
# Using the Callout Gateway

Version 2018.1
2024-05-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

*Using the Callout Gateway*
Caché Version 2018.1 2024-05-02


For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Response Center (WRC)**
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

# Table of Contents

# 1

# About This Book

See the Table of Contents for a detailed listing of the subjects covered in this document.

The InterSystems Callout Gateway (implemented in the Caché **$ZF** functions) allows you to invoke external programs and system services from Caché. It also provides a way to integrate your custom code into Caché.

This book addresses the following topics:

- Introduction — introduces concepts important to understanding the rest of this book, summarizes the capabilities of the InterSystems Callout Gateway, and provides links to other resources.
- Running Programs or System Commands with $ZF(-100) — describes how to use a $ZF option that provides functionality similar to a command line interface.
- Creating an InterSystems Callout Library — describes how to create a Callout shared library with functions that can be invoked from Caché. Code can be written in any language that supports C/C++ calling conventions.
- Invoking Callout Library Functions — describes how to call functions dynamically from a Callout shared library.
- Statically Linked Callout Functions — describes how to create a customized version of Caché that includes your Callout functions.
- Special Considerations for UNIX® — discusses issues relevant only to UNIX® and related operating systems.
- InterSystems Callout Quick Reference — provides a detailed description of each $ZF function.

For general information, see *Using InterSystems Documentation*.

# 2

# Introduction

The InterSystems Callout Gateway allows Caché applications to invoke shell or operating system commands, run external programs in spawned processes, and call functions from specially written shared libraries. The Callout Gateway is implemented as a suite of related functions contained in the Caché **$ZF** system function.

This chapter discusses the following topics:

- Callout Gateway Concepts and Terminology — is a list of things you should know before reading the rest of this book.
- Overview of $ZF Functions — is a quick summary of the capabilities provided by the Callout Gateway.
- Compatible Languages and Compilers — discusses some of the software required to take full advantage of the Callout Gateway.

## 2.1 Callout Gateway Concepts and Terminology

Here are some important concepts that you should understand before reading the rest of this book:

### Identifying individual $ZF functions

Individual functions in the **$ZF** suite are identified by the first argument of the function call, which will be a negative number, `-100` or `-3` through `-6`. For example, the function that calls an operating system command has the form `$ZF(-100, <oscommand>)`, where `<oscommand>` is a string containing the command to be executed. When this function is discussed, it will be referred to as **$ZF(-100)**. In the same way, the other functions will be referred to as **$ZF(-3)** through **$ZF(-6)**, using only the first parameter of the actual function call.

The **$ZF(-4)** function is a special case, since it is actually just a container for eight utility functions that are identified by the first two parameters: **$ZF(-4,1)** through **$ZF(-4,8)**.

For a brief summary of these functions, see "Overview of $ZF Functions" later in this chapter. See the "InterSystems Callout Quick Reference" for a complete list of all **$ZF** functions, information on how they are used, and links to more detailed information and examples.

### Callout Libraries

In this book, the term *shared library* refers to a dynamically linked file (a DLL file on Windows or an SO file on UNIX® and related operating systems). A *Callout library* is a shared library that includes hooks to the Callout Gateway, allowing various **$ZF** functions to load it at runtime and invoke its functions. Callout libraries are usually written in C (see "Compatible Languages and Compilers"), but could potentially be written in any other compiled language that uses a calling convention understood by your C compiler.

See "Creating an InterSystems Callout Library" for details about how to write a Callout library.

### Callout library interfaces

All of the **$ZF** functions except **$ZF(-100)** are used to provide some form of access to Callout libraries. The **$ZF(-3)**, **$ZF(-5)**, and **$ZF(-6)** functions provide three different interfaces for invoking Callout library functions, and **$ZF(-4)** is a container for various utility functions used with the **$ZF(-5)** and **$ZF(-6)** interfaces.

See "Invoking Callout Library Functions" for a details of the various ways to access a Callout library.

# 2.2 Overview of $ZF Functions

The **$ZF** function suite includes the following functions:

### The $ZF(-100) function

The **$ZF(-100)** function is used to run shell commands and operating system service calls. It is not used to access Callout libraries, and can be called without any previous setup.

See "Running Programs or System Commands with $ZF(-100)" for details.

### The $ZF(-3) function

The **$ZF(-3)** function is a simple way to load a Callout library and invoke a library function with a single statement. Both the library and its functions are specified by name, and the library remains in memory until replaced by a call to a different library.

See "Using $ZF(-3) for Simple Library Function Calls" for details.

### The $ZF(-4) function

The **$ZF(-4)** function just is a container for a set of eight utility functions. The **$ZF(-5)** function interface uses functions **$ZF(-4,1)** through **$ZF(-4,3)**, and the **$ZF(-6)** function interface uses functions **$ZF(-4,5)** through **$ZF(-4,8)**. See the following descriptions for more details.

### The $ZF(-5) function interface

The **$ZF(-5)** function and its utility functions allow multiple libraries to be handled efficiently. Both the library and its functions are identified by system-defined ID values. Several libraries can be in virtual memory at the same time. The following **$ZF(-4)** functions are used to load and unload libraries, and to obtain library and function ID values:

- **$ZF(-4,1)** loads a library specified by name, and returns a library ID.
- **$ZF(-4,2)** unloads a library.
- **$ZF(-4,3)** returns a function ID for a specified library ID and function name.

See "Using $ZF(-5) to Access Libraries by System ID" for details.

### The $ZF(-6) function interface

The **$ZF(-6)** function and its utility functions provide a way to write Callout applications that do not require hard-coded library names. Instead, the actual library filenames are contained in a separate index table, where each library is associated with a unique, user-defined index number. Once the index table is defined, it is available to

all processes in an instance of Caché. Callout applications identify a library by index number and load it by reading the index table. Several libraries can be in memory at the same time. The following functions are used to manage indexes and load or unload libraries:

- **$ZF(-6)** invokes a library function, and loads the library if it is not already in memory.
- **$ZF(-4,4)** unloads a library.
- **$ZF(-4,5)** and **$ZF(-4,6)** are used to create and maintain the system index table, which can be accessed by all processes in an instance of Caché.
- **$ZF(-4,7)** and **$ZF(-4,8)** are used to create and maintain a process index table, which can be used to override the system index within a single process.

See "Using $ZF(-6) to Access Libraries by User Index" for details.

### The $ZF() function

Although your C function code will usually be used to generate a Callout library, it could also be directly linked into a custom version of the Caché executable. The **$ZF()** function (with no negative number argument) is the interface for invoking statically linked Callout functions. Unlike **$ZF(-3)**, **$ZF(-5)**, or **$ZF(-6)**, it does not need to specify an external library identifier, so statically linked functions can be called by just specifying the function name and arguments.

See "Statically Linked Callout Functions" for details.

# 2.3 Compatible Languages and Compilers

Using the Callout Gateway you can call routines written in languages other than ObjectScript. On all platforms that support Caché you can call routines written in the C language. In theory you should be able to call routines written in any compiled language that is compatible with C. Two compatibility issues arise. First, the compiler must use an Application Binary Interface (ABI) that is compatible with C. Second, the compiler must generate code that does not rely on any runtime library features that are not compatible with Caché.

InterSystems supports using the same C compiler that we use to generate Caché on all platforms:

| Platform | Compiler |
|---|---|
| IBM AIX | IBM XL C for AIX |
| Mac OS X (Darwin) | Xcode |
| Microsoft Windows | Microsoft Visual Studio |
| Linux (all variants) | GNU Project GCC C |

Most platforms have a standardized Application Binary Interface (ABI), making most compilers compatible. The Intel x86-32 and x86-64 platforms are major exceptions, Multiple calling conventions exist for these platforms. See (https://en.wikipedia.org/wiki/X86_calling_conventions) for a discussion of calling conventions on these platforms.

Calling languages that use different calling conventions may be possible, as many C compilers allow declaring an external routine to have another calling convention. Therefore is it often possible to call routines written in other languages if one is willing to write a C wrapper routine.

# 3

# Running Programs or System Commands with $ZF(-100)

The **$ZF(-100)** function permits a Caché process to invoke an executable program or a command of the host operating system. This is the only InterSystems Callout function that can be used without a special Callout shared library (see "Creating an InterSystems Callout Library"). The following topics are discussed in this chapter:

- Introduction — Overview of **$ZF(-100)** syntax and functionality.
- Program Execution— Programs can optionally run asynchronously or within an operating system shell.
- Logging Commands and Redirecting Output — Optional settings can log commands or redirect I/O.
- Adding the %System_Callout:USE Privilege — Security settings higher than minimal require this privilege.

**Note:** **$ZF(-100)** replaces deprecated functions **$ZF(-1)** and **$ZF(-2)**, and should be preferred in all cases.

## 3.1 Introduction

**$ZF(-100)** provides functionality similar to that of a command line interface, allowing you to invoke an executable program or a command of the host operating system. The syntax of this function is:

`sc = $ZF(-100,` *keywords*, *command*, *arguments* `)`

The first argument must be a literal `-100`. The other three arguments specify the following information:

- *keywords* — a string containing keywords that specify various options. For example, the string `"/ASYNC/LOGCMD"` specifies that the program should run asynchronously and write the command line to a log file.
- *command* — a string specifying the program or system command to invoke. If the full path to an executable is not specified, the operating system will apply standard search path rules.
- *arguments* — command arguments are specified as a series of comma-delimited expressions (as demonstrated in the example below).

The **$ZF(-100)** function returns an exit status code determined by the operating system and the program that was invoked.

The following example passes three strings to the `echo` command and then displays the status code. This example does not use any keywords, so the *keywords* argument is an empty string. The final command argument specifies a quoted string (following standard ObjectScript string rules):

```
USER>set status = $ZF(-100,"","echo","hello","world","""goodbye now""")
hello world "goodbye now"

USER>write status
0
```

The following sections provide more examples for various **$ZF(-100)** options. See "$ZF(-100): Running Programs or System Commands" in the Callout Quick Reference chapter for a summary of keywords and other options.

# 3.2 Program Execution

**$ZF(-100)** allows you to run a program or command either synchronously or asynchronously, with or without invoking the operating system shell. The default is to execute synchronously without invoking a shell. Default execution can be overridden by specifying optional keywords in the function call.

The following keywords can be used to control program execution:

- `/ASYNC` — Indicates that the program should run asynchronously, allowing the **$ZF(-100)** call to return without waiting for the program to complete.
- `/SHELL` — Indicates that the program should run in an operating system shell.

As mentioned in the last section, you can specify an empty string for the *keyword* parameter if you don't want to use either of these options. This example deliberately tries to list nonexistent files so that an error code `1` will be generated:

```
USER>set status = $ZF(-100,"", "ls","*.scala")
ls: cannot access *.scala: No such file or directory

USER>write st
1
```

If we run the same command asynchronously, the output is not displayed and *st* is undefined because no error code has been returned:

```
USER>kill st
USER>set status = $ZF(-100,"/ASYNC", "ls","*.scala")
USER>write st

WRITE st
^
<UNDEFINED> *st
```

See "Using I/O Redirection" in the next section for a way to redirect error output in situations like this.

# 3.3 Logging Commands and Redirecting Output

The following keywords control logging and I/O redirection:

- `/LOGCMD` — causes the program command and arguments to be sent to the console log.
- `/STDIN`, `/STDOUT`, and `/STDERR`— are used to redirect standard input, standard output, and standard error for the program invoked. These keywords must be followed by a file specification (see "Using I/O Redirection" below).

## 3.3.1 Logging Command Arguments

The `/LOGCMD` keyword causes command arguments and the exit status code to be logged in the console log (<install-dir>\mgr\messages.log). This is intended primarily as a debugging tool that allows you to see how expressions passed to **$ZF(-100)** were actually evaluated.

In most cases, the command and its arguments are logged on one line, and the return value on the next. For example, `set st=$ZF(-100,"/LOGCMD","echo","hello","world")` produces the following log entry under Windows:

```
03/28/18-11:49:51:898 (26171) 0 $ZF(-100) cmd=echo "hello" "world"
03/28/18-11:49:51:905 (26171) 0 $ZF(-100) ret=0
```

However, on UNIX® when `/SHELL` is not specified, values are logged one per line:

```
03/28/18-12:09:22:243 (26171) 0 $ZF(-100) argv[0]=echo
03/28/18-12:09:22:500 (26171) 0 $ZF(-100) argv[1]=hello
03/28/18-12:09:22:559 (26171) 0 $ZF(-100) argv[2]=world
03/28/18-12:09:22:963 (26171) 0 $ZF(-100) ret=0
```

In either case, arguments are logged exactly as they are received by the program.

## 3.3.2 Using I/O Redirection

The following keywords and file specifiers control I/O redirection:

- `/STDIN=`*input-file*
- `/STDOUT=`*output-file* or `/STDOUT+=`*output-file*
- `/STDERR=`*error-file* or `/STDERR+=`*error-file*

I/O redirection keywords are followed by an operator ( `=` or `+=` ) and a filename or file path. Spaces are permitted around the operators. Standard input should point to an existing file. The standard output and standard error files are created if they don't exist and are truncated if they already exist. Use the = operator to create or truncate a file, or the += operator to append to an existing file. To make standard error and standard output to go to the same file, specify the same file for both keywords.

In the following example, the first line redirects the standard output from the `echo` command to file temp.txt., and the second line displays the resulting file contents:

```
USER>set st = $ZF(-100,"/STDOUT=""temp.txt""","echo","-e","three\ntwo\none\nblastoff")
USER>set st = $ZF(-100,"","cat","temp.txt")
three
two
one
blastoff
```

In this next example, we display two lines of temp.txt in a different way, by redirecting the file to standard input. The `tail` command accepts the input and displays the last two lines:

```
USER>set st=$ZF(-100,"/STDIN=""temp.txt""","tail","-n2")
one
blastoff
```

This final example redirects standard error to temp.txt, and attempts to display a nonexistent file. It also uses the `/ASYNC` keyword to run the command asynchronously, causing the **$ZF(-100)** call to return before the error message can be displayed.

The second line (identical to the previous example) again displays the last two lines of the file, which now contain the redirected error message:

```
USER>set st = $ZF(-100,"/ASYNC /STDERR+=""temp.txt""","cat","nosuch.file")

USER>set st=$ZF(-100,"/STDIN=""temp.txt""","tail","-n2")
blastoff
cat: nosuch.file: No such file or directory
```

# 3.4 Adding the %System_Callout:USE Privilege

If your InterSystems security setting is higher than minimal, 0 **$ZF(-100)** requires the **%System_Callout:USE** privilege.

The **%System_CallOut** resource is already assigned to the **%Developer** Role, which you will have if you selected the Developer setup during installation. The following procedures describe how to assign this resource if you do not already have it:

### Assigning a Role to a User

Use the following procedure to assign the **%Developer** role to a User:

1. Open the Management Portal, go to System Administration > Security Management > Users and click Edit on the user description you want to use, then select the [Roles] tab.
2. Move the **%Developer** role from the Available column to the Selected column and click Assign. The role will appear in the list of roles assigned to this user. (If the **%Developer** role does not appear in the Available column, check the list of roles to see if you already have this role).
3. Click the Profile button. The **%Developer** role should be listed on the Roles: line.

### Creating a New Role

In some cases, it may be desirable to have a role that allows use of **$ZF(-100)** but does not grant any other privileges. Use the following procedure to create a new role that grants only the **%System_CallOut:USE** privilege:

1. Open the Management Portal and go to System Administration > Security Management > Roles.
2. Click the *Create New Role* button to bring up the Edit Roles page.
3. Fill in the name and description:
   - Name: UseCallout
   - description: Grants privilege to use %System_CallOut resource

   When you click Save, an Add button appears on the form.
4. Click the Add button to pop up a scrolling list of resources, select **%System_CallOut** from the list, and click Save. Click Close on the Edit Role form.
5. On the Roles page, the new **UseCallout** role is now in the list of role definitions.

# 4

# Creating an InterSystems Callout Library

An InterSystems Callout library is a shared library that contains your custom Callout functions and the enabling code that allows Caché to use them. This chapter describes how to create a Callout library and access it at runtime.

The following topics are discussed:

- Introduction to Callout Libraries — describes how a Callout library is created and accessed.
- ZFEntry Linkage Options — provides a detailed description of the linkage options that determine how function arguments are passed.
- Callout Library Runup and Rundown Functions — describes two optional functions that will be called automatically when a Callout library is loaded or unloaded.
- Troubleshooting — describes some coding practices that should be avoided.

**Note:** **Shared Libraries and Callout Libraries**

In this book, the term *shared library* refers to a dynamically linked file (a DLL file on Windows or an SO file on UNIX® and related operating systems). A *Callout library* is a shared library that includes hooks to the Callout Gateway, allowing it to be loaded and accessed at runtime by various **$ZF** functions.

## 4.1 Introduction to Callout Libraries

There are several different ways to access a Callout library from ObjectScript code, but the general principal is to specify the library name, the function name, and any required arguments (see "Invoking Callout Library Functions"). For example, the following code invokes a simple Callout library function:

### Invoking function AddInt from Callout library simplecallout.dll

The following ObjectScript code is executed at the Terminal. It loads a Callout library named simplecallout.dll and invokes a library function named **AddInt**, which adds two integer arguments and returns the sum.

```
USER> set sum = $ZF(-3,"simplecallout.dll","AddInt",2,2)
USER> write "The sum is ",sum,!
The sum is 4
```

This example uses **$ZF(-3)**, which is the simplest way to invoke a single Callout library function. See "Invoking Callout Library Functions" for other options.

The simplecallout.dll Callout library is not much more complex than the code that calls it. It contains three elements required by all Callout libraries:

1. Standard code provided when you include the cdzf.h Callout header file.
2. One or more functions with correctly specified parameters.
3. Macro code for a *ZFEntry table*, which generates the mechanism that Caché will use to locate your Callout functions when the library is loaded (see "Creating a ZFEntry Table" for details).

Here is the code that was compiled to produce the simplecallout.dll Callout library:

### Callout code for simplecallout.dll

```
#define ZF_DLL  /* Required only for dynamically linked libraries. */
#include <cdzf.h>  /* Required for all Callout code. */

int AddTwoIntegers(int a, int b, int*outsum) {
   *outsum = a+b;   /* set value to be returned by the $ZF function call */
   return 0;   /* set the exit status code */
}
ZFBEGIN
   ZFENTRY("AddInt","iiP",AddTwoIntegers)
ZFEND
```

- The first line must define `ZF_DLL`, which is the switch that indicates we are creating a dynamically linked Callout library rather than statically linking the code into Caché (as described later in "Statically Linked Callout Functions").
- The second line includes the cdzf.h file, required for both dynamic and static linked code.
- The **AddTwoIntegers()** function is defined next. It has the following features:
  - Two input parameters, integers *a* and *b*, and one output parameter, integer pointer **outsum*.
  - A statement assigning a value to output parameter **outsum*. This will be the value returned by the call to **$ZF(-3)**.
  - The `return` statement does not return the function output value. Instead, it specifies the exit status code that will be received by Caché if the **$ZF** call is successful. If the function fails, Caché will receive an exit status code generated by the system.
- The last three lines are macro calls that generate the ZFEntry table used by Caché to locate your Callout library functions. This example has only a single entry, where:
  - `"AddInt"` is the string used to identify the function in a **$ZF** call.
  - `"iiP"` is a string that specifies datatypes for the two input values and the output value.
  - `AddTwoIntegers` is the entry point name of the C function.

The ZFEntry table is the mechanism that allows a shared library to be loaded and accessed by the Callout Gateway (see "Creating a ZFEntry Table"). A `ZFENTRY` declaration specifies the interface between the C function and the ObjectScript **$ZF** call. Here is how the interface works in this example:

- *The C function declaration* specifies three parameters:

  ```
  int AddTwoIntegers(int a, int b, int*outsum)
  ```

  Parameters *a* and *b* are the inputs, and *outsum* will receive the output value. The return value of **AddTwoIntegers** is an exit status code, not the output value.

- *The ZFENTRY macro* defines how the function will be identified in Caché, and how the parameters will be passed:

  ```
  ZFENTRY("AddInt","iiP",AddTwoIntegers)
  ```

  `"AddInt"` is the library function identifier used to specify C function **AddTwoIntegers** in a **$ZF** call. The linkage declaration (`"iiP"`) declares parameters *a* and *b* as linkage type `i` (input-only integers), and *outsum* as linkage type `P` (an integer pointer that can be used for both input and output).

- *The **$ZF(-3)** function call* specifies the library name, the library function identifier, and the input parameters, and returns the value of the output parameter:

  ```
  set sum = $ZF(-3,"simplecallout.dll","AddInt",2,2)
  ```

  Parameters *a* and *b* are specified by the last two arguments. No argument is required for the third parameter, *outsum*, because it is used only for output. The value of *outsum* is assigned to *sum* when the **$ZF(-3)** call returns .

### 4.1.1 Creating a ZFEntry Table

Every Callout library must define a ZFEntry table, which allows Caché to load and access your Callout functions (see the "Introduction to Callout Libraries" for a simple example). The ZFEntry table is generated by a block of macro code beginning with **ZFBEGIN** and ending with **ZFEND**. Between these two macros, a **ZFENTRY** macro must be called once for each function to be exposed.

Each **ZFENTRY** call takes three arguments:

```
zfentry(zfname,linkage,entrypoint)
```

where *zfname* is the string used to specify the function in a **$ZF** call, *linkage* is a string that specifies how the arguments are to be passed, and *entrypoint* is the entry point name of the C function.

By default, the ZFEntry table will generate code that can only be used for statically linked functions (see "Statically Linked Callout Functions" for details). To create a Callout library, your code must contain a `#define ZF_DLL` directive, which is a switch that causes the macro code to generate a different mechanism for locating library functions. Instead of a static pointer table, it generates an internal **GetZFTable** function. When a Callout library is loaded, Caché calls this function to initialize the library for subsequent lookups of library function names.

**Note:** **ZFEntry Sequence Numbers**

The position of an entry in the ZFEntry table can be significant. The **$ZF(-5)** and **$ZF(-6)** interfaces (described in the next chapter, "Invoking Callout Library Functions") both invoke a library function by specifying its sequence number (starting with `1`) in the table. For example, **$ZF(-6)** would invoke the third function in a ZFEntry table with the following call:

```
x = $ZF(-6,libID,3)
```

where *libID* is the library identifier and `3` is the sequence number of the third entry in the table.

## 4.2 ZFEntry Linkage Options

Each **ZFENTRY** statement (see "Creating a ZFEntry Table") requires a string that determines how function arguments are passed. This section provides a detailed description of available linkage options.

- Introduction to Linkages — provides an overview of the various linkage types and lists all of the linkage options discussed in this chapter.

- Using Numeric Linkages — describes linkage options for numeric parameters.
- Passing Null Terminated Strings with C Linkage Types — describes linkage options for null terminated strings.
- Passing Short Counted Strings with B Linkage Types — describes linkages that use the ZARRAY structure for counted character arrays.
- Passing Standard Counted Strings with J Linkage Types — describes linkages that use the Caché CACHE_EXSTR structure for counted character arrays.
- Configuring the $ZF Heap for Short Strings — describes Caché system settings that control memory allocation for legacy short string parameter passing.

## 4.2.1 Introduction to Linkages

Each **ZFENTRY** statement (see "Creating a ZFEntry Table") requires a string that describes how the arguments are passed. For example, `"iP"` specifies two parameters: an integer, and a pointer to an integer. The second letter is capitalized to specify that the second argument may used for both input and output. Your code can have up to 32 actual and formal parameters.

If you specify an uppercase linkage type (permitted for all linkage types except `i`), the argument can be used for both input and output. If only one output argument is specified, its final value will be used as the return value of the function. If more than one output argument is specified, all output arguments will be returned as a comma-delimited string.

Output arguments do not have to be used as input arguments. If you specify output-only arguments after all input arguments, the function can be called without specifying any of the output arguments (see "Introduction to Callout Libraries" for an example).

From the prospective of the ObjectScript programmer, parameters are input only. The values of the actual parameters are evaluated by the **$ZF** call and linked to the formal parameters in the C routine declaration. Any changes to the C formal parameters are either lost or are available to be copied to the **$ZF** return value.

If the **ZFENTRY** macro does not specify a formal parameter to be used as the return value, the **$ZF** call will return an empty string (`""`). The linkage declaration can contain more than one output parameter. In this case, all the return values will be converted to a single comma-delimited string. There is no way to distinguish between the comma inserted between multiple return parameters, and a comma present in any one return value, so only the final return value should contain commas.

The following table describes the available options:

| C Datatype | Input | In/Out | Notes |
|---|---|---|---|
| int | `i` | none (use `P`) | The `i` linkage type is input only. To return an integer type, use `P` (int *). Input argument may be a numeric string (see note 1). |
| int * | `p` | `P` | Input argument may be a numeric string (see note 1). |
| double * | `d` | `D` | Input argument may be a numeric string (see note 1). Use `#D` to preserve a double * in radix 2 format (see note 2). |
| float * | `f` | `F` | Input argument may be a numeric string (see note 1). Use `#F` to preserve a float * in radix 2 format (see note 2). |
| char * | `1c` or `c` | `1C` or `C` | This is the common C NULL-terminated string (see note 3). |
| unsigned short * | `2c` or `w` | `2C` or `W` | This is a C style NULL-terminated UTF-16 string (see note 3). |
| wchar t * | `4c` | `4C` | This is a C style NULL-terminated string stored as a vector of wchar_t elements (see notes 3 and 4). |

| C Datatype | Input | In/Out | Notes |
|---|---|---|---|
| ZARRAYP | 1b or b | 1B or B | short 8-bit national strings (up to 32,767 characters). |
| ZWARRAYP | 2b or s | 2B or S | short 16-bit Unicode strings (up to 32,767 characters). |
| ZHARRAYP | 4b | 4B | short Unicode strings (up to 32,767 characters) stored in elements implemented by wchar_t (see note 4) |
| CACHE_EXSTR | 1j or j | 1J or J | Standard string (up to 3,641,144 characters) of 8-bit national characters |
| CACHE_EXSTR | 2j or n | 2J or N | Standard string (up to 3,641,144 characters) of 16-bit Unicode characters |
| CACHE_EXSTR | 4j | 4J | Standard string (up to 3,641,144 characters) of wchar_t characters (see note 4) |

1. i, p, d, f — When numeric arguments are specified, Caché allows the input argument to be a string. See "Using Numeric Linkages" for details.
2. #F, #D— To preserve a number in radix 2 floating point format, use #F for float * or #D for double *. See "Using Numeric Linkages" for details.
3. 1C, 2C, 4C — All strings passed with this linkage will be truncated at the first null character. See "Passing Null Terminated Strings with C Linkage Types" for details.
4. 4B, 4C, 4J— Although wchar_t is typically 32 bits, Caché uses only 16 bits to store each Unicode character, so these linkages are useful only when interacting with existing code or system interfaces that specify use of wchar_t.

Structure and argument prototype definitions (including InterSystems internal definitions) can be seen in the include file cdzf.h.

## 4.2.2 Using Numeric Linkages

Numeric linkage types are provided for the following datatypes:

| C Datatype | Input | In/Out | Notes |
|---|---|---|---|
| int | i | none (use P) | The i linkage type is input only. To return an integer type, use P instead. |
| int * | p | P | Pointer to int. |
| double * | d | D | Use #D (output only) to return a double * in radix 2 format. |
| float * | f | F | Use #F (output only) to return a float * in radix 2 format. |

When numeric arguments are specified, Caché allows the input argument to be a string. When a string is passed, a leading number will be parsed from string to derive a numeric value. If there is no leading number, the value 0 will be received. Thus "2DOGS" is received as 2.0, while "DOG" is received as 0.0. Integer arguments are truncated. For example, "2.1DOGS" is received as 2. For a detailed discussion of this subject, see "String-to-Number Conversion" in *Using ObjectScript*.

**Note:** **Preserving Accuracy in Floating Point Numbers**

When the output linkage is specified by `F` (float *) or `D` (double *), the number you return will be converted to an internal radix 10 number format. To preserve the number in radix 2 format, use `#F` for float * or `#D` for double *.

The # prefix is not permitted for input arguments. In order to avoid conversion, input values must be created with **$DOUBLE** in the ObjectScript code that calls the function, and the corresponding input linkages must be specified as lower case `f` or `d`.

By default, Caché stores fractional numbers in base 10 form. For example, the number `1.3` is stored as `13*.10`. In contrast, most high-level languages store fractional numbers in base 2 form (a binary number times some power of 2). If you describe an argument as `f` (float *) or `d` (double *), Caché converts its value from base 10 to base 2 on input and back to base 10 on output. This conversion may introduce slight inaccuracies.

Caché supports the **$DOUBLE** function for creating a standard IEEE format 64-bit floating point number. These numbers can be passed between external functions and Caché without any loss of precision (unless the external function uses the 32-bit float instead of the 64-bit double). For output, the use of the IEEE format is specified by adding the prefix character # to the `F` or `D` argument type. For example, `"i#D"` specifies an argument list with one integer input argument and one 64-bit floating point output argument.

## 4.2.3 Passing Null Terminated Strings with C Linkage Types

This linkage type should be used only when you know Caché will not send strings containing null (`$CHAR(0)`) characters. When using this datatype, your C function will truncate a string passed by Caché at the first null character, even if the string is actually longer. For example, the string `"ABC"_$CHAR(0)_"DEF"` would be truncated to `"ABC"`.

| C Datatype | Input | In/Out | Notes |
|---|---|---|---|
| char * | `1c` or `c` | `1C` or `C` | This is the common C NULL-terminated string. |
| unsigned short * | `2c` or `w` | `2C` or `W` | This is a C style NULL-terminated UTF-16 string. |
| wchar t * | `4c` | `4C` | This is a C style NULL-terminated string stored as a vector of wchar_t elements. |

Here is a short Callout library that uses all three linkage types to return a numeric string:

### Using C linkages to pass null-terminated strings

Each of the following three functions generates a random integer, transforms it into a numeric string containing up to 6 digits, and uses a C linkage to return the string .

```
#define ZF_DLL   // Required when creating a Callout library.
#include <cdzf.h>
#include <stdio.h>
#include <wchar.h>   // Required for 16-bit and 32-bit strings

int get_sample(char* retval) {  // 8-bit, null-terminated
   sprintf(retval,"%d",(rand()%1000000));
   return ZF_SUCCESS;
}

int get_sample_W(unsigned short* retval) {  // 16-bit, null-terminated
   swprintf(retval,6,L"%d",(rand()%1000000));
   return ZF_SUCCESS;
}

int get_sample_H(wchar_t* retval) {  // 32-bit, null-terminated
   swprintf(retval,6,L"%d",(rand()%1000000));
   return ZF_SUCCESS;
}

ZFBEGIN
ZFENTRY("GetSample","1C",get_sample)
```

```
ZFENTRY("GetSampleW","2C",get_sample_W)
ZFENTRY("GetSampleH","4C",get_sample_H)
ZFEND
```

## 4.2.4 Passing Short Counted Strings with B Linkage Types

The cdzf.h Callout header file defines counted string structures ZARRAY, ZWARRAY, and ZHARRAY, representing a short string (an InterSystems legacy string type). These structures contain an array of character elements (8-bit, 16-bit Unicode, or 32-bit wchar t, respectively) and a short integer (maximum value 32,768) specifying the number of elements in the array. For example:

```
typedef struct zarray {
    unsigned short len;
    unsigned char data[1]; /* 1 is a dummy value */
   } *ZARRAYP;
```

where

- *len* — contains the length of the array
- *data* — is an array that contains the character data. Element types are unsigned char for ZARRAY, unsigned short for ZWARRAY, and wchar_t for ZHARRAY.

The B linkages specify pointer types ZARRAYP, ZWARRAYP, and ZHARRAYP, corresponding to the three array structures. The maximum size of the array returned is 32,767 characters.

| C Datatype | Input | In/Out | Notes |
|---|---|---|---|
| ZARRAYP | 1b or b | 1B or B | short national string containing up to 32,767 8-bit characters. |
| ZWARRAYP | 2b or s | 2B or S | short Unicode string containing up to 32,767 16-bit characters. |
| ZHARRAYP | 4b | 4B | short Unicode string containing up to 32,767 wchar_t characters. |

The maximum total length of the arguments depends on the number of bytes per character (see “Configuring the $ZF Heap”).

Here is a Callout library that uses all three linkage types to return a numeric string:

### Using B linkages to pass counted strings

Each of the following three functions generates a random integer, transforms it into a numeric string containing up to 6 digits, and uses a B linkage to return the string .

```
#define ZF_DLL   // Required when creating a Callout library.
#include <cdzf.h>
#include <stdio.h>
#include <wchar.h>   // Required for 16-bit and 16-bit characters

int get_sample_Z(ZARRAYP retval) {  // 8-bit, counted
   unsigned char numstr[6];
   sprintf(numstr,"%d",(rand()%1000000));
   retval->len = strlen(numstr);
   memcpy(retval->data,numstr,retval->len);
   return ZF_SUCCESS;
}

int get_sample_ZW(ZWARRAYP retval) {  // 16-bit, counted
   unsigned short numstr[6];
   swprintf(numstr,6,L"%d",(rand()%1000000));
   retval->len = wcslen(numstr);
   memcpy(retval->data,numstr,(retval->len*sizeof(unsigned short)));
   return ZF_SUCCESS;
}

int get_sample_ZH(ZHARRAYP retval) {  // 32-bit, counted
   wchar_t numstr[6];
```

```
   swprintf(numstr,6,L"%d",(rand()%1000000));
   retval->len = wcslen(numstr);
   memcpy(retval->data,numstr,(retval->len*sizeof(wchar_t)));
   return ZF_SUCCESS;
}


ZFBEGIN
ZFENTRY("GetSampleZ","1B",get_sample_Z)
ZFENTRY("GetSampleZW","2B",get_sample_ZW)
ZFENTRY("GetSampleZH","4B",get_sample_ZH)
ZFEND
```

**Note:** Commas are used as separators in an output argument string that contains multiple values. Because commas can also be a part of counted string arrays, declare these arrays at the *end* of the argument list and use one array per call.

## 4.2.5 Passing Standard Counted Strings with J Linkage Types

The callin.h header file defines counted string structure CACHE_EXSTR, representing a standard InterSystems string. This structure contains an array of character elements (8-bit, 16-bit Unicode, or 32–bit wchar t) and an int value (maximum value 3,641,144) specifying the number of elements in the array:

```
typedef struct {
   unsigned int   len;         /* length of string */
   union {
      Callin_char_t  *ch;      /* text of the 8-bit string */
      unsigned short *wch;     /* text of the 16-bit string */
      wchar_t        *lch;     /* text of the 32-bit string */
/* OR unsigned short *lch   if 32-bit characters are not enabled */
   } str;
} CACHE_EXSTR, *CACHE_EXSTRP;
```

| C Datatype | Input | In/Out | Notes |
|---|---|---|---|
| CACHE_EXSTR | 1j or j | 1J or J | Standard string of 8-bit national characters |
| CACHE_EXSTR | 2j or n | 2J or N | Standard string of 16-bit Unicode characters |
| CACHE_EXSTR | 4j | 4J | Standard string of 32-bit characters wchar_t characters |

Two InterSystems Callin functions (see the "Callin Function Reference" in *Using the Callin API*) are used to create or delete CACHE_EXSTR:

- **CacheExStrKill** — Releases the storage associated with a CACHE_EXSTR string.
- **CacheExStrNew**[**W**][**H**] — Allocates the requested amount of storage for a string, and fills in the CACHE_EXSTR structure with the length and a pointer to the value field of the structure.

See the following example for a demonstration of how these functions are used.

Here is a Callout library that uses all three linkage types to return a numeric string:

### Using J linkages to pass strings

Each of the following three functions generates a random integer, transforms it into a numeric string containing up to 6 digits, and uses a J linkage to return the string .

```
#define ZF_DLL   // Required when creating a Callout library.
#include <cdzf.h>
#include <stdio.h>
#include <wchar.h>
#include <callin.h>

int get_sample_L(CACHE_EXSTRP retval) {  // 8-bit characters
   Callin_char_t numstr[6];
```

```
   size_t len = 0;
   sprintf(numstr,"%d",(rand()%1000000));
   len = strlen(numstr);
   CACHEEXSTRKILL(retval);
   if (!CACHEEXSTRNEW(retval,len)) {return ZF_FAILURE;}
   memcpy(retval->str.ch,numstr,len);   // copy to retval->str.ch
   return ZF_SUCCESS;
}

int get_sample_LW(CACHE_EXSTRP retval) {  // 16-bit characters
   unsigned short numstr[6];
   size_t len = 0;
   swprintf(numstr,6,L"%d",(rand()%1000000));
   len = wcslen(numstr);
   CACHEEXSTRKILL(retval);
   if (!CACHEEXSTRNEW(retval,len)) {return ZF_FAILURE;}
   memcpy(retval->str.wch,numstr,(len*sizeof(unsigned short)));   // copy to retval->str.wch
   return ZF_SUCCESS;
}

int get_sample_LH(CACHE_EXSTRP retval) {  // 32-bit characters
   wchar_t numstr[6];
   size_t len = 0;
   swprintf(numstr,6,L"%d",(rand()%1000000));
   len = wcslen(numstr);
   CACHEEXSTRKILL(retval);
   if (!CACHEEXSTRNEW(retval,len)) {return ZF_FAILURE;}
   memcpy(retval->str.lch,numstr,(len*sizeof(wchar_t)));   // copy to retval->str.lch
   return ZF_SUCCESS;
}

ZFBEGIN
ZFENTRY("GetSampleL","1J",get_sample_L)
ZFENTRY("GetSampleLW","2J",get_sample_LW)
ZFENTRY("GetSampleLH","4J",get_sample_LH)
ZFEND
```

**Note:** **Always kill CACHE_EXSTRP input arguments**

In the previous example, `CACHEEXSTRKILL(retval)` is always called to remove the input argument from memory. This should always be done, even if the argument is not used for output. Failure to do so may result in memory leaks.

## 4.2.6 Configuring the $ZF Heap for Short Strings

**Note:** This section applies only to legacy short strings (see "Passing Short Counted Strings with B Linkage Types"). Standard InterSystems strings (see "Passing Standard Counted Strings with J Linkage Types") use their own stack.

The **$ZF** heap is the virtual memory space allocated for all **$ZF** short string input and output parameters. It is controlled by the following Caché system settings:

- *ZFString* is the number of characters permitted for a single string parameter. The number of bytes this actually requires will vary depending on whether you are using 8-bit characters, 16-bit Unicode characters, or 32-bit characters on UNIX®. The permitted range for this setting is `0` to `32767` characters. The default is `0`, indicating that the maximum value should be used.
- *ZFSize* is the total number of bytes Caché allocates for all **$ZF** input and output parameters. The permitted range for this setting is `0` to `270336` bytes, where `0` (the default setting) indicates that Caché should calculate an appropriate value based on the value of *ZFString*.

Calculate *ZFSize* (total number of bytes) based on *ZFString* (maximum number of characters per string) as follows:

```
ZFSize = (<bytes per character> * ZFString) + 2050
```

For example, suppose *ZFString* has the default value of 32767 characters:

- Using Unicode 16-bit characters, an appropriate value for *ZFSize* is `(2 * 32767 + 2050) = 67584` bytes.

- Using UNIX® 32-bit characters, an appropriate value for *ZFSize* is `(4 * 32767 + 2050) = 133118` bytes.

These settings can be changed in either of the following places:

- The configuration parameter file (see "zfheap" in the "[config]" section of the *Configuration Parameter File Reference*)
- The Management Portal (see the *ZFSize* and *ZFString* entries under "Advanced Memory Settings" in the *Additional Configuration Settings Reference*).

# 4.3 Callout Library Runup and Rundown Functions

An InterSystems Callout library can include custom internal functions that will be called when the shared object is loaded (runup) or unloaded (rundown). No arguments are passed in either case. The functions are used as follows:

- **ZFInit** — is invoked when a Callout library is first loaded by **$ZF(-3)**, **$ZF(-4,1)**, or **$ZF(-6)**. The return code from this function should be zero to indicate absence of error, or non-zero to indicate some problem. If the call was successful, the address of your **ZFUnload** rundown function is saved.
- **ZFUnload** — is invoked when a Callout library is unloaded or replaced by a call to **$ZF(-3)**, or is unloaded by **$ZF(-4,2)** or **$ZF(-4,4)**. It is not invoked at process halt. If some error occurs during the rundown function, further calls to it will be disabled to allow unloading of the Callout library. The return value from **ZFUnload** is currently ignored.

When building the Callout library, you may need to explicitly export the symbols **ZFInit** and **ZFUnload** during the link procedure.

# 4.4 Troubleshooting

Just because you can call almost any routine with the Callout Gateway doesn't mean you should. It is best used for math functions, and can be used effectively for interface to external devices not well handled with Caché I/O, or for some system services where a Caché interface does not otherwise exist.

The following actions can cause serious problems:

- *Accessing any memory that doesn't belong to you*

  Memory access violations will be handled by Caché, and will be treated as bugs in Caché.

- *Encountering any other errors handled by traps*

  Errors handled by traps (such as divide by zero errors on most platforms) will also be treated as bugs in Caché.

- *Changing your processes priority*

  Caché needs to interact with other processes running Caché. Lowering your priority can be just as bad as raising it. For example, imagine that your process acquires a spin-lock protected resource just before relinquishing the CPU. If your priority is too low, other processes with higher priority can fight for the resource, effectively preventing your process running so it can release the spin-lock.

- *Masking interrupts*

  You might mask interrupts very briefly to implement your own interlock, but you should be very careful to not leave interrupts masked for any period of time.

- *Creating or opening any resource that you can't clean-up*

It is fine to open files, and allocate memory with malloc, because those resources will be closed or freed upon termination of your process. If you create a second thread, you can't guarantee the second thread will exit gracefully before the Caché process exits, so don't create a second thread.

- *Returning non-opaque objects from your non-ObjectScript code*

  Don't malloc a block of memory in your code and expect to be able to use `$VIEW(address,-3,size)` to read it. Also, you should not pass a malloc block back to your non-ObjectScript code. Your code should return an opaque handle, and later when it receives an opaque handle it should verify that it is valid before using it.

- *Exiting your process*

  You should never just call `exit`. Always return with either `ZF_SUCCESS` or `ZF_FAILURE` (and remember that the implementation of these values differs among Caché platforms).

- *Exiting by calling any variant of exec*

  You can fork and then call exec in the child process, but be very sure that the parent will always return to Caché and the child process will never return to Caché.

- *Changing the error handling behavior for the process*

  Unlike other operating systems, UNIX® systems only allow you to establish local error handling for the current and inner frames.

# 5

# Invoking Callout Library Functions

The Callout Gateway provides three different interfaces that can be used to load an InterSystems Callout library at runtime and call functions from that library. This chapter covers the following topics:

- Callout Library Interfaces — provides an overview of the three Callout library interfaces and how they are used.
- Using $ZF(-3) for Simple Library Function Calls — describes **$ZF(-3)**, which provides a convenient way to load and use one library at a time.
- Using $ZF(-5) to Access Libraries by System ID — describes **$ZF(-5)** and related functions, which provide a more efficient way to load and use libraries.
- Using $ZF(-6) to Access Libraries by User Index — describes **$ZF(-6)** and related functions, which provide an interface similar to **$ZF(-5)** except that they load libraries by reading a user-created index table rather then specifying hard-coded library filenames.

**Note:** **Shared libraries and Callout libraries**

In this book, the term *shared library* refers to a dynamically linked file (a DLL file on Windows or an SO file on UNIX® and related operating systems). A *Callout library* is a shared library that includes hooks to the Callout Gateway, allowing various **$ZF** functions to load it at runtime and invoke its functions. Instructions for writing Callout libraries are provided in "Creating an InterSystems Callout Library".

## 5.1 Callout Library Interfaces

The **$ZF(-3)**, **$ZF(-5)**, and **$ZF(-6)** functions are all used to invoke Callout library functions, but each of these functions has its own strengths and weaknesses.

### The $ZF(-3) Interface

The **$ZF(-3)** function loads a Callout library and invokes a library function by specifying the library file path and the function name. It is simple to use, but can only have one library at a time in virtual memory. Unlike the other interfaces, it does not require any initialization before a library function can be invoked. See "Using $ZF(-3) for Simple Library Function Calls" for details.

### The $ZF(-5) Interface

The **$ZF(-5)** function invokes Callout library functions by specifying system-defined numeric identifiers for the Callout library and the library functions. It can keep several libraries in memory at the same time, and individual function calls are much more efficient than **$ZF(-3)**. The **$ZF(-5)** interface also includes utility functions **$ZF(-4,1)**, **$ZF(-4,2)**, and **$ZF(-4,3)**, which must be used to obtain library and function IDs and to load or unload libraries. See "Using $ZF(-5) to Access Libraries by System ID" for details.

### The $ZF(-6) Interface

The **$ZF(-6)** interface is similar to **$ZF(-5)** except that Callout applications can load libraries without hard-coding the actual library filenames. Instead, a separate index table contains that information, and each indexed library is given a unique, user-defined index number. Once the index table is defined in an instance of Caché, it is available to all Callout applications in that instance. The **$ZF(-6)** function invokes a library function by specifying the index number and a function ID. If necessary, it automatically reads the index table and loads the specified library. The **$ZF(-6)** interface also includes utility functions **$ZF(-4,4)** through **$ZF(-4,8)**, which must be used to unload libraries and create or maintain indexes. See "Using $ZF(-6) to Access Libraries by User Index" for details.

# 5.2 Using $ZF(-3) for Simple Library Function Calls

The **$ZF(-3)** function is used to load a Callout library and execute a specified function from that library. **$ZF(-3)** is most useful if you are only using one library, or aren't making enough calls to worry about the overhead of loading libraries. It allows you to call any available library function by specifying the library name, the function name, and a comma-separated list of function arguments:

```
result = $ZF(-3, library_name[, function_name[, arguments]])
```

The specified library is loaded if it hasn't already been loaded by a previous call to **$ZF(-3)**. Only one library can be loaded at a time. When a subsequent **$ZF(-3)** call specifies a different library, the old library is unloaded and the new one replaces it. The library stays loaded as long as subsequent **$ZF(-3)** calls specify the same library. After a library has been loaded, the library name can be specified as a null string ("") in subsequent calls.

You can load or unload a library without calling a function. To load a new library, specify only the library name. To unload the current library without loading a new one, specify only a null string. In either case, **$ZF(-3)** returns a status code indicating whether the load or unload was successful.

The following ObjectScript code calls two different functions from each of two different libraries, and then unloads the current library:

#### Using $ZF(-3) to load libraries and call functions

**ObjectScript**

```
// define Callout library paths
set libOne = "c:\intersystems\cache\bin\myfirstlibrary.dll"
set libTwo = "c:\intersystems\cache\bin\anotherlibrary.dll"

//load and call
SET result1=$ZF(-3,libOne,"FuncA",123)   // loads libOne and calls FuncA
SET result2=$ZF(-3,"","FuncB","xyz")   // calls FuncB from same library

//load, then call with null name
SET status=$ZF(-3,libTwo)   // unloads libOne, loads libTwo
SET result1=$ZF(-3,"","FunctionOne","arg1")
SET result2=$ZF(-3,"","FunctionTwo","argA", "argB")

//unload
SET status=$ZF(-3,"")   // unloads libTwo
```

- For convenience, the library names are assigned to strings *libOne* and *libTwo*.
- The first call to **$ZF(-3)** loads Callout library *libOne* and invokes function **FuncA** from that library.
- The second call specifies a null string for the library name, indicating that currently loaded *libOne* should be used again, and invokes function **FuncB** from that library.
- The third call to **$ZF(-3)** specifies only library name *libTwo*. This unloads *libOne* and loads *libTwo*, but does not invoke any library functions. The call returns a status code indicating whether *libTwo* was successfully loaded.
- The fourth and fifth calls invoke library functions **FunctionOne** and **FunctionTwo** from currently loaded *libTwo*.
- The final **$ZF(-3)** call does not invoke a library function, and specifies a null string for the library name. This unloads *libTwo* and does not load a new library. The call returns a status code indicating whether *libTwo* was successfully unloaded.

The following sections of this chapter describe **$ZF** functions that can load more than one library at a time. These functions will not conflict with **$ZF(-3)**. You can always use **$ZF(-3)** as if it were loading and unloading its own private copy of a library.

# 5.3 Using $ZF(-5) to Access Libraries by System ID

The **$ZF(-5)** function uses system-defined library and function identifiers to invoke library functions. Utility functions **$ZF(-4,1)**, **$ZF(-4,2)** and **$ZF(-4,3)** are used to get the required identifiers and to load or unload libraries. Multiple libraries can be open at the same time. Unlike **$ZF(-3)** (see "Using $ZF(-3) for Simple Library Function Calls"), **$ZF(-5)** can not be used until the utility functions have been called to load libraries and get identifiers. However, each library only needs to be loaded once, and each library or function identifier only has to be generated once. In applications that make many library function calls, this can significantly reduce processing overhead.

The following **$ZF** functions are discussed in this section:

- **$ZF(-5)** — invokes a Callout library function referenced by system-defined library and function identifiers.
- **$ZF(-4,1)** — loads a Callout library specified by filename, and returns a system-defined ID value for it.
- **$ZF(-4,2)** — unloads a Callout library specified by library ID.
- **$ZF(-4,3)** — returns a function ID value for a given library ID and function name.

The **$ZF(-4,1)** and **$ZF(-4,3)** functions are used to load Callout libraries and get library and function identifiers. The syntax for **$ZF(-4,1)** is:

```
lib_id = $ZF(-4,1,lib_name)   // get library ID
```

where *lib_name* is the full name and path of the shared library file, and *lib_id* is the returned library ID. The syntax for **$ZF(-4,3)** is:

```
func_id=$ZF(-4,3,lib_id, func_name)   // get function ID
```

where *lib_id* is the library ID, *func_name* is the library function name, and *func_id* is the returned function ID value.

The following ObjectScript code loads Callout library mylibrary.dll and gets the library ID, then gets the function ID for `"MyFunction"` and invokes it with **$ZF(-5)**:

### Loading a library and invoking a function with $ZF(-5)

#### ObjectScript

```
set libID = $ZF(-4,1,"C:\calloutlibs\mylibrary.dll")
set funcID = $ZF(-4,3,libID, "MyFunction")
set x = $ZF(-5,libID, funcID, "arg1")
```

Once the identifiers have been defined, the library will remain loaded until unloaded by **$ZF(-4,2)**, and the identifiers can be used without any further calls to **$ZF(-4,1)** or **$ZF(-4,3)**. This eliminates a significant amount of processing overhead when functions from several libraries are invoked many times.

The following ObjectScript code loads two different libraries and invokes functions from both libraries in long loops. A function in inputlibrary.dll acquires data, and functions in outputlibrary.dll plot and store the data:

### Using $ZF(-5) with multiple libraries and many function calls

```
Method GraphSomeData(loopsize As %Integer=100000) As %Status
 {
   // load libraries and get system-defined ID values
   set InputLibID = $ZF(-4,1,"c:\intersystems\cache\bin\inputlibrary.dll")
   set OutputLibID = $ZF(-4,1,"c:\intersystems\cache\bin\outputlibrary.dll")
   set fnGetData = $ZF(-4,3,InputLibID,"GetData")
   set fnAnalyzeData = $ZF(-4,3,OutputLibID,"AnalyzeData")
   set fnPlotPoint = $ZF(-4,3,OutputLibID,"PlotPoint")
   set fnWriteData = $ZF(-4,3,OutputLibID,"WriteData")

   // call functions from each library until we have 100000 good data items
   do {
      set datapoint = $ZF(-5,InputLibID,fnGetData)
      set normalized = $ZF(-5,OutputLibID,fnAnalyzeData,datapoint)
      if (normalized>"") { set flatdata($INCREMENT(count)) = normalized }
   } while (count<loopsize)
   set status = $ZF(-4,2,InputLibID)   //unload "inputlibrary.dll"

   // plot results of the previous loop and write to output
   for point=1:1:loopsize {
      set list = $ZF(-5,OutputLibID,fnPlotPoint,flatdata(point))
      set x = $PIECE(list,",",1)
      set y = $PIECE(list,",",2)
      set sc = $ZF(-5,OutputLibID,fnWriteData,flatdata(point),x,y,"outputfile.dat")
   }
   set status = $ZF(-4,2,OutputLibID)   //unload "outputlibrary.dll"
   quit 0
 }
```

- The calls to **$ZF(-4,1)** load Callout libraries inputlibrary.dll and outputlibrary.dll into virtual memory and return system-defined library IDs for them.
- The calls to **$ZF(-4,3)** use the library IDs and function names to get IDs for the library functions. The returned function IDs are actually ZFEntry table sequence numbers (see "Creating a ZFEntry Table" in the previous chapter).
- The first loop uses **$ZF(-5)** to call a function from each library:
  - The **GetData()** function from inputlibrary.dll reads raw data from some unspecified source.
  - The **AnalyzeData()** function from outputlibrary.dll either normalizes the raw data or rejects it and returns an empty string.
  - Each normalized *datapoint* is stored in *flatdata(count)* (where the first call to Caché function $INCREMENT creates *count* and initializes it to `1`).

  By default, the loop fetches `100000` items. Since both libraries have been loaded and remain in memory, there is no processing overhead for switching between two different libraries.
- After the first loop ends, library inputlibrary.dll is no longer needed, so **$ZF(-4,2)** is called to unload it. Library outputlibrary.dll will remain in memory.

- The second loop processes each item from array *flatdata* and writes it to a file at some unspecified location:
  - Library function **PlotPoint()** reads the item and returns a comma-delimited string containing the coordinates at which it will be plotted (see "Introduction to Linkages" for a description of how multiple output parameters are returned by a library function).
  - The $PIECE function is used to extract coordinate values *x* and *y* from the string.
  - Library function **WriteData()** stores the item and coordinates in file outputfile.dat, which will be used by some other application to print a graph.
- After the second loop finishes, **$ZF(-4,2)** is called again to unload library outputlibrary.dll.

The following section describes the **$ZF(-6)** interface, which loads libraries into the same virtual memory space as the **$ZF(-5)** interface.

# 5.4 Using $ZF(-6) to Access Libraries by User Index

The **$ZF(-6)** function provides an efficient interface that allows access to Callout libraries through a globally defined index, usable even by applications that do not know the location of the shared library files. The user-defined index table stores a key/value pair consisting of a library ID number and a corresponding library filename. The filename associated with a given library ID can be changed when a library file is renamed or relocated. This change will be transparent to applications that load the library by index number. Other **$ZF** functions are provided to create and maintain index tables, and to unload libraries loaded by **$ZF(-6)**.

The following **$ZF** functions are discussed in this section:

- **$ZF(-6)** — invokes a function from a Callout library referenced by user-specified index number. Automatically loads the library if it is not already loaded.
- **$ZF(-4,4)** — unloads a Callout library specified by index number.
- **$ZF(-4,5)** and **$ZF(-4,6)** — creates or deletes an entry in the system index table. The system index is globally available to all processes within an instance of Caché.
- **$ZF(-4,7)** and **$ZF(-4,8)** — creates or deletes an entry in a process index table. Process tables are searched before the system table, so they can be used within a process to override system-wide definitions.

The **$ZF(-6)** interface is similar to the one used by **$ZF(-5)** (see "Using $ZF(-5) to Access Libraries by System ID") with the following differences:

- Before **$ZF(-6)** can be used, a library index table must be created. Library index values are user-defined, and can be changed or overridden at runtime.
- Library names are stored in the index, which does not have to be defined by the application that loads the library. The name and location of the library file can be changed in the index without affecting dependent applications that load the library by index value.
- There is no separate **$ZF** function to load a library. Instead, a library is loaded automatically by the first **$ZF(-6)** call that invokes one of its functions.
- It is assumed that the developer will already know the library function IDs (which are determined by their order in the ZFEntry table), so there is no **$ZF** function that will return a function ID for a given name and library index value.

The following examples demonstrate how the **$ZF(-6)** interface is used. The first example defines a library ID in the system index table, and the second example (which may be called from a different application) uses the library ID to invoke a library function:

### Defining a system index entry with $ZF(-4,5) and $ZF(-4,6)

This example sets `100` as the library ID for mylibrary.dll in the system index table. If a definition already exists for that number, it is deleted and replaced.

#### ObjectScript

```
set LibID = 100
set status = $ZF(-4,6,LibID)  // clear any old entries with this ID value
set status = $ZF(-4,5,LibID,"C:\calloutlibs\mylibrary.dll")  // set system ID
```

- *LibID* is the index number chosen by the developer. It can be any integer greater than zero, except reserved system values `1024` through `2047`.
- If the system index already contains an entry that uses index number `100`, the call to **$ZF(-4,6)** will delete it. This is good practice, since index entries cannot be overwritten.
- The call to **$ZF(-4,5)** associates index number `100` with library file mylibrary.dll.

Once the library ID is defined in the system index table, it is globally available to all processes within the current instance of Caché.

### Invoking a function with $ZF(-6)

This example uses the system index table created in the previous example. It uses **$ZF(-6)** to load the library and invoke a library function, then unloads the library. This code does not have to be called from the same application that defined the library ID in the system index:

#### ObjectScript

```
set LibID = 100   // library ID in system index table
set FuncID = 2    // second function in library ZFEntry table
set x = $ZF(-6,LibID, FuncID, "arg1")   // call function 2
set status = $ZF(-4,4,LibID)            // unload the library
```

- *LibID* is the library ID defined in the system index. This application does not have to know the library name or path in order to use library functions.
- *FuncID* is the function identifier for the second function listed in the ZFEntry table of library *LibID*. It is assumed that the developer has access to the library code — the **$ZF(-6)** interface does not have a function to retrieve this number by specifying the library function name.
- The call to **$ZF(-6)** specifies `100` as the library ID, `2` as the function ID and `"arg1"` as the argument passed to the function. This call will load Callout library mylibrary.dll if it isn't already loaded, and will invoke the second function listed in the ZFEntry table.
- The call to **$ZF(-4,4)** unloads the library. Each library loaded by **$ZF(-6)** will remain resident until the process ends or until unloaded by **$ZF(-4,4)**.

## 5.4.1 Using the $ZF(-6) Interface to Encapsulate Library Functions

It would be simple to write an example for the **$ZF(-6)** interface that works just like the example for the **$ZF(-5)** interface (see "Using $ZF(-5) to Access Libraries by System ID" earlier in this chapter), but this would not demonstrate the advantages of using **$ZF(-6)**. Instead, this section will present ObjectScript classes that allow an end user to perform exactly the same task without knowing anything about the contents or location of the Callout libraries.

The **$ZF(-5)** example invoked functions from Callout libraries inputlibrary.dll and outputlibrary.dll to process some experimental data and produce a two-dimensional array that could be used to draw a graph. The examples in this section perform the same tasks using the following ObjectScript code:

- Class User.SystemIndex — encapsulates the file names and index numbers used to define entries in the system index table.
- Class User.GraphData — provides methods that encapsulate functions from both libraries.
- Method **GetGraph()** — is part of an end user program that calls the User.GraphData methods. The code in this method performs exactly the same task as the **$ZF(-5)** example, but never calls a **$ZF** function directly.

The User.SystemIndex class allows applications that use the Callout libraries to create and access system index entries without hard coding index numbers or file locations:

### ObjectScript Class User.SystemIndex

#### Class Definition

```
Class User.SystemIndex Extends %Persistent
{
/// Defines system index table entries for the User.GraphData libraries
ClassMethod InitGraphData() As %Status
{
 // For each library, delete any existing system index entry and add a new one
    set sc = $ZF(-4,6,..#InputLibraryID)
    set sc = $ZF(-4,5,..#InputLibraryID,"c:\intersystems\cache\bin\inputlibrary.dll")
    set sc = $ZF(-4,6,..#OutputLibraryID)
    set sc = $ZF(-4,5,..#OutputLibraryID,"c:\intersystems\cache\bin\outputlibrary.dll")
    quit 0
}

Parameter InputLibraryID = 100;
Parameter OutputLibraryID = 200;
}
```

- The **InitGraphData()** method adds the libraries for User.GraphData to the system index table. It could be called automatically when the instance of Caché starts, making the libraries available to all processes within the instance.
- The *InputLibraryID* and *OutputLibraryID* class parameters are made available so that dependent applications don't have to hard code the index values (as demonstrated by the **Init()** method of User.GraphData in the following example).

The User.GraphData class allows end users to invoke library functions without knowing anything about the actual Callout libraries.

### ObjectScript Class User.GraphData

#### Class Definition

```
Class User.GraphData Extends %Persistent
{
/// Gets library IDs and updates the system index table for both libraries.
Method Init() As %Status
{
    set InLibID = ##class(User.GraphDataIndex).%GetParameter("InputLibraryID")
    set OutLibID = ##class(User.GraphDataIndex).%GetParameter("OutputLibraryID")
    quit ##class(User.SystemIndex).InitGraphData()
}
Property InLibID As %Integer [Private];
Property OutLibID As %Integer [Private];

/// Calls function "FormatData" in library "inputlibrary.dll"
Method FormatData(rawdata As %Double) As %String
{
    quit $ZF(-6,..InLibID,1,rawdata)
}
```

```
/// Calls function "RefineData" in library "outputlibrary.dll"
Method RefineData(midvalue As %String) As %String
{
    quit $ZF(-6,..OutLibID,1,midvalue)
}
/// Calls function "PlotGraph" in library "outputlibrary.dll"
Method PlotGraph(datapoint As %String, xvalue As %Integer) As %String
{
    quit $ZF(-6,..OutLibID,2,datapoint,xvalue)
}
/// Unloads both libraries
Method Unload() As %String
{
    set sc = $ZF(-4,4,..InLibID)   // unload "inputlibrary.dll"
    set sc = $ZF(-4,4,..OutLibID)   // unload "outputlibrary.dll"
    quit 0
}
}
```

- The **Init()** method calls a class method from User.SystemIndex that will set or update the system index entries for inputlibrary.dll and outputlibrary.dll. It also gets the current values for the library IDs. The developer of this class still needs to know something about the Callout library code, but future changes to the system index will be transparent.
- Methods **FormatData()**, **RefineData()**, and **PlotGraph()** each encapsulate a call to one library function. Since they contain only the unconditional **$ZF** function calls, the Caché compiler can optimize these methods to run just as fast as the original **$ZF** calls.
- The **Unload()** method unloads either or both libraries.

The following example demonstrates how an end user might use the methods in User.GraphData. The **GetGraph()** method uses the Callout libraries to perform exactly the same task as the **GraphSomeData()** method in the **$ZF(-5)** interface example (see "Using $ZF(-5) to Access Libraries by System ID" earlier in this chapter), but it does not directly call any **$ZF** functions:

### Method GetGraph()

```
Method GetGraph(loopsize As %Integer = 100000) As %Status
{
   // Get an instance of class GraphData and initialize the system index
   set graphlib = ##class(User.GraphData).%New()
   set sc = graphlib.Init()

   // call functions from both libraries repeatedly
   // each library is loaded automatically on first call
   for count=1:1:loopsize {
      set midvalue = graphlib.FormatData(^rawdata(count))
      set flatdata(count) = graphlib.RefineData(midvalue)
   }

   // plot results of the previous loop
   for count=1:1:loopsize {
      set x = graphlib.PlotGraph(flatdata(count),0)
      set y = graphlib.PlotGraph(flatdata(count),x)
      set ^graph(x,y) = flatdata(count)
   }

   //return after unloading all libraries loaded by $ZF(-6)
   set status = graphlib.Unload()
   quit 0
}
```

- The User.GraphData class is instantiated as *graphlib*, and the **Init()** method is called to initialize the system index. This method does not necessarily have to be called here, since the system index only has to be initialized once for all processes in an instance of Caché.
- The first loop indirectly uses **$ZF(-6)** to call a functions from each library, and **$ZF(-6)** automatically loads each library the first time it is needed. Library inputlibrary.dll is loaded by the first call to **FormatData()**, and outputlibrary.dll is loaded on the first call to **RefineData()**.
- The second loop invokes **PlotGraph()** from library outputlibrary.dll, which has already been loaded.

- The call to **Unload()** indirectly calls **$ZF(-4,4)** on both libraries.

## 5.4.2 Using a Process Index for Testing

As previously mentioned, a process index table is searched before the system index table, so it can be used within a process to override system-wide definitions. The following example creates a process index that is used to test a new version of one of the libraries used in the previous section.

**Using a process index to test a new version of "inputlibrary.dll"**

**ObjectScript**

```
// Initialize the system index and generate output from standard library
set testlib = ##class(User.GraphData).%New()
set sc = testlib.Init()
set sc = graphgen.GetGraph()   // get 100000 data items by default
merge testgraph1 = ^graph
kill ^graph

// create process index and test new library with same instance of testproc
set sc = $ZF(-4,4,100)   // unload current copy of inputlib
set sc = $ZF(-4,8)   // delete existing process index, if any
set sc = $ZF(-4,7,100, "c:\testfiles\newinputlibrary.dll")  // override system index
set sc = graphgen.GetGraph()
merge testgraph2 = ^graph

// Now compare testdata1 and testdata2
```

- In the first three lines, this test code initializes the system index and generates a graph, just like the previous example. The graph has been plotted using the standard version of inputlibrary.dll (identified by the system index entry with ID value `100`), and has been saved to *testgraph1*.
- The call to **$ZF(-4,4)** unloads inputlibrary.dll, which is identified by library ID `100` in the system index table.
- **$ZF(-4,8)** is called without specifying a library ID, indicating that all entries in the current process index table are to be deleted.
- The call to **$ZF(-4,7)** adds an entry to the process index table that sets `100` as the library ID for test library newinputlibrary.dll. This overrides the entry for that ID in the system index. Library ID `100` now points to newinputlibrary.dll rather than inputlibrary.dll.
- **GetGraph()** is called again, using the same instance of User.GraphData. Nothing has changed except that the standard version of inputlibrary.dll has been unloaded, so **GetGraph()** will now load and use the new version of the library. The test then compares graphs *testgraph1* and *testgraph2* to verify that both versions are producing the same results.

# 6

# Statically Linked Callout Functions

Previous chapters in this book have discussed how to create a stand-alone Callout library that can be loaded at runtime. However, it is also possible to statically link your Callout functions into a custom version of Caché so that they are always available. Since the functions are linked into Caché, there is no need to explicitly load a Callout library or to specify a library when invoking a function.

This chapter discusses the following topics:

- Invoking Statically Linked Callout Functions — describes the advantages of using statically linked functions.
- Creating a Custom CACHE.EXE File in Windows — describes using Visual Studio to link your Callout code into a custom version of Caché.
- Creating a Custom CACHE.EXE File in UNIX®, Linux, or OS-X — describes how to link your Callout code into a custom version of Caché.

## 6.1 Invoking Statically Linked Callout Functions

The calling convention for a statically linked Callout function is different than that of a dynamic library. For example:

```
$ZF(-3,"mylibrary","MYFUNC",args...)
```

would become:

```
$ZF("MYFUNC",args...)
```

Since your functions are statically linked, there is no need to specify a subfunction number and library name (such as `-3,"mylibrary"` in this example).

## 6.2 Creating a Custom CACHE.EXE File in Windows

It is assumed that you have a file named mycallouts.c that contains code identical to the code for a Callout library (see "Creating an InterSystems Callout Library") except that it does not contain a `#define ZF_DLL` directive. The steps to compile and link it are as follows:

### Compile or assemble the external functions

To compile mycallouts.c on Windows, use the command:

```
cl -c mycallouts.c
```

This produces an output file called mycallouts.obj that you can link to Caché.

### Link the object file and Caché object files into a new version of Caché

To link the C functions for use with **$ZF**:

1. Either perform your build in the <install-dir>/Dev/Cache/callin directory (where <*install-dir*> is the root directory for your configuration), or make a private directory of your own with a copy of the required files. You are linking together the following files:
   - cache.obj
   - main.obj
   - shdir.obj

   You link them with the file mycallouts.obj that you created by compiling mycallouts.c in your private directory.
2. Replace the file czf.obj with mycallouts.obj.
3. Build the project. This produces the file cache.exe in your private directory. If you keep the C functions in separate files, you need to compile them separately and declare them in mycallouts.c, but not include them there. If you get the message "Unresolved externals...," you may need to include other libraries. Call InterSystems Worldwide Response Center if you need help in determining which files to include.
4. Save the current installed version of cache.exe that is in <install-dir>/bin (where <*install-dir*> is the root directory for your configuration), so you can restore it.

### Shut down Caché and replace the Caché executable with the new file

To reinstall Caché for Windows:

1. Stop Caché using the **Stop Caché** choice from the cube menu.
2. Copy the cache.exe file from your private directory to <install-dir>/bin, where <*install-dir*> is the root directory for your configuration.

When you restart Caché, the functions are available.

### Restart Caché

On Windows, start Caché from the Caché Cube.

You now have a version of Caché that includes your C functions.

To return to the original version of Caché in the Caché manager's directory (on Windows):

1. Shut down Caché.
2. Copy the original saved cache.exe into <install-dir>/bin, where <*install-dir*> is the root directory for your configuration.

# 6.3 Creating a Custom CACHE.EXE File in UNIX®, Linux, or OS-X

It is assumed that you have a file named mycallouts.c that contains code identical to the code for a Callout library (see "Creating an InterSystems Callout Library") except that it does not contain a `#define ZF_DLL` directive. The steps to compile and link it are as follows:

### Compile or assemble the external functions

To compile mycallouts.c, use the command:

```
cc -c mycallouts.c
```

### Link the object file and Caché object files into a new version of Caché

To link the C functions for use with $ZF:

1. Either perform your build in the <install-dir>/dev/cache/callin/samples directory, or make a private directory of your own with a copy of the required files. You are linking together the following files:
   - cache.o (provided with the Caché distribution kit)
   - main.o (provided with the Caché distribution kit)
   - shdir.o (shdir.c must be compiled manually)
   - mycallouts.o (created by you in your private namespace)
2. Build the project. If you get the message "Unresolved externals," you may need to include other libraries; call InterSystems if you need help in determining which files to include. Note that if you place the C functions in separate files, you need to compile them separately and declare them in mycallouts.c, but not include them there.
3. Save the version of Caché that is in /cachesys/bin/ (or wherever your installation directory may be) so that you can restore it, if necessary.

### Shut down Caché and replace the Caché executable with the new file

To replace the Caché executable:

1. Shut down Caché by calling

   ```
   ccontrol stop <configname>
   ```

   where <*configname*> is the name of the Caché installation.
2. Copy the file cache from the private directory to <install-dir>/bin/, where <*install-dir*> is the root directory for your configuration.

### Restart Caché

Start up Caché with the cstart script or by calling:

```
ccontrol start <configname>
```

where <*configname*> is the installation of Caché to start.

To return to the original version of Caché in the Caché system manager's directory:

1. Shut down Caché.
2. Copy the original saved cache back into <install-dir>/bin, where <*install-dir*> is the root directory for your configuration.

# 7

# Special Considerations for UNIX® and Related Operating Systems

Some system calls may fail if the process receives a signal, the most common being: **open**, **read**, **write**, **close**, **ioctl**, and **pause**.

If the function uses any of these system calls, you have to code carefully to be able to distinguish among real errors, a `Ctrl-C`, and a call that should be restarted. A small set of functions was created to allow you to check for asynchronous events and to set a new alarm handler in **$ZF**.

## 7.1 Additional Call-Out Signal Processing

The function declarations are included in cdzf.h:

- `int sigrtclr();` — Clears retry flag. Should be called once before using **sigrtchk()**.
- `int dzfalarm();` — Establishes new SIGALRM handler.

  On entry to **$ZF**, the previous handler is automatically saved. On exit, it is restored automatically. A user program should not alter the handling of any other signal.
- `int sigrtchk();` — Checks for asynchronous events. Should be called whenever one of the following system calls fails: **open**, **close**, **read**, **write**, **ioctl**, **pause**, or any call that fails when the process receives a signal. It returns a code indicating the action that the user should take:
  - -1 — Not a signal. Check for I/O error. See contents of *errno* variable.
  - 0 — Other signal. Restart operation from point at which it was interrupted.
  - 1 — SIGINT/SIGTERM. Exit from $ZF with a SIGTERM "return 0". These signals are trapped appropriately.

A typical **$ZF** function used to control some device would use logic similar to the following pseudo-code:

```
if ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
    Set some flags
    Call zferror
    return 0;
}
```

The **open** system call may fail if the process receives a signal. Usually this situation is not an error and the operation should be restarted. Depending on the signal, however, you might take other actions. So, in order to take account of all the possibilities, consider using the following code:

```
sigrtclr();
while (TRUE) {
    if (sigrtchk() == 1) return 1 or 0;
    if ((fd = open(DEV_NAME, DEV_MODE)) < 0) {
        switch (sigrtchk()) {
             case -1:
                /* This is probably a real device error */
                Set some flags
                Call zferror
                return 0;
            case 0:
                /* A innocuous signal was received. Restart. */
                continue;
            case 1:
                /* Someone is trying to terminate the job. */
                Do cleanup work
                return 1 or 0;
        }
    }
    else break;
}
/*
Code to handle the normal situation:
open() system call succeeded
*/
```

Remember you must not set any signal handler except via **dzfalarm**.

# 8

# InterSystems Callout Quick Reference

The **$ZF()** function provides a set of subordinate functions identified by one or two numeric arguments (for example, the **$ZF(-100)** subordinate function runs an external program or system command, and the **$ZF(-4,1)** subordinate function loads a Callout library). The following list shows only the arguments that identify a specific **$ZF()** subordinate function. Most of these functions also take additional arguments, as described in the detailed entry for each function.

Detailed function descriptions are organized under the following headings:

- Using Statically Linked Functions
  - **$ZF()** (no subordinate function arguments) — invokes a Callout function that is statically linked to the current instance of Caché.
- $ZF(-100): Running Programs or System Commands
  - **$ZF(-100)** — executes a program or system command.
- $ZF(-3) and $ZF(-5): Accessing Libraries by Name
  - **$ZF(-3)** — loads a Callout library and invokes a library function.
  - **$ZF(-4,1)** — loads a Callout library specified by name, and returns an ID number for it.
  - **$ZF(-4,2)** — unloads a Callout library specified by ID number, or unloads all libraries.
  - **$ZF(-4,3)** — returns an ID number for a function in the specified library.
  - **$ZF(-5)** — invokes a function from a Callout library referenced by system-defined ID number.
- $ZF(-6): Accessing Libraries by User Index
  - **$ZF(-4,4)** — unloads a Callout library specified by index number.
  - **$ZF(-4,5)** — creates an entry in the Callout system index table
  - **$ZF(-4,6)** — deletes an entry in the Callout system index table
  - **$ZF(-4,7)** — creates an entry in the Callout process index table
  - **$ZF(-4,8)** — deletes an entry in the Callout process index table
  - **$ZF(-6)** — invokes a function from a Callout library referenced by user-specified index number.

# 8.1 Using Statically Linked Functions

When a function is statically linked with Caché (as described in "Statically Linked Callout Functions"), it is immediately available to **$ZF()** without the need to load a Callout library or specify a library name or identifier.

### $ZF()

Calls a statically linked function. No subordinate **$ZF** function calls are required, since there is no need to load and identify a separate Callout library.

```
retval = $ZF(func_name[, arg1[, ...argN]])
retval = $ZF(func_id[, arg1[, ...argN]])
```

*parameters:*

- *func_name* — The name of the library function as specified in the ZFEntry table (see "Creating a ZFEntry Table").
- *func_id* — sequence number of the library function within the ZFEntry table. If this number is known, it may be used instead of the function name for faster access (entries are numbered consecutively from 1).
- *args* — (optional) a comma-delimited list containing any arguments required by the library function.

*returns:*

- *retval* — the output value of the library function, or `NULL` if the library function does not set an output value.

*see also:*

See "Creating an InterSystems Callout Library" and "Statically Linked Callout Functions" for details and examples.

# 8.2 $ZF(-100): Running Programs or System Commands

The **$ZF(-100)** function is used to run an external program or system command, or to launch an operating system shell. This is the only **$ZF** function that can be used without a Callout library (see "Running Programs or System Commands with $ZF(-100)" for more information and examples).

### $ZF(-100)

Executes a program or an operating system command.

```
$ZF(-100, keyword_flags, program, arguments )
```

*parameters:*

- *keyword_flags* — (optional) A string expression consisting of a sequence of flags of the form `/keyword`. Keywords can be in upper or lowercase, and blanks are allowed between flags. I/O redirection keywords are followed by an operator and a path string (`/keyword=path` or `/keyword+=path`) as described below (see "Specifying Keywords").
- *program* — Specifies the program to be executed. It can be a full path or simply a name, in which case the usual operating system search path rules are followed.

- *arguments* — (optional) A comma-delimited list of program arguments. A variable number of parameters can be also be specified with *arg...* syntax (see "Variable Number of Parameters" in *Using Caché ObjectScript*).

*returns:*

One of the following status codes:

- `-1` — An operating system error occurred and the details are logged in SYSLOG.
- `0` — If `/ASYNCH` is specified, indicates that the program was successfully started.
- *status* — If `/ASYNCH` is not specified, *status* is the exit code (`0` or a positive number) returned by the program when it ends.

### Specifying Keywords

The following keywords control program execution and logging:

- `/SHELL` — Indicates that the program should be invoked within an operating system shell. The default is to not use a shell.
- `/ASYNC` — Indicates that the program should run asynchronously, allowing the **$ZF(-100)** call to return without waiting for it to complete.
- `/LOGCMD` — Causes the program command line to be logged in messages.log. This is a debugging tool that provides a way to view the arguments exactly as they are received by the program.

The following keywords and file specifiers control I/O redirection:

- `/STDIN=`*input-file*
- `/STDOUT=`*output-file* or `/STDOUT+=`*output-file*
- `/STDERR=`*error-file* or `/STDERR+=`*error-file*

I/O redirection keywords are followed by an operator ( `=` or `+=` ) and a filename or file path. Spaces are permitted around the operators. Standard input should point to an existing file. The standard output and standard error files are created if they don't exist and are truncated if they already exist. Use the = operator to create or truncate a file, or the += operator to append to an existing file. To make standard error and standard output to go to the same file, specify the same file for both keywords.

*see also:*

See "Running Programs or System Commands with $ZF(-100)" for more information and examples.

# 8.3 $ZF(-3) and $ZF(-5): Accessing Libraries by Name

The **$ZF(-3)** and **$ZF(-5)** functions allow an application to load InterSystems Callout shared libraries and invoke library functions at runtime. Library paths and library function names must be know by the calling application. **$ZF(-3)** specifies library and function names as arguments. **$ZF(-5)** specifies libraries and functions by system-defined ID numbers. Before **$ZF(-5)** can be used, the ID numbers must be obtained by calling utility functions (**$ZF(-4,1)** through **$ZF(-4,3)**) that take library and function names as arguments.

### $ZF(-3)

Loads a Callout library and executes a library function. Only one **$ZF(-3)** library may be loaded at a time. If a call to **$ZF(-3)** specifies a different library from the previous call, the previous library is unloaded and replaced.

```
retval = $ZF(-3, lib_name, func_name[, arg1[, ...argN]])
retval = $ZF(-3, lib_name, func_id[, arg1[, ...argN]])
```

*parameters:*

- *lib_name* — The name of the Callout library as specified in the ZFEntry table (see "Creating a ZFEntry Table"). If a library has already been loaded by a previous call to **$ZF(-3)**, an empty string (`""`) can be used to specify the current library.
- *func_name* — The name of the function to look up within the Callout library.
- *func_id* — sequence number of the library function within the ZFEntry table. If this number is known, it may be used instead of the function name for faster access (entries are numbered consecutively from 1).
- *args* — (optional) a comma-delimited list containing any arguments required by the library function.

*returns:*

- *retval* — the output value of the library function, or `NULL` if the library function does not set an output value.

*see also:*

See "Using $ZF(-3) for Simple Library Function Calls" for details and examples. See **$ZF(-4,3)** for another way to obtain the ZFEntry table sequence number.

### $ZF(-4, 1)

Utility function used with **$ZF(-5)**. Loads a Callout library specified by name, and returns an ID number for it.

```
lib_id = $ZF(-4,1, lib_name)
```

*parameter:*

- *lib_name* — The name of the Callout library to be loaded.

*returns:*

- *lib_id* — A system-defined identifier used to reference *lib_name*.

*see also:*

See "Using $ZF(-5) to Access Libraries by System ID" for details and examples.

### $ZF(-4, 2)

Utility function used with **$ZF(-5)**. Unloads a Callout library specified by ID number. If no ID is specified, it unloads all libraries in the process that were loaded by either **$ZF(-4,1)** or **$ZF(6)**. Does not unload the library loaded by **$ZF(-3)**.

```
$ZF(-4,2[,lib_id])
```

*parameter:*

- *lib_id* — The system-defined identifier returned by **$ZF(-4,1)**. If not specified, all libraries loaded by **$ZF(-4,1)** or **$ZF(6)** are unloaded.

*see also:*

See "Using $ZF(-5) to Access Libraries by System ID" for details and examples. Also see "Using a Process Index for Testing" for an example using **$ZF(-4,2)** without a library ID parameter

### $ZF(-4, 3)

Utility function used with **$ZF(-5)**. Returns an ID number for a function with the specified library ID and function name. This number is actually the sequence number of the function within ZFEntry table (see "Creating a ZFEntry Table").

```
func_id = $ZF(-4,3, lib_id, func_name)
```

*parameters:*

- *lib_id* — The system-defined library identifier returned by **$ZF(-4,1)**.
- *func_name* — The name of the function to look up within the Callout library.

*returns:*

- *func_id* — The returned ID number for the specified library function.

*see also:*

See "Using $ZF(-5) to Access Libraries by System ID" for details and examples.

**Note:** For **$ZF(-4, 4)** through **$ZF(-4, 8)**, see the next section ("Using Indexed Callout Libraries")

### $ZF(-5)

Calls a function from a Callout library referenced by system-defined ID number.

```
retval = $ZF(-5,lib_id,func_id,args)
```

*parameters:*

- *lib_id* — The Callout library ID number supplied by **$ZF(-4,1)**.
- *func_id* — The library function ID number supplied by **$ZF(-4,3)**.
- *args* — (optional) a comma-delimited list containing any arguments required by the library function.

*returns:*

- *retval* — The output value of the library function, or `NULL` if the library function does not set an output value.

*see also:*

See "Using $ZF(-5) to Access Libraries by System ID" for details and examples.

# 8.4 $ZF(-6): Accessing Libraries by User Index

The **$ZF(-6)** interface provides access to Callout libraries through a user-defined index table, usable even by applications that do not know the location of the shared library files. Utility functions **$ZF(-4, 4)** through **$ZF(-4, 8)** are used to create and maintain indexes.

**Note:** For information on **$ZF(-4, 1)**, **$ZF(-4, 2)**, and **$ZF(-4, 3)**, see the previous section ("$ZF(-3) and $ZF(-5): Accessing Libraries by Name")

### $ZF(-4, 4)

Utility function used with **$ZF(-6)**. Unloads a Callout library specified by index number.

```
$ZF(-4,4,lib_index)
```

*parameter:*

- *lib_index* — A user-specified Callout library index number (created by **$ZF(-4,5)** or **$ZF(-4,7)**)

*see also:*

See "Using $ZF(-6) to Access Libraries by User Index" for details and examples.

### $ZF(-4, 5)

Utility function used with **$ZF(-6)**. Creates an entry in the Callout system index table.

```
$ZF(-4,5,lib_index,lib_name)
```

*parameters:*

- *lib_index* — A unique user-specified number that will be used to reference the Callout library.
- *lib_name* — The name of the Callout library to be indexed.

*see also:*

Details and examples in "Using $ZF(-6) to Access Libraries by User Index".

### $ZF(-4, 6)

Utility function used with **$ZF(-6)**. Deletes an entry in the Callout system index table.

```
$ZF(-4,6,lib_index)
```

*parameters:*

- *lib_index* — An index number previously defined by a call to **$ZF(-4,5)**. This argument is required (unlike **$ZF(-4,8)**, where it can be omitted).

*see also:*

See "Using $ZF(-6) to Access Libraries by User Index" for details and examples.

### $ZF(-4,7)

Utility function used with **$ZF(-6)**. Creates an entry in the Callout process index table.

```
$ZF(-4,7,lib_index,lib_name)
```

*parameters:*

- *lib_index* — A unique user-specified number that will be used to reference the Callout library.
- *lib_name* — The name of the Callout library to be indexed.

*see also:*

See "Using $ZF(-6) to Access Libraries by User Index" for details and examples.

### $ZF(-4,8)

Utility function used with **$ZF(-6)**. Deletes an entry from the Callout process index table. If no index number is specified, all index entries are deleted.

```
$ZF(-4,8,lib_index)
```

*parameters:*

- *lib_index* — (optional) An index number previously defined by a call to **$ZF(-4,7)**. If not specified, all index entries are deleted.

*see also:*

See "Using $ZF(-6) to Access Libraries by User Index" for details and examples.

**Note:** For **$ZF(-5)**, see the previous section ("$ZF(-3) and $ZF(-5): Accessing Libraries by Name")

### $ZF(-6)

Look up and execute a function in an indexed Callout library.

```
retval = $ZF(-6,lib_index,func_id,args)
```

*parameters:*

- *lib_index* — A user-specified index to a Callout library (created by **$ZF(-4,5)** or **$ZF(-4,7)**)
- *func_id* — (optional) The ID number of the function within the Callout library, as supplied by **$ZF(-4,3)**. If omitted, call verifies the validity of *lib_index*, loads the library, and returns the full library filename.
- *args* — (optional) a comma-delimited list containing any arguments required by the library function.

*returns:*

- *retval* — The output value of the library function, or `NULL` if the library function does not set an output value.

*see also:*

See "Using $ZF(-6) to Access Libraries by User Index" for details and examples.