the COBOL argument. The digit length of a COBOL argument is the sum of the number of 9 and P symbols used in its PICTURE character-string. The setting of the PF_SIZE flag and the value of the *Size* parameter must be the same as specified in the call to CobolToNumericString (described on page F-31) for the same argument.

PointerBaseToCobol

PointerBaseToCobol modifies the COBOL pointer argument. The contents of the C pointer are moved to the base address component. If the PF_POINTER_MAX_SIZE flag is set, binary ones are moved to the size component. If the PF_POINTER_RESET_OFFSET flag is set, a value of 0 is moved to the offset component. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pPointerBaseToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void **Parameter);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for PointerBaseToCobol are:

- PF_OUT
- PF_POINTER_MAX_SIZE
- PF_POINTER_RESET_OFFSET
- PF_RETURN_VALUE
- PF_SILENT

Parameter is the address of the C pointer.

PointerOffsetToCobol

PointerOffsetToCobol modifies the COBOL pointer argument. The contents of the C parameter are moved to the offset component. If the PF_POINTER_MAX_SIZE flag is set, binary ones are moved to the size component. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pPointerOffsetToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for PointerOffsetToCobol are:

- PF_OUT
- PF_POINTER_MAX_SIZE
- PF_RETURN_VALUE
- PF_SILENT

Parameter is the address of the C parameter.

Size is the size of the C parameter.

PointerSizeToCobol

PointerSizeToCobol modifies the COBOL pointer argument. The contents of the C parameter are moved to the size component. If the PF_POINTER_RESET_OFFSET flag is set, a value of zero is moved to the offset component. This function returns TRUE if it is successful and FALSE if there is an error.

Calling Sequence

```
int _rmdll_RtCall->pPointerSizeToCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     void *Parameter,
     int Size);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for PointerSizeToCobol are:

- PF_OUT
- PF_RETURN_VALUE
- PF_POINTER_RESET_OFFSET
- PF_SILENT

Parameter is the address of the C parameter.

Size is the size of the C parameter.

StringToCobol

StringToCobol converts a C null-terminated string to the COBOL non-numeric argument. This function returns TRUE if it is successful and FALSE if there is an error.

By convention, the CobolToString function (see page F-38) should be called prior to this function for the same argument number. Do not set the PF_IN flag for output-only conversions. Because the call to the CobolToString function may perform memory management operations that are not needed for output-only conversions, this call may be omitted.

Calling Sequence

```
int _rmdll_RtCall->pStringtoCobol
    (short ArgCount,
     short ArgNumber,
     struct ARGUMENT_ENTRY Arguments[],
     int Flags,
     int Occurs,
     void *Parameter,
     int Repeat,
     int Size,
     short Value1,
     short Value2);
```

ArgCount is the argument count for the current COBOL CALL.

ArgNumber is -1 for the argument in the GIVING (RETURNING) phrase or the zero-based number of an argument from the USING phrase.

Arguments is the address of the argument descriptor array.

Flags modify the behavior of the conversion. Valid flag values (see Table F-2 on page F-5) for StringToCobol are:

- PF_LEADING_SPACES
- PF_LEADING_VALUE
- PF_OCCURS
- PF_OUT
- PF_REPEAT
- PF_RETURN_VALUE
- PF_SILENT
- PF_SIZE
- PF_TRAILING_SPACES
- PF_TRAILING_VALUE

Occurs is the array size if the C parameter is an array. A value of zero may be specified if the C parameter is a scalar; negative values for the *Occurs* parameter are allowed, but are treated as equivalent to zero. If the value is greater than 1, we recommend the PF_OCCURS flag be set, although it is documentary only.

Parameter is the address of the C parameter. It may also be the address of an array of string values if the PF_OCCURS flag is set.

Repeat is the repeat count if the PF_REPEAT flag is set.

Size is the conversion buffer length override when the PF_SIZE flag is set. If the PF_SIZE flag is not set, the default conversion buffer length is one more than the length of the COBOL argument. The setting of the PF_SIZE flag and the value of the *Size* parameter must be the same as specified in the call to CobolToString (described on page F-38) for the same argument.

Value1 is the strip/fill character value if the PF_LEADING_VALUE flag is set.

Value2 is the strip/fill character value if the PF_TRAILING_VALUE flag is set.

Appendix G: Non-COBOL Subprogram Internals for Windows

This appendix describes the internal details of how a non-COBOL subprogram is called from an RM/COBOL program running under 32-bit Windows. While it is possible to write non-COBOL subprograms that directly use this information to handle COBOL argument conversions, it is highly recommended that CodeBridge be used for this purpose instead. This appendix also provides information on preparing a non-COBOL subprogram for use by an RM/COBOL program on 32-bit Windows. (For additional information, see the "CALL Statement" section of Chapter 6, *Procedure Division Statements*, in the *RM/COBOL Language Reference Manual*.)

Note The information presented here assumes a working knowledge of the C programming language. The material in Appendix C, *Useful C Information*, is not comprehensive enough to provide this necessary background.

C Subprograms

To modify or write a C subprogram that can be called from the RM/COBOL runtime system requires an understanding of the fundamental tasks involved. First, in order to access C language subprograms from the RM/COBOL runtime system, you must build a dynamic link library (DLL), normally referred to as an "optional support module." (For more information on DLLs and optional support modules, see Appendix D, *Support Modules (Non-COBOL Add-Ons)*, of the *RM/COBOL User's Guide*.)

Methods of Using Non-COBOL Subprograms

Two methods of using non-COBOL subprograms are supported:

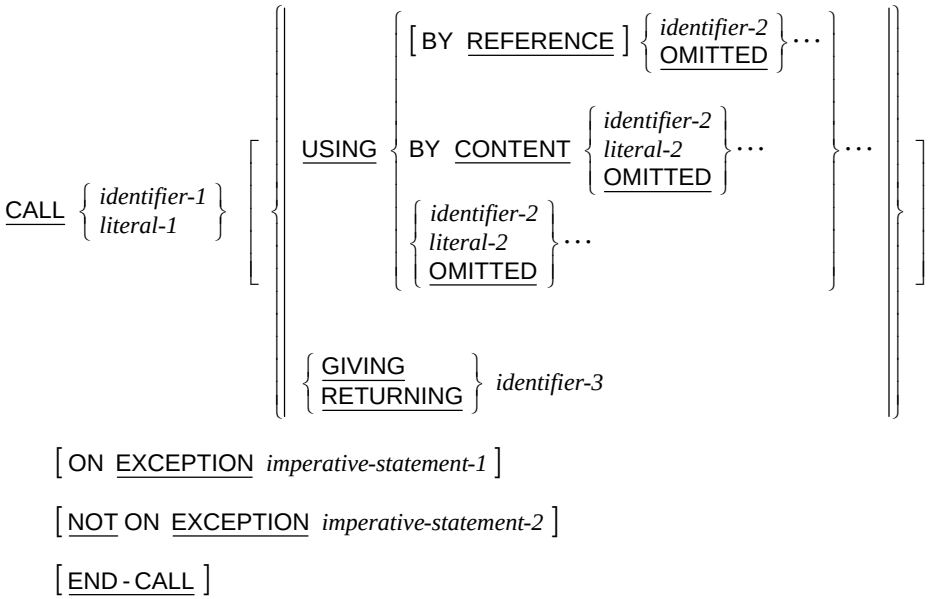
1. A single subprogram can be dynamically loaded by the Runtime Command (**runcobol**) when that subprogram is called from the RM/COBOL program. The subprogram remains resident until canceled by the RM/COBOL program or until the end of the run unit. This method is sometimes referred to as the "call-by-filename" method since the program is loaded because its file name matches the called program name.
2. One or more subprograms can be linked into a non-COBOL subprogram library (DLL) and loaded by the Runtime Command upon run unit initialization. The library is loaded either because it is referenced in an L Runtime Command Option or because it is present in the **rmautold** subdirectory of the execution directory. The library remains resident until the end of the run unit.

Calling C Subprograms from COBOL

This section describes the COBOL CALL syntax and explains how a C programmer can write a subprogram that can be called from RM/COBOL. The COBOL CALL statement explains the use of the non-COBOL subprogram from the COBOL programmer's perspective while the other topics describe the structures and the function prototype that the C programmer needs to understand.

COBOL CALL Statement

The syntax for the Format 2 CALL statement in the RM/COBOL program is as follows:



The value of the contents of the data item specified by *identifier-1* or the value of *literal-1* is the program-name of the subprogram to be called.

identifier-2 or *literal-2* are one or more actual arguments to be passed to the called program. If the BY CONTENT phrase applies to an argument, a temporary copy of the item is passed, thus preventing the subprogram from modifying the original item.

identifier-3 is an actual argument to be passed to the called program for the purposes of returning a result to the calling program.

The RM/COBOL runtime system locates the subprogram with the program-name specified by *literal-1* or the value of the data item referenced by *identifier-1*. See the discussion of "Subprogram Loading" in Chapter 8, *RM/COBOL Features*, of the *RM/COBOL User's Guide*, for additional information on locating subprograms.

The subprogram also must be a dynamic link library file (**.dll**) and is loaded with the Windows LoadLibrary function.

C Subprogram Name Table Structure

The RM/COBOL runtime system can locate the C subprograms only if their names are exported and either (1) their names appear in the subprogram name table, or (2) the DLL contains an .EDATA section. The subprogram name table is an array of name table entries. Each name table entry is a C structure that is defined as follows:

```
typedef struct EntryTable
{
    char       *EntryPointCobolName;   /* name of subroutine as in call */
    int        (*EntryPointAddress)(); /* entry point address */
    char       *EntryPointName;        /* name of entry point in object */
} ENTRYTABLE;
```

Character-strings must be null terminated. The last array entry must consist of NULLs. The name of the subprogram name table must be **RM_EntryPoints** and this name must be exported, but an .EDATA section is not required in the DLL when the subprogram name table exists. When the subprogram name table exists, any .EDATA section in the DLL, if present, is ignored.

The RM/COBOL runtime system does not use the EntryPointAddress entry in this structure. Instead, the EntryPointName entry is used to find the procedure address for the procedure that has the given name. Thus, each value supplied in an EntryPointName entry must match that of an exported symbol in the DLL. When the DLL is loaded, the runtime system looks up the procedure address for each entry using the supplied name; if the name is not found, an error occurs and the runtime system is terminated with an appropriate message. The exported symbol may be different than the function name in the C source when a **.def** file is used during linking since **.def** files can contain an exports list that specifies different names to be exported for the C functions.

RM_EntryPoints is one of the predefined symbols in an optional support module. For complete information about all of the predefined symbols, see "Special Entry Points for Support Modules" on page G-12.

Note The ENTRYTABLE typedef is defined in **rmc85cal.h**, which is provided with RM/COBOL systems. This header file should be included (with a preprocessor **#include** statement) in C source that defines COBOL-callable subprograms. Inclusion of this header file will also cause **RM_EntryPoints** symbol to be exported. Other header files (**rtarg.h**, **standdef.h**, and **rmport.h**) are referenced by **rmc85cal.h**. These files are also provided with RM/COBOL systems. When using CodeBridge Library functions, it is generally sufficient to include **cbridge.h**, which includes these other header files.

Example

```
ENTRYTABLE      RM_EntryPoints[] =
{
    {"SUB1NAME",      sub1,      "sub1"},
    {"SUB2NAME",      sub2,      "sub2"},
    {NULL,            NULL,      NULL  }
};
```

In this example, "SUB1NAME" and "SUB2NAME" are the COBOL-callable program-names, sub1 and sub2 are the addresses of the C subprograms (functions), and "sub1" and "sub2" are the exported names of the C subprograms (functions). In this example, it is assumed that a **.def** file, if used, does not rename the C functions in the exports list.

Parameters Passed to the C Subprogram

The RM/COBOL runtime system passes six parameters on the stack to the called C subprogram. The following is a sample COBOL-callable C subprogram function prototype:

```
RM_DLLEXPORT int RM_CDECL sub1
   (
      char                 *name,           /* param1 */
      unsigned short        arg_count,      /* param2 */
      ARGUMENT_ENTRY        arg_vector[],   /* param3 */
      unsigned short        initial_state   /* param4 */
      RM_HWND               window_handle,  /* param5 */
      RUNTIME_CALLS_TABLE   callbacktable   /* param6 */
   );
```

The six parameters are described as follows:

1. Pointer to the called program-name, which is a null-terminated ASCII string containing the name used by the run unit to identify the called subprogram. The called program-name is always uppercase-only, regardless of the case of the name in the calling COBOL program.
2. Argument count, which is the number of arguments, including arguments explicitly specified with the OMITTED keyword, specified in the USING phrase of the CALL statement. The argument in the GIVING (RETURNING) phrase, if specified, is not included in the count.
3. Pointer to the argument array, which is an array of structures describing each of the actual arguments passed in the GIVING (RETURNING) and USING phrases of the

CALL statement. The structure of an argument description entry is described in "COBOL Argument Entry Structure for C" on page G-7 and is defined in the **rmc85cal.h** header file, which is provided with RM/COBOL systems.

4. Initial state flag, which contains a zero to indicate that the subprogram is being called for the first time in the run unit or the first time since a CANCEL statement has been executed for the subprogram name. A nonzero value indicates that the subprogram should remain in its last used state. It is the responsibility of the called subprogram (rather than the runtime system) to examine the initial state flag and decide which variables need to be reinitialized. In any case, on each call, all C automatic variables are reallocated on the stack without being initialized to any particular value (that is, C automatic variables have arbitrary values).

5. Windows handle of the calling program window (runtime window), which is needed for some calls to the Windows Application Programming Interface (API).

6. Pointer to the runtime call-back table, which is a structure that contains the size of the table, the version number of the table, and a list of subprogram addresses in the runtime. The CodeBridge Builder uses the call-back table to obtain access to some utility subprograms in the runtime system. The description of this table is available in **cbridge.h**, a header file provided with CodeBridge. The table is named RUNTIME_CALLS_TABLE.

Note The fifth and sixth parameters are optional. Although the runtime system will always pass these values, the called subprogram does not have to declare them. The prototype for the called function may omit the sixth or both the fifth and sixth parameters. The runtime call-back table is required if the subprogram uses any of the CodeBridge Library functions.

The called subprogram must set an integer return value before returning control to the runtime system. A value of RM_FND (defined as 0 in **rmc85cal.h**) indicates that the subprogram was found and that the runtime should continue executing the COBOL program. A value of RM_STOP (defined as 1 in **rmc85cal.h**) indicates that the subprogram terminated because of a fatal error, such as incorrect parameters, and that the runtime should terminate the run unit. An explicit return statement should be used to set the return value since otherwise the run unit might be unintentionally terminated. The subprogram must not terminate with the system function **exit()**, since the runtime could not do an orderly shutdown of the run unit in this case.

The argument entry table (arg_vector) contains descriptions of the actual arguments specified in the CALL statement. The arg_vector[0] entry describes the first actual argument in the USING phrase of the CALL statement. The arg_vector[arg_count - 1] entry describes the last actual argument in the USING phrase of the CALL statement. The arg_vector[-1] entry describes the argument specified in the GIVING (RETURNING) phrase of the CALL statement. If the GIVING (RETURNING) phrase is omitted from

the CALL statement, or if any actual argument is specified as OMITTED in the USING phrase of the CALL statement, the corresponding arg_vector entry contains a type value 32 (OMITTED, as shown in Table G-1 on page G-8) and the remaining fields are zero.

C subprograms that access the GIVING argument in arg_vector[-1] will function correctly only for RM/COBOL version 7 (or later) runtimes because prior runtimes did not make a GIVING argument entry available in arg_vector[-1]. A subprogram that uses the GIVING argument should verify that it is available by use of the version number in the runtime call-back table, the address of which is provided by the sixth parameter to the subprogram. The version number must be 0x0700 or greater for a GIVING argument to be available.

COBOL Argument Entry Structure for C

To a subprogram written in C, an argument entry is defined by the following structure, which is included in the **rmc85cal.h** header file:

```
typedef struct ArgumentEntry
{
  char          *a_address;   /* pointer to start of argument */
  unsigned long  a_length;    /* length of argument */
  short          a_type;      /* type of argument (RM/COBOL data type) */
  char           a_digits;    /* digit count (0-30) */
  char           a_scale;     /* implied decimal location (signed) */
  char          *a_picture;   /* pointer to encoded edit picture */
} ARGUMENT_ENTRY;
```

a_address specifies the lowest addressed byte of the argument.

a_length specifies the number of bytes allocated to the argument.

a_type specifies the RM/COBOL data type as a number from Table G-1 (see page G-8). Names for these type numbers are defined in **rmc85cal.h**. (For an explanation of the data type abbreviations and a description of the RM/COBOL data types listed in Table G-1, see Table 9-2 in Chapter 9, *Debugging*, and Appendix C, *Internal Data Formats*, of the *RM/COBOL User's Guide*.)

a_digits specifies the actual number of digits in a numeric data item (where the type of argument is in the range 0 through 12). It is set to zero for nonnumeric data items.

a_scale specifies the power of 10 by which the digits in a numeric data item (where the type of argument is in the range 0 through 12) must be multiplied to obtain the numeric value of the data item. The power of 10 is represented as a signed, 2's complement number. It is set to zero for nonnumeric data items.

a_picture specifies the lowest addressed byte of the encoded picture for edited items (type of argument equals 0, 20 or 21). It is set to zero for all other types.

Table G-1: RM/COBOL Data Types as Numbers

Type Number	RM/COBOL Data Type	Type Number	RM/COBOL Data Type
0	NSE	16	ANS
1	NSU	17	ANS (justified right)
2	NTS	18	ABS
3	NTC	19	ABS (justified right)
4	NLS	20	ANSE
5	NLC	21	ABSE
6	NCS	22	GRP (fixed length)
7	NCU	23	GRPV (variable length)
8	NPP	25	PTR
9	NPS	26	NBSN
10	NPU	27	NBUN
11	NBS	32	OMITTED
12	NBU		

Note The data type GRPV (23) does not occur when C$CARG is called with the formal argument name or when C$DARG is called with an actual argument number that corresponds to an argument that is a variable-length group. In all cases, RM/COBOL passes variable-length group actual arguments as if they were a fixed-length group of the maximum length. (See Appendix F, *Subprogram Library*, of the *RM/COBOL User's Guide*.)

Preparing C Subprograms

One or more dynamic link libraries (DLLs) may be loaded and called by the RM/COBOL runtime system. The DLL may be specified on the command line by using the L Runtime Command Option, described in the section “Runtime Command Options” in Chapter 7, *Running*, of the *RM/COBOL User's Guide*. DLL files may also be placed in the **rmautold** subdirectory of the execution directory for automatic loading when the runtime system is started. The runtime system reads the DLL, locates the entry points, and makes each entry point available to be called as a subprogram.

If a program-name used in a CALL statement cannot be resolved as a COBOL routine and is not found in any already loaded non-COBOL library, a search is made for a file with that name and an extension of **.dll**. If such a file is found, it is loaded and one of the following occurs:

- If the DLL exports either of the symbols **RM_EnumEntryPoints** or **RM_EntryPoints**, then the first specified entry point is called. For a definition of these symbols, see "Special Entry Points for Support Modules" that begins on page G-12. Any additional entry points that these symbols may define are ignored when the DLL is loaded by this method;
- Otherwise, if the DLL contains an .EDATA section that specifies an entry point exported as nonresident ordinal one, then that entry point is called. Any other exported entry points are ignored when the DLL is loaded by this method;
- Otherwise, a procedure error 204 occurs.

This method of loading a DLL is sometimes referred to as "call-by-filename" to contrast it with the method of calling a program-name defined in a library loaded because an L Runtime Command Option refers to it or the presence of the library in the **rmautold** subdirectory of the execution directory.

Note Old 16-bit DLLs are still supported for backward compatibility, on Windows 9x-class operating systems; however, some of the new features discussed in this appendix do not apply to 16-bit DLLs. Specifically, the special entry points, described in "Special Entry Points for Support Modules" on page G-12, are not recognized in a 16-bit DLL.
To make use of the special entry points, the DLL must be rewritten as a 32-bit DLL. A 16-bit DLL may be loaded by any of the three methods discussed in this appendix, including its being present in the **RmAutoLd** subdirectory of the execution directory.

The following steps may be used to prepare a non-COBOL subprogram for calls from a COBOL program (compiler-specific comments are included):

1. Generate a non-COBOL source file(s) containing one or more subprograms that will serve as entry points for the COBOL program. Entry points that are normally associated with a DLL, such as LibMain (or DllMain or DllEntryPoint), should be defined and may contain minimal code. These entry points and the additional entry points that you define must be exported in the manner described for your compiler.

   Use C calling conventions (instead of PASCAL conventions). Stack-based parameter passing also should be used.

2. Translate the source file into a valid object file (**.obj**) with your compiler.
3. Create the dynamic link library using the linker in your C development system. Use linker options to assign an ordinal value of one to an entry point. The

RM/COBOL runtime system will associate the DLL filename with entry point one. The procedures in the DLL are now ready to be called as a subprogram from RM/COBOL.

Note While some C compilers produce case-insensitive entry point names, others produce case-sensitive entry point names. In addition, some C compilers may pre-pend or append an underscore character to the entry point name.

Parameters are passed to the DLL as described in "Parameters Passed to the C Subprogram" on page G-5.

The following code sequences illustrate how a COBOL-callable DLL may be written in C. Include the **standdef.h** header file (provided by Liant) to access RM/COBOL standard definitions. On Windows systems, inclusion of **standdef.h** will cause inclusion of the Microsoft **windows.h** file, which provides access to Windows operating system functions such as MessageBox(). Define RMLittleEndian with a value of 1 for the Intel 80x86 architecture. Include the **rmc85cal.h** header file to obtain ARGUMENT_ENTRY structure definition, various type definitions, and LDLONG, LDSHORT, STLONG, STSHORT macros. Include the **cbridge.h** header file if the CodeBridge Library is used by the subprogram. Since **cbridge.h** includes **standdef.h** and **rmc85cal.h**, it is not necessary to include these header files when **cbridge.h** is included.

The following is a sample RM/COBOL-callable DLL file written in C, named **msgbox.c**.

```
#include "standdef.h"

#define RMLittleEndian 1

#include "rmc85cal.h"

RM_DLLEXPORT int RM_CDECL
MsgBox(char *Name, unsigned short ArgCount, ARGUMENT_ENTRY *ArgEntry,
       unsigned short State)
{
    short sButton;
    long lButton;
    char Buf[64];
    short i;
    char *p;
    short n;

    if (ArgCount != 2)
        return (RM_STOP);

    /* -- check arguments */
    switch (ArgEntry[0].a_type)
    {
        /* -- various displayable types */
        case RM_ANS:    case RM_ANSR:
        case RM_ABS:    case RM_ABSR:
        case RM_NSE:
        case RM_GRPF:
            break;

        default:
            return (RM_STOP);
    }

    switch (ArgEntry[1].a_type)
    {
        /* -- only return binary types size 2 or 4 */
        case RM_NBS: case RM_NBU:
            if ((ArgEntry[1].a_length == 2)
                || (ArgEntry[1].a_length == 4))
                break;

        default:
            return (RM_STOP);
    }

    p = ArgEntry[0].a_address;
    n = (short) ArgEntry[0].a_length;
    for (i = 0; i < n; i++)
        Buf[i] = *p++;
    Buf[i] = '\0';

    lButton =
    sButton = MessageBox(NULL, Buf, NULL, MB_YESNO |
                                          MB_ICONQUESTION |
                                          MB_SETFOREGROUND);

    /* -- return value in second argument */
    p = ArgEntry[1].a_address;
    if (ArgEntry[1].a_length == 4)
        STLONG (lButton, p);
    else if (ArgEntry[1].a_length == 2)
        STSHORT (sButton, p);
    return (RM_FND);

}
```

This sample DLL can be compiled using the 32-bit Microsoft Visual C++ compiler with the following command:

```
cl msgbox.c -Zp1 /link -out:msgbox.dll -dll -export:MsgBox,@1
  -section:.edata,IRD user32.lib
```

It also can be built using the 32-bit Watcom C compiler, version 10.6 or later, with the following command:

```
wclX86 -l=nt_dll -bd msgbox.c -"export MSGBOX.1=_MsgBox"
```

The following source fragments from a COBOL program could be used to call the DLL:

```
DATA DIVISION.

WORKING-STORAGE SECTION.
01  RETURN-BINARY   PIC 9(4)   Binary(2) Value Zero.
01  DISPLAY-TEXT    PIC X(24)  Value "Do you wish to continue?".

PROCEDURE DIVISION.
    CALL "MSGBOX" USING DISPLAY-TEXT RETURN-BINARY.
```

Special Entry Points for Support Modules

When the runtime system (or other RM/COBOL component) loads an optional support module, it looks for certain predefined symbols (entry points and variable names), and varies its actions based on the presence or absence of these symbols. One such variable name is **RM_EntryPoints** (discussed in "C Subprogram Name Table Structure" on page G-4). The example subprogram, **msgbox.c**, which is distributed with the RM/COBOL system, contains examples of all of these entry points and symbols, except for **RM_EnumEntryPoints**. This example can be used as a starting point when developing optional support modules for Windows.

The complete list of these special names is as follows:

- RM_AddOnBanner
- RM_AddOnCancelNonCOBOLProgram
- RM_AddOnInit
- RM_AddOnLoadMessage
- RM_AddOnTerminate
- RM_AddOnVersionCheck
- RM_EntryPoints and RM_EnumEntryPoints

Note On Windows, all these entry points are optional if the DLL is linked such that an .EDATA section is produced. If the DLL is linked without producing an .EDATA section, the **RM_EntryPoints** or **RM_EnumEntryPoints** symbols must be defined for there to be any COBOL callable routines in the DLL.

The following sections describe these entry points and special variables.

RM_AddOnBanner

This entry point, if present, should return a pointer to a character string. This character string will be displayed along with the runtime system banner message. The support module banner may be used to display any required copyright notice. The support module banner is displayed only if the K Option of the Runtime Command is not present.

Note The Windows runtime supports the "call-by-filename" loading of DLLs as described in "Methods of Using Non-COBOL Subprograms" on page G-2. For DLLs loaded in this manner, the RM_AddOnBanner entry point is not called and no banner is produced. The entry point is called and a banner is produced if the DLL is loaded because of the L Runtime Command Option or because the DLL is present in the **rmautold** subdirectory of the execution directory.

Function declaration for **RM_AddOnBanner**:

```
char*     RM_AddOnBanner(void);
```

RM_AddOnCancelNonCOBOLProgram

This entry point, if present, is called by the runtime system when a CANCEL verb is executed for a program-name that is defined in the optional support module. It allows the support module to do any cleanup actions that may be necessary. For example, this entry point might be specified to allow the support module to close any open files when the COBOL program cancels the associated non-COBOL subprogram. The program-name of the non-COBOL subprogram for which a CANCEL has been performed is passed as a parameter to the entry point.

Function declaration for **RM_AddOnCancelNonCOBOLProgram**:

```
void      RM_AddOnCancelNonCOBOLProgram(char* ProgramName);
```

RM_AddOnInit

This entry point, if present, is called to initialize the optional support module. All support modules will be initialized (if initialization is requested) before the runtime system begins executing the first COBOL program, except that DLLs loaded by the "call-by-filename" method (as described in "Methods of Using Non-COBOL Subprograms" on page G-2) will be initialized when they are loaded at the time they are referenced by a CALL statement.

The entry point should return zero to indicate successful initialization or a non-zero value to indicate that the support module initialization failed. If the initialization fails, the runtime system will display an appropriate message and then terminate.

Note If the support module determines that successful initialization is not possible, the support module should produce appropriate messages to allow the user to correct the problem.

The support module is passed the Runtime Command line arguments in the arguments Argc (the argument count) and Argv (the argument vector). The support module is also passed a pointer to the runtime call back table.

Function declaration for **RM_AddOnInit**:

```
int       RM_AddOnInit(int Argc,
                       char** Argv,
                       RUNTIME_CALLS_TABLE *pRtCall);
```

RM_AddOnLoadMessage

This entry point, if present, should return a pointer to a character string that is displayed along with the load messages of other optional support modules. These load messages allow the user to verify which support modules the runtime system has loaded. The message may contain text to identify the support module and, if desired, the version number or the build date. Load messages are displayed only if the V Runtime Command Option is present, the V=DISPLAY keyword-value pair is specified in the RUN-OPTION configuration record, or the RM_DYNAMIC_LIBRARY_TRACE environment variable
is defined.

If load messages are being displayed, the runtime system generates a load message consisting of the complete pathname for the support module regardless of whether the **RM_AddOnLoadMessage** entry point is defined or not defined in the support module. If the **RM_AddOnLoadMessage** entry point is defined, the returned string is appended to the pathname in this load message.

Note The Windows runtime supports the "call-by-filename" loading of DLLs as described in "Methods of Using Non-COBOL Subprograms" on page G-2. For DLLs loaded in this manner, the RM_AddOnLoadMessage entry point is not called and no load message is produced. The entry point is called and a load message is produced if the DLL is loaded because of the L Runtime Command Option or because the DLL is present in the **rmautold** subdirectory of the execution directory.

Function declaration for **RM_AddOnLoadMessage**:

```
char*     RM_AddOnLoadMessage(void);
```

RM_AddOnTerminate

This entry point, if present, is called by the runtime system during termination. Execution of all COBOL programs is complete when the runtime system calls this entry point. It allows the optional support module to perform any cleanup actions that may be necessary.

Note The Windows runtime supports the "call-by-filename" loading of DLLs as described in "Methods of Using Non-COBOL Subprograms" on page G-2. DLLs loaded with this method will be unloaded when a CANCEL statement references them. In this case, the **RM_AddOnTerminate** entry point is called just prior to unloading the DLL, after having called **RM_AddOnCancelNonCOBOLProgram**, and the runtime system is not necessarily about to terminate.

The **RM_AddOnTerminate** function is called when the module is unloaded, even if the **RM_AddOnInit** function (see page G-14) for the module did not succeed. Thus, the code for this function must not depend on the success of the **RM_AddOnInit** function.

Function declaration for **RM_AddOnTerminate**:

```
void      RM_AddOnTerminate(void);
```

RM_AddOnVersionCheck

This entry point, if present, provides a method of verifying that the runtime system and the optional support module are compatible.

If **RM_AddOnVersionCheck** is not present, the support module is assumed to support the current interface version of the runtime system that calls the support module.

If **RM_AddOnVersionCheck** is present, it will be passed a version string, two support module interface versions, and a pointer for the support module to store a desired interface version. The version string (for example, 8.0*n.nn*) is defined by the VERSION macro in the header file **version.h** (provided with the RM/COBOL system). The runtime

support module interface versions indicate the minimum and maximum versions that the runtime system can support. The RM/COBOL runtime system (version 7.50 or later) supports support module interface versions 1 and 2. For Windows, these two interface versions are identical. In the future, the runtime system may support other, partially or completely incompatible, interface versions.

It is the responsibility of the support module to verify that it supports one of the interface versions supported by the runtime system and to return the interface version it supports. If the support module does not support any of the interfaces supported by the runtime system, the support module should return FALSE (0). In this case, or if the support module returns an invalid interface version, the runtime system will display an appropriate message and then terminate. Returning TRUE (1) and an interface version in the range supported by the runtime system allows the runtime system to continue. The support module may use the current interface version by returning the value CURRENT_SUPPORT_MODULE_INTERFACE_VERSION (defined in the supplied header file, **rmc85cal.h**).

The support module may also use the value of the version string to verify compatibility with the runtime system. If the support module determines that it is not compatible with the runtime system, it should return FALSE. In this case, the support module might display a meaningful message before the runtime system displays its message and terminates.

Function declaration for **RM_AddOnVersionCheck**:

```
BOOLEAN   RM_AddOnVersionCheck(char* Version,
                               int MinRuntimeInterfaceVersion,
                               int MaxRuntimeInterfaceVersion,
                               int* DesiredInterfaceVersion);
```

RM_EntryPoints and RM_EnumEntryPoints

When the runtime system loads an optional support module, it looks for the exported symbols **RM_EntryPoints** and **RM_EnumEntryPoints** to determine whether the support module contains any COBOL-callable functions. Each optional support module defines only those COBOL-callable functions defined in that support module using either the **RM_EntryPoints** symbol declaration or the **RM_EnumEntryPoints** entry point. If neither of these symbols is exported, then the runtime system looks for an .EDATA section in the DLL. If the .EDATA section is found, the exported names listed in the .EDATA section are considered to be COBOL-callable functions; otherwise, the DLL is considered not to contain any COBOL-callable functions.

The use of the subprogram name table **RM_EntryPoints** is described on page G-4.

If the entry point **RM_EnumEntryPoints** is found, it is called repeatedly to obtain the COBOL-callable names, function addresses, and function names of the COBOL-callable functions in the support module. This function should return a pointer to a structure that is equivalent to one entry in the **RM_EntryPoints** table. The end of the entry points is indicated by returning a null pointer or a structure whose first pointer is NULL. The index parameter starts at zero for the first call and is incremented for each subsequent call.

If both symbols are present, **RM_EnumEntryPoints** takes precedence.

See the example on page G-5 for the symbol declaration for **RM_EntryPoints**.

Function declaration for **RM_EnumEntryPoints**:

```
ENTRYTABLE*      RM_EnumEntryPoints(int index);
```

Debugging C Subprograms

Non-COBOL subprograms can be debugged using the debugger supplied with the C compiler used to build the DLL.

In order to include debugging information in the DLL, use the following command for the 32-bit Microsoft Visual C++ compiler:

```
cl msgbox.c -Zpl -Zi /link -out:msgbox.dll -dll -export:MsgBox,@1
   -section:.edata,IRD user32.lib
```

Alternatively, use the following command for the 32-bit Watcom C compiler, version 10.6 or later:

```
wclX86 -l=nt_dll -bd -d2 msgbox.c -"export MSGBOX.1=_MsgBox"
```

After creating a version of the DLL containing debugging information, start the debugger on **runcobol.exe**. The Microsoft debugger allows you to add both runcobol.exe and the DLL file to a project and then set a breakpoint in the DLL before beginning execution.

The Watcom debugger allows you to set a breakpoint that is triggered when the module containing the DLL is loaded. Once it has been loaded, the source for the module can be viewed and additional breakpoints can be set. For more information, see the documentation supplied with the debugger you are using.

Calling a CodeBridge Subprogram Library

It is possible to use non-COBOL subprogram libraries built using CodeBridge and call them in the manner described in this appendix.

The CodeBridge Builder generates functions that are to be called by RM/COBOL. These generated functions then call the C functions that are described in the template file. The name of the generated function is the same as the C function name with a prefix of "RMDLL" added to it. For example, if the name of the C function is MessageBox, the name of the generated function is RMDLLMessageBox.

It is possible for a C function that calls the CodeBridge Library functions directly also to call functions that were built by CodeBridge Builder. A C function could call RMDLLMessageBox directly either by using the ARGUMENT_ENTRY structure that was passed from RM/COBOL or by constructing one that suited the needs of the C function.

One use of this capability would be to hide conversions of C data items to COBOL data items. "Example 6: Converting Buffered C Data" on page B-18 in Appendix B, *CodeBridge Examples*, describes a case in which such conversions are necessary even though CodeBridge is being used. In that example, the function **cstring2text** is called from COBOL to convert data stored in a buffer by a C function call. This conversion could be hidden from the RM/COBOL program by embedding the conversion in a C function that first calls the C function to store the data in the buffer and then also calls the generated C function, **RMDLLcstring2text**.

Appendix H: Non-COBOL Subprogram Internals for UNIX

This appendix describes the internal details of how a non-COBOL subprogram is called from an RM/COBOL program running under UNIX. While it is possible to write non-COBOL subprograms that directly use this information to handle COBOL argument conversions, it is highly recommended that CodeBridge be used for this purpose instead. This appendix also provides information on preparing a non-COBOL subprogram for use by an RM/COBOL program on UNIX. (For additional information, see the "CALL Statement" section of Chapter 6, *Procedure Division Statements*, in the *RM/COBOL Language Reference Manual*.)

Note The information presented here assumes a working knowledge of the C programming language. The material in Appendix C, *Useful C Information*, is not comprehensive enough to provide this necessary background.

C Subprograms

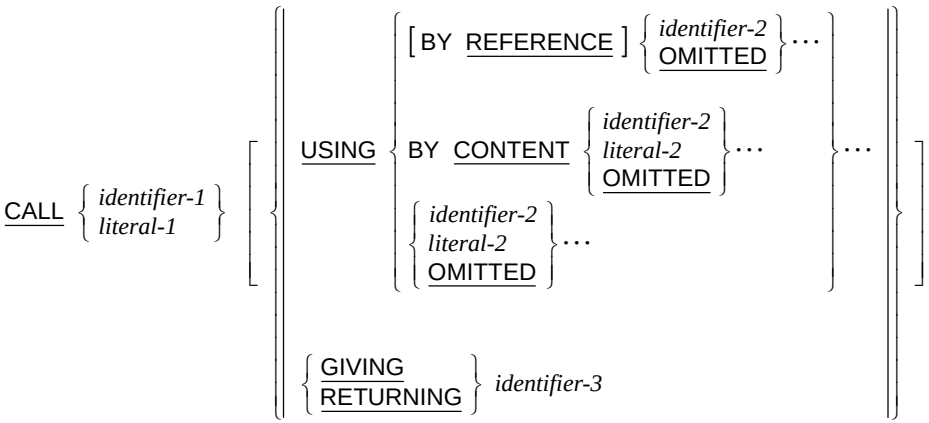
To modify or write a C subprogram that can be called from the RM/COBOL runtime system requires an understanding of the fundamental tasks involved. First, in order to access C language subprograms from the RM/COBOL runtime system, you must build a shared object, normally referred to as an "optional support module." (For more information on shared objects and optional support modules, see Appendix D, *Support Modules (Non-COBOL Add-Ons)*, of the *RM/COBOL User's Guide*.) The shared object must then be placed so that the RM/COBOL runtime system can locate it, either by looking in a special subdirectory (**rmcobolso**) of the runtime execution directory (normally **/usr/bin**) or by using the L Option on the Runtime Command. Finally, you must provide information about what entry points you wish the runtime system to use.

Calling C Subprograms from COBOL

This section describes the COBOL CALL syntax and explains how a C programmer can write a subprogram that can be called from RM/COBOL. The COBOL CALL statement explains the use of the non-COBOL subprogram from the COBOL programmer's perspective while the other topics describe the structures and the function prototype that the C programmer needs to understand.

COBOL CALL Statement

The syntax for the Format 2 CALL statement in the RM/COBOL program is as follows:



[ON EXCEPTION *imperative-statement-1*]

[NOT ON EXCEPTION *imperative-statement-2*]

[END-CALL]

The value of the contents of the data item specified by *identifier-1* or the value of *literal-1* is the program-name of the subprogram to be called.

identifier-2 or *literal-2* are one or more actual arguments to be passed to the called program. If the BY CONTENT phrase applies to an argument, a temporary copy of the item is passed, thus preventing the subprogram from modifying the original item.

identifier-3 is an actual argument to be passed to the called program for the purposes of returning a result to the calling program.

The RM/COBOL runtime system locates the subprogram with the program-name specified by *literal-1* or the value of the data item referenced by *identifier-1*. See the discussion of "Subprogram Loading" in Chapter 8, *RM/COBOL Features*, of the *RM/COBOL User's Guide*, for additional information on locating subprograms.

C Subprogram Name Table Structure

The RM/COBOL runtime system can locate the C subprograms only if their names appear in the subprogram name table. The subprogram name table is an array of name table entries. Each name table entry is a C structure that is defined as follows:

```
typedef struct EntryTable
{
    char       *EntryPointCobolName;   /* name of subroutine as in call */
    int        (*EntryPointAddress)(); /* entry point address */
    char       *EntryPointName;        /* name of entry point in object */
} ENTRYTABLE;
```

Character strings must be null terminated. The last array entry must consist of NULLs. The name of the subprogram name table must be **RM_EntryPoints**.

The RM/COBOL runtime system does not use the EntryPointAddress entry in this structure. Instead, the EntryPointName entry is used to find the procedure address for the procedure that has the given name. Thus, each value supplied in an EntryPointName entry must match that of an external symbol in the shared object. When the shared object is loaded, the runtime system looks up the procedure address for each entry using the supplied name; if the name is not found, an error occurs and the runtime system is terminated with an appropriate message.

RM_EntryPoints is one of the predefined symbols in an optional support module. For complete information about all of the predefined symbols, see "Special Entry Points for Support Modules" on page H-12.

Note The ENTRYTABLE typedef is defined in **rmc85cal.h**, which is provided with RM/COBOL systems. This header file should be included (with a preprocessor **#include** statement) in C source that defines COBOL-callable subprograms. Other header files (**rtarg.h**, **standdef.h**, and **rmport.h**) are referenced by **rmc85cal.h**. These files are also provided with RM/COBOL systems.

Example

```
ENTRYTABLE      RM_EntryPoints[] =
{
    {"SUB1NAME",      sub1,      "sub1"},
    {"SUB2NAME",      sub2,      "sub2"},
    {NULL,            NULL,      NULL  }
};
```

In this example, "SUB1NAME" and "SUB2NAME" are the COBOL-callable program-names, sub1 and sub2 are the addresses of the C subprograms (functions), and "sub1" and "sub2" are the names of the C subprograms (functions).

Parameters Passed to the C Subprogram

The RM/COBOL runtime system passes six parameters on the stack to the called C subprogram. The following is a sample COBOL-callable C subprogram function prototype:

```
int sub1
   (
      char                 *name,           /* param1 */
      unsigned short        arg_count,      /* param2 */
      ARGUMENT_ENTRY        arg_vector[],   /* param3 */
      unsigned short        initial_state,  /* param4 */
      void                 *reserved,       /* param5 */
      RUNTIME_CALLS_TABLE   callbacktable   /* param6 */
   );
```

Note The above function prototype does not work on Windows. See page I-2 for a function that does work for either Windows or UNIX.

The six parameters are described as follows:

1. Pointer to the called program-name, which is a null-terminated ASCII string containing the name used by the run unit to identify the called subprogram. The called program-name is always uppercase-only, regardless of the case of the name in the calling COBOL program.
2. Argument count, which is the number of arguments, including arguments explicitly specified with the OMITTED keyword, specified in the USING phrase of the CALL statement. The argument in the GIVING (RETURNING) phrase, if specified, is not included in the count.

3. Pointer to the argument array, which is an array of structures describing each of the actual arguments passed in the GIVING (RETURNING) and USING phrases of the CALL statement. The structure of an argument description entry is described in "COBOL Argument Entry Structure for C" on page H-6 and is defined in the **rtarg.h** header file, which is provided with RM/COBOL systems.

4. Initial state flag, which contains a zero to indicate that the subprogram is being called for the first time in the run unit or the first time since a CANCEL statement has been executed for the subprogram name. A nonzero value indicates that the subprogram should remain in its last used state. It is the responsibility of the called subprogram (rather than the runtime system) to examine the initial state flag and decide which variables need to be reinitialized. In any case, on each call, all C automatic variables are reallocated on the stack without being initialized to any particular value (that is, C automatic variables have arbitrary values).

5. Pointer value NULL (for compatibility with Windows non-COBOL subprograms).

6. Pointer to the runtime call-back table, which is a structure that contains the size of the table, the version number of the table, and a list of subprogram addresses in the runtime system. The CodeBridge Builder uses the call-back table to obtain access to some utility subprograms in the runtime system. The description of this table is available in **rtcallbk.h**, a header file provided with RM/COBOL systems. The table is named RUNTIME_CALLS_TABLE.

Note The fifth and sixth parameters are optional. Although the runtime system will always pass these values, the called subprogram does not have to declare them. The prototype for the called function may omit the sixth or both the fifth and sixth parameters. The runtime call-back table is required if the subprogram uses any of the CodeBridge Library functions.

The called subprogram must set an integer return value before returning control to the runtime system. A value of RM_FND (defined as 0 in **rtarg.h**) indicates that the subprogram was found and that the runtime system should continue executing the COBOL program. A value of RM_STOP (defined as 1 in **rtarg.h**) indicates that the subprogram terminated because of a fatal error, such as incorrect parameters, and that the runtime system should terminate the run unit. An explicit return statement should be used to set the return value since otherwise the run unit might be unintentionally terminated. The subprogram must not terminate with the system function **exit()**, since the runtime system could not do an orderly shutdown of the run unit in this case.

Once an optional support module is loaded, it remains loaded until the runtime system terminates. Use of the CANCEL statement to cancel a C subprogram sets the initial flag to zero on the next entry into the subprogram, but has no effect on the values of the external and static variables used in the C subprogram.

The argument entry table (arg_vector) contains descriptions of the actual arguments specified in the CALL statement. The arg_vector[0] entry describes the first actual argument in the USING phrase of the CALL statement. The arg_vector[arg_count - 1] entry describes the last actual argument in the USING phrase of the CALL statement. The arg_vector[-1] entry describes the argument specified in the GIVING (RETURNING) phrase of the CALL statement. If the GIVING (RETURNING) phrase is omitted from the CALL statement, or if any actual argument is specified as OMITTED in the USING phrase of the CALL statement, the corresponding arg_vector entry contains a type value 32 (OMITTED, as shown in Table H-1 on page H-7) and the remaining fields are zero.

C subprograms that access the GIVING argument in arg_vector[-1] will function correctly only for RM/COBOL version 7 (or later) runtimes because prior runtimes did not make a GIVING argument entry available in arg_vector[-1]. A subprogram that uses the GIVING argument should verify that it is available by use of the version number in the runtime call-back table, the address of which is provided by the sixth parameter to the subprogram. The version number must be 0x0700 or greater for a GIVING argument to be available.

COBOL Argument Entry Structure for C

To a subprogram written in C, an argument entry is defined by the following structure, which is included in the **rtarg.h** header file:

```
typedef struct ArgumentEntry
{
  char          *a_address;   /* pointer to start of argument */
  BIT32          a_length;    /* length of argument */
  BIT16          a_type;      /* type of argument (RM/COBOL data type) */
  char           a_digits;    /* digit count (0-30) */
  char           a_scale;     /* implied decimal location (signed) */
  BYTE          *a_picture;   /* pointer to encoded edit picture */
} ARGUMENT_ENTRY;
```

a_address specifies the lowest address byte of the argument.

a_length specifies the number of bytes allocated to the argument.

a_type specifies the RM/COBOL data type as a number from Table H-1 (see page H-7). Names for these type numbers are defined in **rtarg.h**. (For an explanation of the data type abbreviations and a description of the RM/COBOL data types listed in Table H-1, see Table 9-2 in Chapter 9, *Debugging*, and Appendix C, *Internal Data Formats*, of the *RM/COBOL User's Guide*.)

a_digits specifies the actual number of digits in a numeric data item (where the type of argument is in the range 0 through 12). It is set to zero for nonnumeric data items.

a_scale specifies the power of 10 by which the digits in a numeric data item (where the type of argument is in the range 0 through 12) must be multiplied to obtain the numeric value of the data item. The power of 10 is represented as a signed, 2's complement number. It is set to zero for nonnumeric data items.

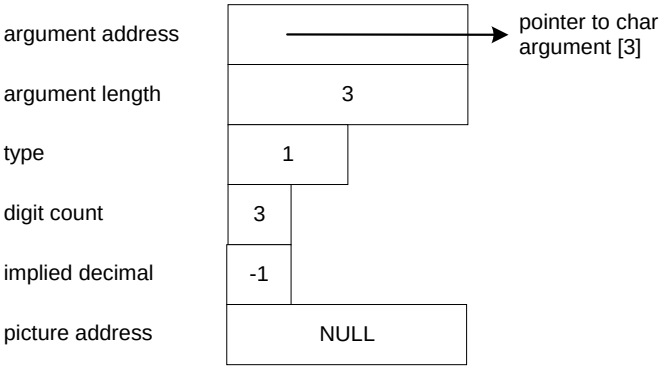
a_picture specifies the lowest addressed byte of the encoded picture for edited items (type of argument equals 0, 20 or 21). It is set to zero for all other types.

Table H-1: RM/COBOL Data Types as Numbers

Type Number	RM/COBOL Data Type	Type Number	RM/COBOL Data Type
0	NSE	16	ANS
1	NSU	17	ANS (justified right)
2	NTS	18	ABS
3	NTC	19	ABS (justified right)
4	NLS	20	ANSE
5	NLC	21	ABSE
6	NCS	22	GRP (fixed length)
7	NCU	23	GRPV (variable length)
8	NPP	25	PTR
9	NPS	26	NBSN
10	NPU	27	NBUN
11	NBS	32	OMITTED
12	NBU		

Note The data type GRPV (23) does not occur when C$CARG is called with the formal argument name or when C$DARG is called with an actual argument number that corresponds to an argument that is a variable-length group. In all cases, RM/COBOL passes variable-length group actual arguments as if they were a fixed-length group of the maximum length. (See Appendix F, *Subprogram Library*, of the *RM/COBOL User's Guide*.)

For example, suppose a CALL statement specifies one argument in its USING list and this argument refers to a three-byte numeric unsigned (NSU) data item with a PICTURE character-string of 99V9. The following is a diagram of the structure in C.



Accessing C Subprograms

You can access a C language subprogram from the RM/COBOL runtime system by either of the following two methods:

- Give each C subprogram a unique name and entry point. Source module **usrsub.c** (delivered with the RM/COBOL system) provides an example of this method.
- Give each C subprogram a unique name and share the same entry point.

In the second case, it is necessary to determine which C subprogram has been called. The following example illustrates one way this might be accomplished.

```
#include "rmc85cal.h"

int library
    (
        char           *name,
        unsigned short  arg_count,
        ARGUMENT_ENTRY  arg_vector[],
        unsigned short  initial_state
    );

ENTRYTABLE RM_EntryPoints[] =
{
    {"SUBA", (int (*)())library, "library" },
    {"SUBB", (int (*)())library, "library" },
    {NULL,   (int (*)())NULL,    NULL      }
};

int library
    (
        char           *name,
        unsigned short  arg_count,
        ARGUMENT_ENTRY  arg_vector[],
        unsigned short  initial_state
    )
{
    int       entry_no;
    const int MAX_ENTRIES =
        (sizeof(RM_EntryPoints)/sizeof(RM_EntryPoints[0])) - 1;

    for ( entry_no = 0; entry_no < MAX_ENTRIES; entry_no++ )
    {
        if (
             !strcmp
                 (
                   RM_EntryPoints[entry_no].EntryPointCobolName, name
                 )
           )
            break; /* matching name found */
    }

    switch ( entry_no )
    {
        case 0:  /* "SUBA" called */
            /*
             *  "SUBA" code goes here
             */
            return RM_FND;

        case 1:  /* "SUBB" called */
            /*
             *  "SUBB" code goes here
             */
            return RM_FND;

        default:
            return RM_STOP;  /* logic error, stop run unit */
    }
}
```

Preparing C Subprograms

This section explains how to create an optional support module using either a new C subprogram or an existing object for a C subprogram that was previously being linked into the RM/COBOL runtime system using the **customiz** script.

Creating a Support Module from a C Source

C subprograms must be compiled and linked to produce a shared object to be used as a support module. In the discussion below, C source files are assumed to have an extension of **.c** and C object files are assumed to have an extension of **.o**. Optional support modules must have an extension of **.so**.

A **makefile** is provided with the RM/COBOL development and runtime systems that can be used or modified to build a shared object. You may modify the makefile by adding a new target for your support module or you may modify module **usrsub.c** (delivered with the RM/COBOL system). The makefile includes the C compiler options used by Liant Software to build the optional support modules shipped with the RM/COBOL release on your particular platform.

Note These C compiler options in the makefile may not be appropriate or correct for your C compiler. In order to build a shared object to be used as a support module with the RM/COBOL runtime system, you must specify options to tell the compiler and linker that you want to produce an ELF (Executable and Linking Format) object file (for example, -b elf), that you want to produce a dynamically-linked executable (for example, -d y), and that you want the linker to produce a shared object (for example, -G).

Producing a support module for use on HP/UX version 10.20 and later requires that you specify an additional C compiler option to generate position-independent code. Other UNIX systems do not require position-independent code for support modules. The **makefile** includes the appropriate compiler option to generate position-independent code on HP/UX.

Linking a support module for IBM AIX 4.2 requires both an “import” file, **runcobol.imp**, to make RM/COBOL runtime system symbols available and an “export” file to make support module symbols available. The **runcobol.imp** file is supplied with the RM/COBOL development and runtime systems for IBM AIX 4.2. The “export” file must be provided by the user. A sample export file, **libusr.exp**, is also provided with the RM/COBOL release as an example of what the user must provide. The **makefile** includes appropriate loader options to use the import and export files when building support modules on IBM AIX 4.2

A separate “samples” makefile is provided with the RM/COBOL development system in the **cbsample** subdirectory. This makefile has targets that are called by the various script files used to demonstrate CodeBridge. Additional information about the CodeBridge samples may be found in the **README.txt** file in the CodeBridge samples directory. For the remainder of this section, **makefile** refers to the makefile that is present in the main installation directory (normally, **/usr/rmcobol**) rather than the special CodeBridge “samples” makefile.

Assuming a C source file named **usrsub.c**, the following command generates the subprogram object file and links a shared object to be used as an optional support module with the runtime system:

```
make libusr MODULES=usrsub.o
```

The **makefile** compiles and links the default subprogram module **usrsub.c**. The resulting optional support module **libusr.so** is then copied into the **rmcobolso** subdirectory of the current directory. The following describes each of the files involved in the process:

- **usrsub.c** is your C subprogram source file that will be compiled to produce **usrsub.o**.
- **usrsub.o** is the C subprogram object file that is linked to create **libusr.so**.
- **libusr.so** is the resulting shared object (optional support module). Although it is unnecessary to name your support module **libusr.so**, the name chosen must have an extension of **.so**.

Note Filenames of optional support modules must be unique even if the modules are located in different directories. The runtime system assumes that support modules with the same name are the same and, therefore, ignores all subsequent support modules with the same name as one already loaded.

If your optional support module uses functions from the C library that are not also used by the runtime system, you will see a message similar to the following when the runtime system tries to load the support module:

```
dynamic linker: runcobol: relocation error: symbol not found: symbol
```

You will need to add the C library name to the compile/link command (for example, cc). Depending on your particular support module, other library names may also need to be added.

You can test the newly built shared object by using the L (Library) Option on the RM/COBOL Runtime Command (see Chapter 7, *Running*, in the *RM/COBOL User’s Guide*) to specify the location of the support module in the test subdirectory. After testing is complete, you should copy the support module into the **rmcobolso** subdirectory of the executable directory (normally **/usr/bin**) so that the runtime system will

automatically load your support module. Once this has been done, your support module will be available for use in production mode.

Creating a Support Module from a C Object (No Source)

If you have old C subprograms that you have been linking into the runtime system, but no longer have the source (to be able to build a shared object), it may still be possible to build a shared object from the old object (**.o**) file. You will need to write a C wrapper module. You can use **usrsub.c** as a starting point, which is the method used in the remainder of this topic. Modify the entry points table to include the COBOL-callable name(s) of the C functions you wish to access in the old object. Then modify the entry points table to reference the proper C function(s) name(s) (the UNIX command **nm** may help you determine the function names). Finally, include an `extern` declaration for the C function names in the **usrsub.c** source as follows:

```
extern int oldcfunction();
```

Use the following command to build the shared object:

```
make libusr MODULES="usrsub.o oldcobject.o"
```

If you want to modify the **makefile** to change the name of the shared object, simply duplicate the **libusr** section of the makefile and change the names as appropriate or rename file **libusr.so** to the desired filename.

Special Entry Points for Support Modules

When the runtime system (or other RM/COBOL component) loads an optional support module, it looks for certain predefined symbols (entry points and variable names), and varies its actions based on the presence or absence of these symbols. One such variable name is **RM_EntryPoints** (discussed in "C Subprogram Name Table Structure" on page H-3). The example subprogram, **usrsub.c**, which is distributed with the RM/COBOL system, contains examples of all of these entry points and symbols. It can be used as a starting point when developing optional support modules.

The complete list of these special names is as follows:

- RM_AddOnBanner
- RM_AddOnCancelNonCOBOLProgram
- RM_AddOnInit
- RM_AddOnLoadMessage
- RM_AddOnTerminate
- RM_AddOnVersionCheck
- RM_EntryPoints and RM_EnumEntryPoints

Note On UNIX, only the **RM_EntryPoints** symbol declaration (or the **RM_EnumEntryPoints** entry point) is required for an optional support module. All other entry points are optional.

The following sections describe these entry points and special variables.

RM_AddOnBanner

This entry point, if present, should return a pointer to a character string that will be displayed along with the runtime system banner message. The support module banner may be used to display any required copyright notice. The support module banner is displayed only if the K Option of the Runtime Command is not present.

Function declaration for **RM_AddOnBanner**:

```
char*     RM_AddOnBanner ();
```

RM_AddOnCancelNonCOBOLProgram

This entry point, if present, is called by the runtime system when a CANCEL verb is executed for a program-name that is defined in the optional support module. It allows the support module to do any cleanup actions that may be necessary. For example, this entry point might be specified to allow the support module to close any open files when the COBOL program cancels the associated non-COBOL subprogram. The program-name of the non-COBOL subprogram for which a CANCEL has been performed is passed as a parameter to the entry point.

Function declaration for **RM_AddOnCancelNonCOBOLProgram**:

```
void      RM_AddOnCancelNonCOBOLProgram (char* ProgramName);
```

RM_AddOnInit

This entry point, if present, is called to initialize the optional support module. All support modules will be initialized (if initialization is requested) before the runtime system begins executing the first COBOL program. The entry point should return zero to indicate successful initialization or a non-zero value to indicate that the support module initialization failed. If the initialization fails, the runtime system will display an appropriate message and then terminate.

Note If the support module determines that successful initialization is not possible, the support module should produce appropriate messages to allow the user to correct the problem.

The support module is passed the shell command line arguments in the arguments Argc (the argument count) and Argv (the argument vector). The support module is also passed a pointer to the runtime call back table if the support module interface version is set to 2.

Function declaration for **RM_AddOnInit** for interface version 1:

```
int       RM_AddOnInit (int Argc, char** Argv);
```

Function declaration for **RM_AddOnInit** for interface version 2:

```
int       RM_AddOnInit(int Argc,
                       char** Argv,
                       RUNTIME_CALLS_TABLE *pRtCall);
```

RM_AddOnLoadMessage

This entry point, if present, should return a pointer to a character string that is displayed along with the load messages of other optional support modules. These load messages allow the user to verify which support modules the runtime system has loaded. The message may contain text to identify the support module and, if desired, the version number or the build date. Load messages are displayed only if the V Runtime Command Option is present, the V=DISPLAY keyword-value pair is specified in the RUN-OPTION configuration record, or the RM_DYNAMIC_LIBRARY_TRACE environment variable
is defined.

If load messages are being displayed, the runtime system generates a load message consisting of the complete pathname for the support module regardless of whether the **RM_AddOnLoadMessage** entry point is defined or not defined in the support module. If the **RM_AddOnLoadMessage** entry point is defined, the returned string is appended to the pathname in this load message.

Function declaration for **RM_AddOnLoadMessage**:

```
char*     RM_AddOnLoadMessage ();
```

RM_AddOnTerminate

This entry point, if present, is called by the runtime system during termination. Execution of all COBOL programs is complete when the runtime system calls this entry point. It allows the optional support module to perform any cleanup actions that may be necessary.

The **RM_AddOnTerminate** function is called when the module is unloaded, even if the **RM_AddOnInit** function (see page H-14) for the module did not succeed. Thus, the code for this function must not depend on the success of **RM_AddOnInit** function.

Function declaration for **RM_AddOnTerminate**:

```
void      RM_AddOnTerminate ();
```

RM_AddOnVersionCheck

This entry point, if present, provides a method of verifying that the runtime system and the optional support module are compatible.

If **RM_AddOnVersionCheck** is not present, the support module is assumed to support the current interface version of the runtime system that calls the support module.

If **RM_AddOnVersionCheck** is present, it will be passed a version string, two support module interface versions, and a pointer for the support module to store a desired interface version. The version string (for example, 8.0*n.nn*) is defined by the VERSION macro in the header file **version.h** (provided with the RM/COBOL system). The runtime support module interface versions indicate the minimum and maximum versions that the runtime system can support. The RM/COBOL runtime system (version 7.50 or later) supports support module interface versions 1 and 2. For UNIX, these two interface versions differ only in the arguments passed to **RM_AddOnInit**, as documented in the description of that special entry point (see page H-14). Interface version 1 was the support module interface version supported by the version 7.10 runtime system. Interface version 2 is the new current support module interface version supported by version 7.50 or later runtime systems. In the future, the runtime system may support other, partially or completely incompatible, interface versions.

It is the responsibility of the support module to verify that it supports one of the interface versions supported by the runtime system and to return the interface version it supports. If the support module does not support any of the interfaces supported by the runtime

system, the support module should return FALSE (0). In this case, or if the support module returns an invalid interface version, the runtime system will display an appropriate message and then terminate. Returning TRUE (1) and an interface version in the range supported by the runtime system allows the runtime system to continue. The support module may use the current interface version by returning the value CURRENT_SUPPORT_MODULE_INTERFACE_VERSION (defined in the supplied header file, **rmc85cal.h**).

The support module may also use the value of the version string to verify compatibility with the runtime system. If the support module determines that it is not compatible with the runtime system, it should return FALSE. In this case, the support module might display a meaningful message before the runtime system displays its message and terminates.

Function declaration for **RM_AddOnVersionCheck**:

```
BOOLEAN   RM_AddOnVersionCheck (char* Version,
                                int MinRuntimeInterfaceVersion,
                                int MaxRuntimeInterfaceVersion,
                                int* DesiredInterfaceVersion);
```

RM_EntryPoints and RM_EnumEntryPoints

When the runtime system loads an optional support module, it looks for the symbols **RM_EntryPoints** and **RM_EnumEntryPoints** to determine whether the support module contains any COBOL-callable functions. Each optional support module defines only those COBOL-callable functions defined in that support module using either the **RM_EntryPoints** symbol declaration or the **RM_EnumEntryPoints** entry point.

The use of the subprogram name table **RM_EntryPoints** is described on page H-3.

If the entry point **RM_EnumEntryPoints** is found, it is called repeatedly to obtain the COBOL-callable names, function addresses, and function names of the COBOL-callable functions in the support module. This function should return a pointer to a structure that is equivalent to one entry in the **RM_EntryPoints** table. The end of the entry points is indicated by returning a null pointer or a structure whose first pointer is NULL. The index parameter starts at zero for the first call and is incremented for each subsequent call.

If both symbols are present, **RM_EnumEntryPoints** takes precedence.

See the example on page H-3 for the symbol declaration for **RM_EntryPoints**.

Function declaration for **RM_EnumEntryPoints**:

```
ENTRYTABLE*      RM_EnumEntryPoints (int index);
```

Calling a CodeBridge Subprogram Library

It is possible to use non-COBOL subprogram libraries built using CodeBridge and call them in the manner described in this appendix.

The CodeBridge Builder generates functions that are to be called by RM/COBOL. These generated functions then call the C functions that are described in the template file. The name of the generated function is the same as the C function name with a prefix of "RMDLL" added to it. For example, if the name of the C function is MessageBox, the name of the generated function is RMDLLMessageBox.

It is possible for a C function that calls the CodeBridge Library functions directly also to call functions that were built by CodeBridge Builder. A C function could call RMDLLMessageBox directly either by using the ARGUMENT_ENTRY structure that was passed from RM/COBOL or by constructing one that suited the needs of the C function.

One use of this capability would be to hide conversions of C data items to COBOL data items. "Example 6: Converting Buffered C Data" on page B-18 in Appendix B, *CodeBridge Examples*, describes a case in which such conversions are necessary even though CodeBridge is being used. In that example, the function cstring2text is called from COBOL to convert data stored in a buffer by a C function call. This conversion could be hidden from the RM/COBOL program by embedding the conversion in a C function that first calls the C function to store the data in the buffer and then also calls the generated C function, RMDLLcstring2text.

C Subprograms Performing Terminal I/O

The RM/COBOL runtime system changes terminal characteristics before passing control to a C language subprogram. If any processing requiring terminal I/O occurred (including operating system commands that use the terminal), you must reset the terminal to its original state by making a call to the routine **resetunit()**. If **resetunit()** was called, a call to **setunit()** must be made before control is returned to the run unit. Both functions are part of the runtime system and are described in the section "Runtime Functions for Support Modules" that begins on page H-18.

Debugging C Subprograms

It is recommended that subprograms initially be tested using a C main program that sets up the RM/COBOL argument entries and calls the subprogram. Once the subprograms are functioning properly, then build the shared object and test with the COBOL program.

C Subprogram Example

The C subprogram **usrsub.c** has been provided with your distribution media as an example of the predefined symbols and entry points used in creating optional support modules (shared objects). As distributed, **usrsub.c** does nothing of interest, but does serve as a template for developing an optional support module of your own. Remember, only the **RM_EntryPoints** symbol declaration (or the **RM_EnumEntryPoints** entry point) is required. All other entry points are optional.

Note The special entry points, SYSTEM, DELETE, and RENAME, which were included in the C source **sub.c** on previous releases of RM/COBOL, are not present in **usrsub.c**. These COBOL-callable functions are now part of the runtime system and are fully documented in Appendix F, *Subprogram Library*, of the *RM/COBOL User's Guide*.

Runtime Functions for Support Modules

RM/COBOL provides user-supplied C subprograms with entry points to some COBOL functions. The following routines use the standard C calling and parameter passing conventions:

- **RmForget (int y1, int x1, int y2, int x2).** This function marks the indicated area of screen memory as unknown. By doing so, the next COBOL display to that area will not be optimized based on the screen contents. This allows COBOL output to be correctly displayed over C subprogram output, which is not stored in the in-memory screen image.

  This routine requires four int parameters (two line and position pairs), which specify the upper-left (y1,x1) and lower-right (y2,x2) coordinates of the area of the screen to be marked as unknown. Valid values range from 0 to the line or position limit of the screen. Passing zero values mark the entire screen as being unknown. See the "C$Forget" section in Appendix F of the *RM/COBOL User's Guide* for more information. The function returns an int value of 0 for success.

- **RmRepaintScreen().** This function causes the RM/COBOL runtime system to redraw the entire current screen from an in-memory image. C routine output is erased. This function requires no parameters and does not return a value. See the REPAINT-SCREEN keyword of the CONTROL phrase in Chapter 8, *RM/COBOL Features*, of the *RM/COBOL User's Guide* for more information.
- **RmRefreshCwd().** This function causes the RM/COBOL runtime system to refresh its internal copy of the current working directory. This internal copy is used to construct complete filenames from any filename that is not fully qualified. This function should be called before returning to the COBOL program if a non-COBOL subprogram changes the current working directory with the **chdir()** C library routine. The RmRefreshCwd() routine has no parameters and does not return a value.
- **setunit().** This function restores the terminal to the state the RM/COBOL runtime system requires for terminal I/O. If the **resetunit()** function is called, the **setunit()** function must be called before returning to the runtime system. This function requires no parameters and does not return a value.
- **resetunit().** This function places the terminal in a "normal state" (that is, the state before the RM/COBOL runtime system was executed). This function should be used if any terminal I/O is going to be performed, including operating system commands that use the terminal. This function requires no parameters and does not return a value.

Appendix I: Calling the CodeBridge Library Directly

This appendix provides guidelines for calling the CodeBridge Library directly rather than having the CodeBridge Builder generate the interface code from a template file. In order to call the CodeBridge Library directly, you must use an alternate method for preparing non-COBOL subprograms, as described either in Appendix G, *Non-COBOL Subprogram Internals for Windows*, or Appendix H, *Non-COBOL Subprogram Internals for UNIX*.

Note The information presented here assumes a working knowledge of the C programming language. The material in Appendix C, *Useful C Information*, is not comprehensive enough to provide this necessary background.

In describing direct calls to the CodeBridge Library, the following topics are covered:

Following these discussions, an example of calling the CodeBridge Library directly is given on page I-14.

Including cbridge.h

Instead of including **rmc85cal.h**, include **cbridge.h** (which includes **rmc85cal.h**). **cbridge.h** defines the following:

- Values for the Flags parameter used for most CodeBridge Library functions
- CodeBridge internal conversion table (CONV_TABLE)
- Runtime entry point table (RUNTIME_CALLS_TABLE)
- Function prototype of each CodeBridge Library function
- Initialization and termination logic for the generated interface DLL (for Windows)

Declaring the C Function Return Value and Parameters

The function is called with six parameters. The function should have the form specified on page G-5 for Windows or the form specified on page H-4 for UNIX. The following form may be used if the function is to work under either Windows or UNIX:

```
RM_DLLEXPORT int RM_CDECL
FunctionName(char                 *pCalledName,
             unsigned short        ArgCount,
             ARGUMENT_ENTRY        Arguments[],
             unsigned short        InitialState,
             RM_HWND               hRtWindow,
             RUNTIME_CALLS_TABLE  *pRtCall)

{
    /* function implementation goes here */
    return RM_FND;
}
```

FunctionName is the name of the C function. The function return value must be declared as an int. The value returned to the calling COBOL program must be either RM_FND or RM_STOP (see pages G-6 and H-5).

pCalledName is the *Name* parameter used for the ConversionStartup library function (see page F-42).

ArgCount is the *ArgCount* parameter used for most CodeBridge Library functions.

Arguments is the *Arguments* parameter used for most CodeBridge Library functions.

InitialState could be used as the *Flags* parameter for the CobolInitialState library function (see page F-24), but normally would be used directly by the code.

hRtWindow is the window handle for the runtime on Windows and could be used as the *WindowsHandle* parameter for the CobolWindowsHandle CodeBridge library function (see page F-40), but normally would be used directly by the code. On UNIX, *hRtWindow* is a placeholder that should not be used since there is no window handle on UNIX.

pRtCall points to the runtime entry point table and is used to locate CodeBridge Library functions. For example, you could call DiagnosticMode (see page F-43) as follows:

```
pRtCall->pDiagnosticMode(DF_SILENT);
```

The C subprogram table structure, which defines the COBOL-callable entry points, references the function name as follows:

```
RM_DLLEXPORT ENTRYTABLE RM_EntryPoints[]=
{
    {"ProgramName", (int (RM_CDECL *)())FunctionName, "FunctionName"},
    {NULL,          (int (RM_CDECL *)()) NULL,        NULL}
};
```

ProgramName is the name used in the COBOL program to call the C function. For more information on the C subprogram name table, see page G-4 (for Windows) or page H-3 (for UNIX).

Note The macros RM_DLLEXPORT, RM_CDECL, and RM_HWND, are defined in **rmc85cal.h** (which is included by **cbridge.h**) to aid in writing code that will compile on both Windows and UNIX.

Specifying the COBOL Argument Number

The value of the *Arguments* parameter used for most CodeBridge Library functions is zero-relative. The first argument in the USING phrase of the RM/COBOL CALL statement is argument zero. RM/COBOL allows up to 255 arguments in the USING phrase (numbered 0 through 254). The argument in the GIVING (RETURNING) phrase of the RM/COBOL CALL statement is argument -1 (minus one).

Declaring C Data Items Used in the Conversion Process

This section describes requirements for declaring a C data item that will receive a converted COBOL argument value or whose converted value will be returned to a COBOL argument.

Numeric Conversions

C numeric data items can receive and supply values for Numeric conversions (CobolToFloat, CobolToInteger, FloatToCobol, and IntegerToCobol). For C numeric data items, you must define both the data item and a pointer to the data item. The pointer must be initialized with the address of the data item as follows:

```
type Name; type *pName = &Name;
```

where:

type is a C numeric type (such as int, unsigned short, or double).

Name is the name of the C numeric data item.

pName is the name of the pointer to the C numeric data item.

The pointer is required so that null-valued COBOL pointers can be passed to the C function and converted properly.

Note Because of the way numeric data items are declared (to handle null-valued pointers), you must adjust the way you pass C numeric data items by reference to other C functions. Normally you would pass *&Name*, but when using CodeBridge you must pass *pName* instead.

If an array of numbers is to be passed, you must define a numeric array. To pass an array of five long integers, use the following definition:

```
long MyLongArray[5]; long *pMyLongArray = MyLongArray;
```

String Conversions

C strings can receive and supply values for String conversions (CobolToGeneralString, CobolToNumericString, CobolToString, GeneralStringToCobol, NumericStringToCobol, and StringToCobol). To use C strings in the conversion process, define an uninitialized string pointer as follows:

```
type *pString;
```

where:

type is a C string type (such as char, signed char, or unsigned char).

pString is the name of the string pointer.

Because the actual storage for each C string is allocated dynamically by the CodeBridge Library, it is not necessary to define storage for the string.

If an array of strings is to be passed, you must define an array of string pointers. To pass an array of five strings, use the following definition:

```
char *pMyStringArray[5];
```

Address Conversions

C pointers can receive and supply values for Address conversions (CobolDescriptorAddress, CobolToPointerAddress, CobolToPointerBase, and PointerBaseToCobol). For Address conversions, define a C pointer as follows:

```
type *pCobolData;
```

where:

type is the C data type used for references to the COBOL data.

pCobolData is the name of the pointer to the COBOL data.

Note Be careful when using Address conversions. The address returned in *pCobolData* may be used to directly manipulate COBOL data. It is better to use Numeric and String conversions, which require less knowledge of COBOL data formats to accomplish the same purpose.

Pointer Numeric Component Conversions

Pointer Numeric Component conversions (CobolToPointerOffset, CobolToPointerSize, PointerOffsetToCobol, and PointerSizeToCobol) do not convert to and from COBOL arguments. Instead, they obtain (or set) auxiliary information about the components of RM/COBOL pointer arguments. They are handled in the same manner as Numeric conversions (see page I-4). For Pointer Numeric Component conversions, define both a C data item and a pointer to the data item as follows:

```
type Name; type *pName = &Name;
```

where:

type is a C numeric type (such as int, unsigned short, or double).

Name is the name of the C numeric data item.

pName is the name of the pointer to the C numeric data item.

Other Conversions

Other conversions (BufferLength, CobolDescriptorDigits, CobolDescriptorLength, CobolDescriptorScale, CobolDescriptorType, CobolToPointerLength, and EffectiveLength) do not convert to and from COBOL arguments. Instead, they obtain (or set) auxiliary information about COBOL arguments or components of RM/COBOL pointer arguments. They are handled in the same manner as Numeric conversions

without requiring the additional pointer definition. To use Other conversions, define a C numeric data item as follows:

```
type Name;
```

where:

type is a C numeric type (such as int, unsigned short, or double).

Name is the name of the C numeric data item.

BufferLength and EffectiveLength conversions allow arrays to be passed.

Trivial Conversions

You can call the CodeBridge Library conversion functions (CobolArgCount, CobolInitialState, or CobolWindowsHandle) to convert *ArgCount*, *InitialState*, or *hRtWindow* to a C data item. However, this is a trivial conversion because you must pass the value to the corresponding library function so that the function can store it in the C data item you provide. For example:

```
short WindowsHandle2;

if (!RtCall->pCobolWindowsHandle (0,
                            (void *)WindowsHandle2,
                            sizeof (WindowsHandle2),
                                    WindowsHandle))
{ RtCall->pConversionCleanup(ArgCount, pConvTable);
  return(RM_STOP);
}
```

is equivalent to (though slower and more difficult to understand than):

```
short WindowsHandle2 = hRtWindow;
```

The only benefit to using the conversion routines in this situation is that size error checking may be performed. In the example above, a short data type is used instead of HWND. If the actual value of the handle does not fit into a short data item, then an error would be returned.

Initializing and Terminating the Conversion Process

CodeBridge uses a dynamically allocated table to hold information about the conversion process. The size of this table depends on the actual number of arguments (*ArgCount*) passed from COBOL to C. The table is allocated by ConversionStartup (see page F-42) and deallocated by ConversionCleanup (see page F-41). Several other CodeBridge Library functions use this table. The C function must declare a local variable to hold a pointer to this table as follows:

```
CONV_TABLE *pConvTable;
```

Initialization

Before calling any other CodeBridge Library functions, the C function must initialize the conversion process by calling ConversionStartup as follows:

```
if(!RtCall->pConversionStartup(ArgCount, &pConvTable,
                               pCalledName, Version))
  return(RM_STOP);
```

Note *Version* is the CodeBridge Library version (for version 7.0, use 0x700).

The ConversionStartup call illustrates two general properties of calling CodeBridge Library functions. First, CodeBridge Library functions are called indirectly through pointers in the RUNTIME_CALLS_TABLE, RtCall. Adding the prefix “p” to the CodeBridge Library function name forms the name of the pointer. In the code above, the full reference is:

```
RtCall->pConversionStartup(…)
```

Second, most CodeBridge Library functions return TRUE to indicate success or FALSE to indicate failure. A failure condition indicates that processing should not continue. Hence, the previously listed sequence:

```
if(!RtCall->pConversionStartup(…))
  return(RM_STOP);
```

Termination

Just before returning to the calling COBOL program, the C function must terminate the conversion process by calling ConversionCleanup as follows:

```
RtCall->pConversionCleanup(ArgCount, pConvTable);
```

Note Because a program may have many exits, be sure that ConversionCleanup is called prior to each exit.

For example, the code will typically contain sequences such as:

```
if(!RtCall->pCodeBridgeLibraryFunction(…))
{ RtCall->pConversionCleanup(ArgCount, pConvTable);
  return(RM_STOP);
}
```

Converting COBOL Arguments to C Data Items

CodeBridge Library input conversion functions are used to initialize C data items with information from the calling COBOL program (see “Declaring C Data Items Used in the Conversion Process” on page I-4). For input conversions, the input conversion function must be called before the C function uses the target C data item.

For Numeric and String conversions (see pages I-4 and I-5, respectively), the input conversion function must be called if the corresponding output conversion function will be called. This allows CodeBridge to handle null-valued COBOL pointer arguments and to supply default values for omitted COBOL arguments. Note that for String conversions, a buffer is allocated to hold the string. If only output conversion is needed, do not set the PF_IN flag for the input conversion call.

Specifying the *ArgCount*, *ArgNumber*, and *Arguments* Parameters

The *ArgCount* and *Arguments* parameters are presented in "Declaring the C Function Return Value and Parameters" on page I-2. The *ArgNumber* parameter is explained in "Specifying the COBOL Argument Number" on page I-4.

Specifying the *Parameter* Parameter

For CobolToFloat, CobolToInteger, CobolToPointerOffset, and CobolToPointerSize conversions, the *Parameter* parameter must be:

```
(void **) &pName        /* address of pointer to C data item */
```

where *pName* is defined as described in "Numeric Conversions" on page I-4 and "Pointer Numeric Component Conversions" on page I-6.

For CobolToGeneralString, CobolToNumericString, and CobolToString conversions, the *Parameter* parameter must be:

```
(void **) &pString      /* address of C string pointer */
```

where *pString* is defined as described in "String Conversions" on page I-5.

For CobolDescriptorAddress, CobolToPointerAddress, and CobolToPointerBase conversions, the *Parameter* parameter must be:

```
(void **) &pCobolData   /* address of C pointer to COBOL data*/
```

where *pCobolData* is defined as described in "Address Conversions" on page I-6.

For all other input conversions, the *Parameter* parameter must be:

```
(void *)  &Name         /* address of C numeric data item */
```

where *Name* is defined as described in "Other Conversions" on page I-6.

Specifying the *Size* Parameter

When the target C data item is numeric, CodeBridge supports multiple C numeric data types with each input conversion function. For instance, CobolToInteger can store a converted COBOL numeric argument value in any C integer data type supported by the C compiler. The CodeBridge Library conversion routines determine the size of the C data item using the value of the *Size* parameter, typically **sizeof**(*Name*). For example, to

store a COBOL numeric argument in the C data item, `short MyShort`, call CobolToInteger specifying the *Size* parameter as `sizeof(MyShort)`.

If the target C data item is a string, the *Size* parameter overrides the default string size when the PF_SIZE flag is set. The default size for numeric strings is four more than the digit length of the COBOL argument; for non-numeric strings, it is one more than the length of the COBOL argument.

Specifying Other Parameters

Input String conversion functions, as well as BufferLength and EffectiveLength, require that *pConvTable*, the pointer to the CodeBridge conversion table (see page I-8), be passed in the *ConvTable* parameter.

For a discussion of the *Flags* parameter, see page F-3.

For conversion functions that support passing arrays, the *Occurs* parameter is the array size. The PF_OCCURS flag should be set if the value of this parameter is greater than one.

For Numeric and String conversions, the *Omitted* parameter is the default value for omitted COBOL arguments when the PF_VALUE_IF_OMITTED flag is set. Otherwise, if the PF_OPTIONAL flag is set, the default value for Numeric conversions is zero and the default value for String conversions is the empty string(""). If neither the PF_VALUE_IF_OMITTED flag nor the PF_OPTIONAL flag is set, an error occurs for an omitted argument.

For Numeric and String conversions, the *Repeat* parameter specifies the repeat count when the PF_REPEAT flag is set.

See CobolToInteger on page F-29 for a discussion of the *Scale* parameter.

For non-numeric String conversions, the *Value1* parameter specifies the strip/fill character when the PF_LEADING_VALUE flag is set. Likewise, the *Value2* parameter specifies the strip/fill character when the PF_TRAILING_VALUE flag is set.

Converting C Data Items to COBOL Arguments

CodeBridge Library output conversion functions are used to pass information from C data items back to the calling COBOL program (see "Declaring C Data Items Used in the Conversion Process" on page I-4). For output conversions, the output conversion function must be called after the C function last uses the source C data item and before returning to the calling COBOL program.

Specifying the *ArgCount*, *ArgNumber,* and *Arguments* Parameters

The *ArgCount* and *Arguments* parameters are presented in "Declaring the C Function Return Value and Parameters" on page I-2. The *ArgNumber* parameter is explained in "Specifying the COBOL Argument Number" on page I-4.

Specifying the *Parameter* Parameter

For FloatToCobol and IntegerToCobol conversions, the *Parameter* parameter must be:

```
(void *) pName        /* value of pointer to C data item */
```

where *pName* is defined as described in "Numeric Conversions" on page I-4.

For GeneralStringToCobol, NumericStringToCobol, and StringToCobol conversions, the *Parameter* parameter must be:

```
(void *) pString      /* value of C string pointer */
```

where *pString* is defined as described in "String Conversions" on page I-5.

For PointerBaseToCobol conversions, the *Parameter* parameter must be:

```
(void *) pCobolData   /* value of C pointer to COBOL data*/
```

where *pCobolData* is defined as described in "Address Conversions" on page I-6.

For PointerOffsetToCobol and PointerSizeToCobol conversions, the *Parameter* parameter must be:

```
(void *) pName        /* value of C numeric data item */
```

where *pName* is defined as described in “Pointer Numeric Component Conversions” on page I-6.

Specifying the *Size* Parameter

When the source C data item is numeric, CodeBridge supports multiple C numeric data types with each output conversion function. For instance, IntegerToCobol can convert any C integer data type supported by the C compiler to a COBOL numeric argument. The CodeBridge Library conversion routines determine the size of the C data item using the value of the *Size* parameter, typically **sizeof**(*Name*). For example, to convert the C data item, `short MyShort`, to a COBOL numeric argument, call IntegerToCobol specifying the *Size* parameter as `sizeof(MyShort)`.

If the source C data item is a string, the *Size* parameter overrides the default string size when the PF_SIZE flag is set. The default size for numeric strings is four more than the digit length of the COBOL argument; for non-numeric strings, it is one more than the length of the COBOL argument.

Specifying Other Parameters

For a discussion of the *Flags* parameter, see page F-3.

For conversion functions that support passing arrays, the *Occurs* parameter is the array size. The PF_OCCURS flag should be set if the value of this parameter is greater than one.

For Numeric and String conversions, the *Repeat* parameter specifies the repeat count when the PF_REPEAT flag is set.

See IntegerToCobol on page F-52 for a discussion of the *Scale* parameter.

For non-numeric String conversions, the *Value1* parameter specifies the strip/fill character when the PF_LEADING_VALUE flag is set. Likewise, the *Value2* parameter specifies the strip/fill character when the PF_TRAILING_VALUE flag is set.

Validating Properties of COBOL Arguments

In addition to the input and output conversion functions, the CodeBridge Library also contains functions to validate properties of COBOL arguments. These include the following:

- AssertDigits validates the number of digits in a COBOL numeric argument.
- AssertDigitsLeft validates the number of digits before the decimal point.
- AssertDigitsRight validates the number of digits after the decimal point.
- AssertLength validates the number of bytes in a COBOL argument.
- AssertSigned verifies that a COBOL argument is signed.
- AssertUnsigned verifies that a COBOL argument is unsigned.

These functions may be used with either input or output arguments. The functions can be called anytime after the call to ConversionStartup and before ConversionCleanup.

Follow the guidelines for conversion functions when specifying parameters for validation functions (see "Converting COBOL Arguments to C Data Items" on page I-9 and "Converting C Data Items to COBOL Arguments" on page I-12.)

Note Instead of calling AssertSigned or AssertUnsigned, the following functions may set the PF_ASSERT_SIGNED or PF_ASSERT_UNSIGNED flags to verify that the COBOL argument is signed or unsigned: CobolToFloat, CobolToGeneralString, CobolToInteger, CobolToNumericString, FloatToCobol, GeneralStringToCobol, IntegerToCobol, and NumericStringToCobol.

Example

The following example illustrates calling the CodeBridge Library directly.

```
#include "cbridge.h"
#define CLEANUP pRtCall->pConversionCleanup(ArgCount, pConvTable)
extern void DoTest01(int *OutInteger, char *InOutString);
RM_DLLEXPORT int RM_CDECL Test01(char *pCalledName,
                                 unsigned short  ArgCount,
                                 ARGUMENT_ENTRY  Arguments[],
                                 unsigned short  InitialState,
                                           RM_HWND  hRtWindow,
                            RUNTIME_CALLS_TABLE *pRtCall)
{   int OutInteger;    int *pOutInteger = &OutInteger;
    char *InOutString;
    CONV_TABLE *pConvTable;

    if (pRtCall->table_version < 700)
        return RM_STOP;

    if(!pRtCall->pConversionStartup(ArgCount, &pConvTable,
                                    pCalledName, 0x700))
        return RM_STOP;

    if(!pRtCall->pCobolToInteger(ArgCount, 0, Arguments, PF_IN, 0, 0,
                                 (void **) &pOutInteger, 0, 0,
                                 sizeof(OutInteger)))
    {  CLEANUP; return RM_STOP;  }

    if(!pRtCall->pCobolToString(ArgCount, 1, Arguments, pConvTable,
                                (PF_IN | PF_TRAILING_SPACES), 0, (""),
                                (void **) &InOutString, 0,
                                0, '\0', '\0'))
    {  CLEANUP; return RM_STOP;  }

    DoTest01(pOutInteger, InOutString);

    if(!pRtCall->pIntegerToCobol(ArgCount, 0, Arguments, PF_OUT, 0,
                                 (void *) pOutInteger, 0, 0,
                                 sizeof(OutInteger)))
    {  CLEANUP; return RM_STOP;  }

    if(!pRtCall->pStringToCobol(ArgCount, 1, Arguments,
                                (PF_OUT | PF_TRAILING_SPACES), 0,
                                InOutString, 0, 0, '\0', '\0'))
    {  CLEANUP; return RM_STOP;  }

    CLEANUP;  return RM_FND;
}
```

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W