Progress®



Use Dynamic Call Objects in OpenEdge ABL Applications

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 11.3.0

Updated: 2019/03/01

Table of Contents

Preface....7
Purpose....7
Using ABL documentation....8
References to ABL compiler and run-time features....8
References to ABL data types....9
Typographical conventions....9
Examples of syntax descriptions....10
Long syntax descriptions split across lines....12
Complex syntax descriptions with both required and optional elements....12
OpenEdge messages....12
Obtaining more information about OpenEdge messages....13

Chapter 1: ABL elements related to the call object....15
Attributes of a call object handle....15
Methods of a call object handle....18

Chapter 2: When to use the call object....21

Chapter 3: Code examples....23
Running an internal procedure of a persistent external procedure....23
Running an internal procedure of a single-run or singleton procedure....24
Using a call object multiple times....25
Getting an attribute....26
Setting an attribute....26
Multiple dynamic invokes from a temp-table....27
Invoking a Windows DLL routine....28
Implementing a sleep timer....28

Preface

For details, see the following topics:

- Purpose
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This web paper describes how to:

- Dynamically execute a procedure
- Execute a user-defined function
- Get or set a handle attribute
- Invoke a handle method
- Invoke a Windows Dynamic Link Library (DLL) routine or invoke a UNIX shared library routine using the call object handle

---

Note: In the context of this document, *method* refers to a built-in, handle-based ABL method, such as `qryhandle:GET-NEXT( )`. You cannot run a class-based method using this feature. You can, however, dynamically invoke a class-based method using the `DYNAMIC-INVOKE` function or the `Invoke( )` method of the `Progress.Lang.Class` class. For more information, see *OpenEdge® Development: ABL Reference.*

This web paper includes the following sections:

- ABL elements related to the call object on page 15
- When to use the call object on page 21
- Code examples on page 23

Using ABL documentation

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is *ABL (Advanced Business Language)*. With few exceptions, all keywords of the language appear in all `UPPERCASE`, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates go to https://docs.progress.com.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the *ABL Virtual Machine (AVM)*. When the documentation refers to ABL source code compilation, it specifies *ABL* or *the compiler* as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies *the AVM* as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all `UPPERCASE`, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever *decimal* appears, this is a reference to the `DECIMAL` data type.
- Wherever *numeric* appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all `UPPERCASE`, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever *decimal* appears, this is a reference to the `DECIMAL` data type.
- Wherever *numeric* appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.

Convention	Description
UPPERCASE fixed width	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

In this example, ACCUM is a keyword, and *aggregate* and *expression* are variables:

Syntax

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

Syntax

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

Syntax

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

Syntax

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

Syntax

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

Syntax

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(*n*)], and any number of additional *expression* or SKIP [(*n*)] is allowed:

Syntax

```
MESSAGE { expression | SKIP [ ( n ) ] } ...
```

In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

Syntax

```
{ include-file
    [ argument | &argument-name = "argument-value" ] ... }
```

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, `WITH` is followed by six optional items:

Syntax

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, `ASSIGN` requires either one or more *`field`* entries or one *`record`*. Options available with *`field`* or *`record`* are grouped with braces and brackets:

Syntax

```
ASSIGN   { [ FRAME frame ] { field [ = expression ] }
           [ WHEN expression ] } ...
       | { record [ EXCEPT field ... ] }
```

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is `200`:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help** > **Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help** > **Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

ABL elements related to the call object

The ABL elements related to the call object consist of a statement, a handle, attributes, and methods.

Note: For more information on these ABL elements, see *ABL Reference*.

The `CREATE CALL` statement creates a call object, then stores a handle to it in the handle variable you specify.

A call object handle lets you do the following dynamically:

- Invoke an external procedure, internal procedure, or user-defined function
- Invoke a Windows DLL routine or UNIX shared library routine
- Get or set a handle attribute
- Run a handle method

For details, see the following topics:

- Attributes of a call object handle
- Methods of a call object handle

Attributes of a call object handle

The following table lists the attributes of the call object handle.

Table 1: Attributes of a call object handle

Attribute	Type	Description
ASYNCHRONOUS attribute	LOGICAL	Applies only when the logic to be dynamically invoked resides on a Progress AppServer™.Indicates if you want the logic to run asynchronously.
ASYNC-REQUEST-HANDLE attribute	HANDLE	Applies only when ASYNCHRONOUS is TRUE. A handle to an asynchronous-request object containing detailed information on your asynchronous request. Set by the INVOKE( ) method when ASYNCHRONOUS is TRUE.
CALL-NAME attribute	CHARACTER	The name of one of the following: • A procedure or user-defined function you want to invoke dynamically • An attribute you want to get or set dynamically • A method you want to invoke dynamically
CALL-TYPE attribute	INTEGER	Indicates if you are dynamically: • Invoking a procedure • Invoking a user-defined function • Invoking a Windows DLL or UNIX shared library routine • Getting an attribute or invoking a method • Setting an attribute
EVENT-PROCEDURE attribute	CHARACTER	Applies only when ASYNCHRONOUS is TRUE. The name of an internal procedure you want executed when a dynamic, asynchronous invoke completes. Note: You must set EVENT-PROCEDURE before you execute INVOKE( ).

Attribute	Type	Description
EVENT-PROCEDURE-CONTEXT attribute	HANDLE	Applies only when ASYNCHRONOUS is TRUE. A handle to a running persistent procedure containing an internal procedure you want executed when a dynamic, asynchronous invoke completes.
HANDLE attribute	HANDLE	A handle to the object.
IN-HANDLE attribute	HANDLE	Contains one of the following: • A handle to a persistent, single-run, or singleton procedure • A handle to an object, such as a buffer or socket, whose attributes or methods you want to invoke dynamically
INSTANTIATING-PROCEDURE attribute	HANDLE	Returns the handle to the procedure in which an object was instantiated.
IS-PARAMETER-SET attribute	LOGICAL	Indicates if you have already set a particular parameter.
LIBRARY attribute	CHARACTER	Specifies the name of a Windows DLL or a UNIX shared library.
LIBRARY-CALLING-CONVENTION attribute	CHARACTER	Specifies the calling convention for a Windows DLL or a UNIX shared library.
NUM-PARAMETERS attribute	INTEGER	The number of parameters you are passing.
ORDINAL attribute	INTEGER	Specifies the number of the entry point of the Windows DLL routine to invoke.
PERSISTENT attribute	LOGICAL	Indicates if you want an external procedure you plan to invoke dynamically to run as persistent.[1]
PROCEDURE-TYPE attribute	CHARACTER	Indicates if you want an external procedure you plan to invoke dynamically to run as persistent, single-run, or singleton. Takes values of "PERSISTENT", "SINGLE-RUN", or "SINGLETON", respectively.[1]

[1] See Running an internal procedure of a persistent external procedure on page 23 for details on how the PERSISTENT and PROCEDURE-TYPE attributes interact.

Attribute	Type	Description
RETURN-VALUE attribute	RETURN-VALUE-DATA-TYPE	Applies only to user-defined functions, attributes, and methods.Contains one of the following: • The value returned by a user-defined function • The value of an attribute • The value returned by a method
RETURN-VALUE-DATA-TYPE attribute	CHARACTER	The data type of the return value. Applies only to user-defined functions, attributes, and methods. If you set this before you execute the INVOKE( ) method, the return value is converted to this data type. If you do not set this before you execute the INVOKE( ) method, it indicates the return value's data type.
RETURN-VALUE-DLL-TYPE attribute	CHARACTER	Specifies the expected data type returned from a Windows DLL routine.
SERVER attribute	HANDLE	A handle to an AppServer containing logic you want to invoke dynamically.
TYPE attribute	CHARACTER	The type of a handle.

Methods of a call object handle

The following table lists the methods of a call object handle.

Table 2: Methods of a call object handle

Method	Return Type	Description
CLEAR( ) method	LOGICAL	Resets each attribute of the call object to its default value. **Note:** To reset just the parameters, set the NUM-PARAMETERS attribute to zero.

Method	Return Type	Description
`INVOKE( )` method	None (similar to the `RUN` statement)	Lets you dynamically do the following: • Invoke an external procedure, internal procedure, or user-defined function • Get or set an attribute • Invoke a method **Note:** `INVOKE( )` cannot occur within an expression.
`SET-PARAMETER( )` method	`LOGICAL`	Lets you set parameters for one of the following: • A procedure or user-defined function you want to invoke dynamically • An attribute you want to get or set dynamically • A method you want to invoke dynamically • A Windows DLL routine or a UNIX shared library routine you want to invoke dynamically.

2

When to use the call object

For some use cases, programming with the call object requires more lines of code than other ABL alternatives. The following table shows an example of the code required to invoke an external procedure dynamically and statically.

Table 3: Invoking hello.p dynamically and statically

Dynamic invoke	Static invoke

`DEFINE VARIABLE hCall AS HANDLE`
`  NO-UNDO.`
 	
`CREATE CALL hCall.`
 	
`/* Invoke hello.p non-persistently */`
`ASSIGN`
`  hCall:CALL-NAME      = "hello.p"`
`  hCall:NUM-PARAMETERS = 1.`
 	
`hCall:SET-PARAMETER (1, "CHARACTER",`
`  "INPUT", "Hello world").`
 	
`hCall:INVOKE( ).`
 	
`/* Clean up */`
`DELETE OBJECT hCall.`
	`RUN hello.p ("Hello world").`

As the previous table illustrates, executing `hello.p` dynamically using the call object requires many more lines of code and is therefore less efficient than doing it with static invoke.

Consider using the call object for these situations:

- To invoke an internal or external procedure whose *calling sequence* (number of parameters and the data type of each) is unknown at compile time.

  Note: If the only the **name** of the procedure is unknown at compile time, use the `RUN` statement with the `VALUE` option, and avoid using the call object.

- To invoke a function whose calling sequence is unknown at compile time

  Note: If the only the **name** of the function is unknown at compile time, use the `DYNAMIC-FUNCTION( )` function, and avoid using the call object.

- To reference a widget attribute or method whose name is unknown at compile time.

  If you already know the name of the attribute or procedure, then you also know its syntax, since the name implies certain syntax. If you know the syntax, then you know the calling sequence, since the syntax defines the calling sequence. If you know the calling sequence, you can use widget:attribute or widget:method syntax, and avoid using the call object.

- To dynamically invoke a Windows DLL routine or a UNIX shared library routine when the following is true:
  - The number of parameters and their data type is only known at run time
  - The routine exists in both a Windows DLL and a UNIX shared library
  - The routine has a variable number of parameters

3

Code examples

This section contains code examples showing some ways in which the call object can be used.

For details, see the following topics:

- Running an internal procedure of a persistent external procedure
- Running an internal procedure of a single-run or singleton procedure
- Using a call object multiple times
- Getting an attribute
- Setting an attribute
- Multiple dynamic invokes from a temp-table
- Invoking a Windows DLL routine
- Implementing a sleep timer

Running an internal procedure of a persistent external procedure

The following example, which illustrates using a single call object multiple times, is a bit more complicated than that shown in Attributes of a call object handle on page 15. This example also shows how to:

- Dynamically invoke the external procedure `persis.p` persistently

- Dynamically invoke an internal procedure of `persis.p`, `internal-persis-proc`, with an input parameter of type "`INTEGER`" and with the value 333

```
DEFINE VARIABLE hCall AS HANDLE NO-UNDO.

CREATE CALL hCall.

/* Invoke persis.p persistently */
ASSIGN
  hCall:CALL-NAME  = "persis.p"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE  = PROCEDURE-CALL-TYPE
  hCall:PERSISTENT = TRUE.

hCall:INVOKE.

/* Invoke internal-persis-proc in persis.p */
ASSIGN
  hCall:CALL-NAME      = "internal-persis-proc"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "INTEGER", "INPUT", 333).
hCall:INVOKE( ).

/* Clean up */
DELETE PROCEDURE hCall:IN-HANDLE.
DELETE OBJECT hCall.
```

When persis.p is invoked dynamically, the handle of the running persistent procedure is stored automatically in the call object's `IN-HANDLE` attribute. When `internal-persis-proc` is invoked dynamically, ABL knows it resides in the running persistent procedure whose handle is stored in the call object's `IN-HANDLE` attribute.

You can also run a procedure persistently by setting the `PROCEDURE-TYPE` attribute to "`PERSISTENT`". Setting the `PROCEDURE-TYPE` attribute to "`PERSISTENT`" and setting the `PERSISTENT` attribute to "`TRUE`" are equivalent, and setting one will automatically set the other. Setting the two attributes to conflicting values, e.g., `PROCEDURE-TYPE` to "`SINGLETON`" and `PERSISTENT` to `TRUE`, will cause a run-time error.

Running an internal procedure of a single-run or singleton procedure

The following example shows how to dynamically invoke a procedure as single-run and then dynamically invoke an internal procedure of the single-run. The same syntax can be used to call a procedure as persistent or singleton by setting `PROCEDURE-TYPE` to "`PERSISTENT`" or "`SINGLETON`", respectively.

```
DEFINE VARIABLE hCall AS HANDLE NO-UNDO.

CREATE CALL hCall.

/* Invoke single.p as single-run*/
ASSIGN
  hCall:CALL-NAME  = "single.p"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE  = PROCEDURE-CALL-TYPE
  hCall:PROCEDURE-TYPE = SINGLE-RUN.
```

```
hCall:INVOKE.

/* Invoke internal-single-proc in single.p */

ASSIGN
  hCall:CALL-NAME      = "internal-single-proc"
  /* Sets CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "INTEGER", "INPUT", 333).
hCall:INVOKE( ).

/* Clean up */
DELETE PROCEDURE hCall:IN-HANDLE.
DELETE OBJECT hCall.
```

Using a call object multiple times

This example further demonstrates using a call object multiple times by:

1. Dynamically invoking the procedure file `hello.p` with the "`CHARACTER`" input parameter, string "`HELLO WORLD`"
2. Dynamically invoking the external procedure `persis.p` persistently
3. Dynamically invoking an internal procedure of `persis.p`, `internal-persis-proc`, with an input parameter of type "`INTEGER`" and with the value 333

```
DEFINE VARIABLE hCall AS HANDLE NO-UNDO.

 CREATE CALL hCall.

/* Invoke hello.p nonpersistently */
ASSIGN
  hCall:CALL-NAME      = "hello.p"
  /* Set CALL-TYPE to the default */
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "CHARACTER", "INPUT", "HELLO WORLD").
hCall:INVOKE( ).

/* Reset the call object handle */
hCall:CLEAR( ).

/* Invoke persis.p persistently */
ASSIGN
  hCall:CALL-NAME  = "persis.p"
  /* Set CALL-TYPE to the default */
  hCall:CALL-TYPE  = PROCEDURE-CALL-TYPE
  hCall:PERSISTENT = TRUE.

hCall:INVOKE.

/* Invoke internal-persis-proc in persis.p */
ASSIGN
  hCall:CALL-NAME      = "internal-persis-proc"
  /* Set CALL-TYPE to the default */
```

```
  hCall:CALL-TYPE      = PROCEDURE-CALL-TYPE
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "INTEGER", "INPUT", 333).
hCall:INVOKE( ).

/* Clean up */
DELETE PROCEDURE hCall:IN-HANDLE.
DELETE OBJECT hCall.
```

This example resets the call object handle by using the `CLEAR( )` method between invoking `hello.p` (the first invoke) and invoking `persis.p` (the second invoke).

Getting an attribute

This example gets the current value of the `TITLE` attribute of a frame. The call object's `CALL-TYPE` attribute is set to `GET-ATTR-CALL-TYPE`. The value of the `TITLE` attribute is returned through the call object's `RETURN-VALUE` attribute, as shown:

```
/* Get title of frame */

ASSIGN
  hCall:IN-HANDLE = myframe_handle
  hCaLL:CALL-TYPE = GET-ATTR-CALL-TYPE
  hCall:CALL-NAME = "TITLE".

hCall:INVOKE( ).
Mytitle = hCall:RETURN-VALUE.
```

Setting an attribute

This example sets the `SESSION` handle's `NUMERIC-FORMAT` attribute to "european":

```
/* Set SESSION:NUMERIC-FORMAT to "european" */

ASSIGN
  hCall:IN-HANDLE      = "session"
  hCall:CALL-TYPE      = SET-ATTR-CALL-TYPE
  hCall:CALL-NAME      = "numeric-format"
  hCall:NUM-PARAMETERS = 1.

hCall:SET-PARAMETER(1, "CHARACTER", "INPUT", "european").
hCall:INVOKE( ).
```

In contrast to the Getting an attribute on page 26 example, this "Setting an attribute" example:

- Sets the `CALL-TYPE` attribute to `SET-ATTR-CALL-TYPE`
- Sets the `NUM-PARAMETERS` attribute to 1

- Runs the SET-PARAMETER( ) method to make the first parameter an INPUT parameter of data type CHARACTER with the value "european"

Multiple dynamic invokes from a temp-table

This example shows how a series of dynamic invokes of the call object can be set up using a temp-table, ttParam, whose handle is passed into the procedure:

```
/* ttParam is a temp-table that has one record with these fields:
   parm_1
   parm_2
   ...
   parm_n
   run-name
   nparms
   datatypes, extent nparms
   iomodes, extent nparms
*/

DEFINE INPUT PARAMETER TABLE-HANDLE ttParam NO-UNDO.

DEFINE VARIABLE hCall    AS HANDLE  NO-UNDO.
DEFINE VARIABLE hDtypes  AS HANDLE  NO-UNDO.
DEFINE VARIABLE hIOmodes AS HANDLE  NO-UNDO.
DEFINE VARIABLE ix       AS INTEGER NO-UNDO.

ASSIGN
  hDtypes  = ttParam:BUFFER-FIELD("datatypes")
  hIOmodes = ttParam:BUFFER-FIELD("iOmodes").

ttParam:FIND-FIRST.

CREATE CALL hCall.
ASSIGN
  hCall:CALL-NAME      = ttParam:BUFFER-FIELD("run-name"):BUFFER-VALUE
  hCall:NUM-PARAMETERS = ttParam:BUFFER-FIELD("nparms"):BUFFER-VALUE.

FOR ix = 1 TO hCall:NUM-PARAMETERS:
  hCall:SET-PARAMETER(ix, hDtypes:BUFFER-VALUE(ix),
    hIOmodes:BUFFER-VALUE(ix), ttParam:BUFFER-FIELD(ix):BUFFER-VALUE).
END.

hCall:INVOKE( ).
DELETE OBJECT hCall.
```

To implement this example, you must:

1. Define a temp-table record structure where each field represents one data item involved in dynamic invoke
2. Load the temp-table with a row of data for each dynamic invoke to be performed
3. Read the temp-table and performing a dynamic invoke for every record

Setting up a temp-table in this way allows an application to perform any number of dynamic invokes on the fly.

Invoking a Windows DLL routine

The following example demonstrates the use of the RETURN-VALUE-DLL-TYPE attribute and the RETURN-VALUE attribute when invoking a Windows DLL routine. RETURN-VALUE-DLL-TYPE is set to "LONG", which is the value that the DLL routine expects to receive. For example:

```
FUNCTION GetWinVersion RETURNS INTEGER:
  DEFINE VARIABLE cValue  AS CHARACTER NO-UNDO.
  DEFINE VARIABLE libName AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hCall   AS HANDLE    NO-UNDO.

  CREATE CALL hCall.
  ASSIGN
    hCall:CALL-NAME             = "GetVersion"
    hCall:LIBRARY               = "kernel32.dll"
    hCall:CALL-TYPE             = DLL-CALL-TYPE
    hCall:RETURN-VALUE-DLL-TYPE = "LONG".

  hCall:INVOKE( ).
  cValue = hCall:RETURN-VALUE.

  DELETE OBJECT hCall.
  RETURN cValue.
END FUNCTION.
```

After the invoke, RETURN-VALUE contains the "INTEGER" data type.

Implementing a sleep timer

The following example implements the user-defined function sleep, which causes the AVM to sleep for a specified number of milliseconds:

```
FUNCTION sleep RETURNS INTEGER (msecs AS INTEGER):
  DEFINE VARIABLE cFunction AS CHARACTER NO-UNDO INITIAL "sleep".
  DEFINE VARIABLE cLibrary  AS CHARACTER NO-UNDO INITIAL "libc.so.1".
  DEFINE VARIABLE hCall     AS HANDLE    NO-UNDO.

  CREATE CALL hCall.
  ASSIGN
    cLibrary             = "kernel32.dll" WHEN OPSYS = "WIN32"
    cFunction            = "Sleep" WHEN OPSYS = "WIN32"
    hCall:CALL-NAME      = cFunction
    hCall:LIBRARY        = cLibrary
    hCall:CALL-TYPE      = DLL-CALL-TYPE
    hCall:NUM-PARAMETERS = 1.

  hCall:SET-PARAMETER(1, "LONG", "INPUT", msecs).
  hCall:INVOKE( ).

  DELETE OBJECT hCall.
  RETURN msecs.
END FUNCTION.
```

Note that the code determines on which OS it is running, and invokes the appropriate Windows DLL or UNIX shared library.