HP TestExec SL

Reference

Notice

The information contained in this document is subject to change without notice. Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP makes no warranties of any kind with regard to this document, whether express or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19 (c) (1,2).

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

Printing History

E1074-90000 — Software Rev. 1.00 — First printing - August, 1995

E1074-90005 — Software Rev. 1.50 — Rev. A - March, 1996

Note The documentation expanded into a multi-volume set of books at Rev. B.

E1074-90007 — Software Rev. 1.51 — Rev. B - June, 1996

E2011-90011 — Software Rev. 2.00 — Rev. C - January, 1997

E2011-90014 — Software Rev. 2.10 — Rev. D - May, 1997

E2011-90018 — Software Rev. 3.00 — Rev. E - January, 1998

About This Manual

This manual provides reference descriptions of the syntax for the APIs and functions provided with HP TestExec SL.

Conventions Used in this Manual

Vertical bars denote a hierarchy of menus and commands, such as:

View | Listing | Actions

Here, you are being told to choose the Actions command that appears when you expand the Listing command in the View menu.

If a form uses tabs to organize its contents, the name of a tab may appear in the hierarchy of menus and commands. For example, the Options dialog box has a tab named Search Paths. A reference to that tab looks like this:

View | Options | Search Paths

To make the names of functions stand out in text yet be concise, the names typically are followed by "empty" parentheses—i.e., `MyFunction()`—that do not show the function's parameters.

Some programming examples use the C++ convention for comments, which is to begin commented lines with two slash characters, like this:

```
// This is a comment
```

C++ compilers also will accept the C convention of:

```
/* This is a comment */
```

The C++ convention is used here simply because it results in shorter line lengths, which make examples fit better on a printed page. If you are using a C-only compiler, be sure to follow the C convention.

Contents

Contents

Contents

Contents

1

The HP TestExec SL APIs & Functions

This chapter provides an introduction to the APIs and functions that are a part of HP TestExec SL. These APIs and functions, many of which are used in action routines, are the primary mechanism that user-written code has for interfacting with HP TestExec SL.

Overview of the APIs & Functions

HP TestExec SL's APIs and functions include the following:

- C Action Development API

  The C Action Development API provides functions that let you use a C/C++ compiler to develop action routines, such as functions for:

  - Manipulating data stored by name in data containers that reside in parameter blocks
  - Locating data in parameter blocks
  - Manipulating data stored in data containers that do not reside in parameter blocks
  - Copying and releasing data in data containers
  - Manipulating switching paths from actions
  - Waiting via timers
  - Interacting with arrays
  - Sending and receiving user-defined messages that provide a means of communicating across processes

  For detailed descriptions, see Chapter 2.

- Hardware Handler API & Functions

  Hardware handlers contain user-written functions that control hardware, such as switching modules. Some of those functions include calls to functions that are members of the Hardware Handler API, which lets you define and programmatically interact with hardware.

  For detailed descriptions, see Chapter 3.

- Exception Handling API

  The Exception Handling API provides functions that let you raise and examine exceptions that occur during testing. Also, it includes functions that let you programmatically abort testing.

  For detailed descriptions, see Chapter 4.

- Runtime API

  The Runtime API lets you replace the default user interface for operators with a custom interface for operators or automation. It provides functions for:

  - Registering a personality
  - Controlling the state of the Test Executive
  - Miscellaneous server requests, such as interacting with operator interfaces
  - Callback registration
  - Interacting with system data
  - Controlling datalogging
  - Supporting functionality in operator interfaces created with Visual Basic

  For detailed descriptions, see Chapter 5.

Note

Although the sheer number of APIs and functions may seem intimidating at first, you do not need to understand all of them to use HP TestExec SL. Instead, you can learn about a subset of functions in an API that does specific tasks. For example, suppose you needed to control switching hardware directly from an action routine. Rather than study the entire C Action Development API, you might refer to only the group of functions used to control switching paths from actions.

Interpreting the Syntax of API Functions

Why Do So Many Names Have “Uta” in Them?

API functions or data types with “Uta” or “UTA” in their names are based on Hewlett-Packard’s TestCore services, which provide an open, standardized framework for creating or modifying test systems. HP TestCore provides the underlying concepts upon which HP TestExec SL is built. Also, the characters “UTA” help to eliminate naming conflicts that could occur with other libraries you may be using.

How Do I Interpret the Syntax?

The names of API functions based on HP TestCore are intended to be readily distinguishable from other services you may be using. They are relatively simple to read if you understand the rules or “grammar” used in declaring and calling them.

Suppose we dissect this example:

```
UtaInt32 UtaInt32GetValue(HUTAINT32 hData);
```

Examining the components from left to right shows that:

- The return data type, **UtaInt32**, can be further divided into:

  Uta *and* **Int32**

  where

  Uta identifies this as a portable data type supported by HP TestCore instead of a particular data type associated with a programming language. The use of HP TestCore data types, which are data containers, allows data portability across various programming languages and environments. Most of the HP TestCore data types are easy to interpret because they contain familiar words like “String”, “Array”, or “Real”.

  Int32 identifies the HP TestCore data type as a 32-bit integer.

- The name of the API function, **UtaInt32GetValue**, can be further divided into:

  Uta *and* **Int32** *and* **GetValue**

  where

  The **Uta** prefix indicates the function is based on HP TestCore.

  Int32 identifies as a 32-bit integer the data type upon which the function operates.

  GetValue is a phrase that describes what happens when the function is called. “Get” indicates the function fetches or retrieves something, which in this case is “Value”. Other common phrases you will see in the names of API functions include descriptive words like “Set”, “Create”, “Find”, “Clear”, and “Update”.

- The data type of the parameter passed into the function, **HUTAINT32**, can be further divided into:

  HUTA *and* **INT32**

  where

  HUTA identifies this as a handle (H) to an HP TestCore data type.

  INT32 identifies this HP TestCore data type as a 32-bit integer.

  Among the other HP TestCore data types you will see are HUTAREAL64, HUTASTRING, and HUTAI32ARR. Usually, these are reasonably easy to interpret. For example, HUTAI32ARR is the handle (H) to an HP TestCore data type (UTA) that is an array (ARR) of 32-bit integers (I32). In many cases, reading the description of what a function does provides clues to what data types it must use.

- **hData** is the name of the parameter that is passed into the function. It is a handle to the data container whose 32-bit integer value this API function returns.

Taken collectively, you might read this function as “Retrieve a 32-bit integer value from the data container whose handle is specified, and return that value as a 32-bit integer.”

In a similar fashion, the names of other HP TestCore APIs are constructed from simple components whose meanings become more apparent as you work with them.

A Note About the UTAAPI & UTADLL Macros

Some of the function calls that appear in this book or in the sample files included with HP TestExec SL use macros named UTADLL or UTAAPI. For example, a call to the `Init()` function used in hardware handlers includes the UTADLL macro, like this:

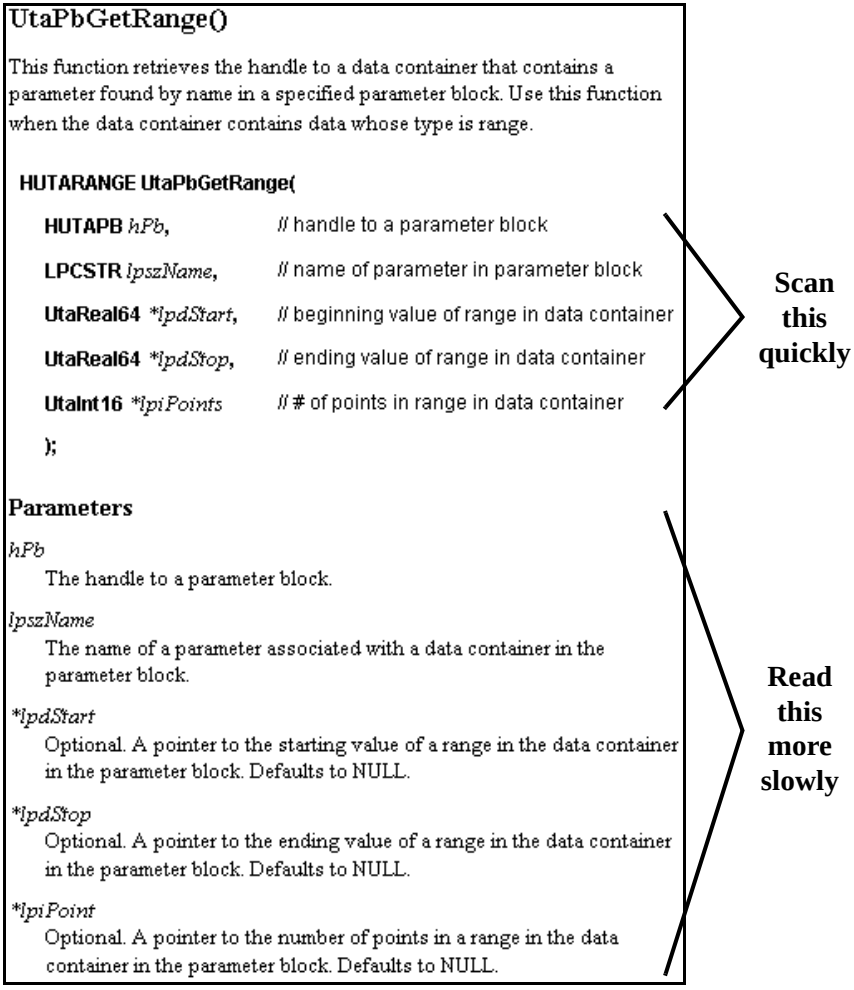
```
LPVOID UTADLL Init (HUTASWMOD hModule, HUTAPB hParameterBlock)
```

These macros enhance code portability across operating platforms. If a function call requires one of these macros, as shown in its syntax description or example, use UTAAPI when declaring the function’s prototype in a header file (“.h”) and use UTADLL when calling the function in an implementation file (“.c” or “.cpp”).

Browsing the Syntax Descriptions

The syntax descriptions in this book have the parameters associated with functions described twice. As shown below, the first description is for more experienced users of HP TestExec SL who simply need to jog their

memories about the order or data type of parameters, and the second is for beginners or those who need to know the details of a specific parameter.



UtaPbGetRange()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains data whose type is range.

```
HUTARANGE UtaPbGetRange(
    HUTAPB hPb,              // handle to a parameter block
    LPCSTR lpszName,         // name of parameter in parameter block
    UtaReal64 *lpdStart,     // beginning value of range in data container
    UtaReal64 *lpdStop,      // ending value of range in data container
    UtaInt16 *lpiPoints      // # of points in range in data container
    );
```

Parameters

hPb
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

**lpdStart*
Optional. A pointer to the starting value of a range in the data container in the parameter block. Defaults to NULL.

**lpdStop*
Optional. A pointer to the ending value of a range in the data container in the parameter block. Defaults to NULL.

**lpiPoint*
Optional. A pointer to the number of points in a range in the data container in the parameter block. Defaults to NULL.

Scan this quickly

Read this more slowly

How HP TestCore API Functions Use Data Containers

What is a Data Container?

The concept of a “data container” is key to understanding and using API functions based on HP TestCore. As the name implies, a data container is something that contains data. The container is an object and the data within it is either a common data type—an integer or string, perhaps—or a custom data type created for some specific purpose, such as data that stores a switching state.

Suppose we draw an analogy between a data container and its data and a bar of candy and its wrapper, where the wrapper is the data container and the candy bar is the data. In both cases, you can see the container’s exterior but you cannot necessarily view its contents. For example, you can see neither the ingredients used to make the candy bar nor the structure of the data inside the data container. In computer science parlance, this is an example of “data hiding.”

Similar to the way in which a maker of candy produces a candy bar and then puts a wrapper around it, there are functions for creating a data container and the data inside it. For example, `UtaInt32Create()` creates a data container that contains a 32-bit integer value. In most cases, the functions let you specify an initial value for the data when creating a data container.

As with a candy bar in its wrapper, you cannot immediately access the data in a data container. Before you can eat the candy bar, you must remove the candy from its wrapper. In the case of a data container, you use a function to access or manipulate the data it contains. For example, `UtaInt32GetValue()` returns the value of a 32-bit integer stored inside a data container created using `UtaInt32Create()`.

How Do Handles Work?

Accessing the contents of a candy bar is as simple as grasping the bar and peeling back its wrapper. The method for grasping a data container is

through its "handle," which is a unique integer associated with a data container when it is created.

When you call the `UtaInt32Create()` function mentioned earlier, it returns a handle to the newly created data container. For example,

```
HUTAINT32 hMyData;
long lValue = 5;
// create the data container
hMyData = UtaInt32Create(lValue);
```

creates and returns the handle to an HP TestCore data container that contains a 32-bit integer whose value is 5.

Note

Although there are functions such as `UtaInt32Create()` that let you programmatically create data containers, in most cases you will be using API functions to manipulate data in existing parameter blocks created with HP TestExec SL's Action Definition Editor.

A call to `UtaInt32GetValue()` to retrieve the value from the data container might look like this:

```
long lMyInt;
// put the value in lMyInt
lMyInt = UtaInt32GetValue(hMyData);
```

Notice that you are not accessing the data directly, as you would with a simple data type. Instead, you are using a function to access it via its handle, which references its container.

Suppose you wished to change the value of the data inside a container. Continuing with the example, you might do this:

```
Value = 10;
// update the data in container
UtaInt32SetValue(hMyData, lValue);
```

When finished with the data container, you could free the memory it was using, like this:

```
UtaDataRelease((HUTADATA)hMyData);
```

Notice the use of the cast operator to cast the data type being released to HP TestCore's generic data type, HUTADATA. This lets you use `UtaDataRelease()` to free memory for all types of data containers.

The APIs based on HP TestCore are rich with functions that let you manipulate data containers and their contents. For example, you can copy data from one container to another, work with far more complex types of data than shown in these simple examples, and more.

Why Should I Use APIs & Data Containers?

HP TestExec SL's use of data containers provides several benefits over using the common data types found in programming languages:

- Speed

  Because HP TestCore data types make extensive use of handles, they let you manipulate data via addresses in memory (fast) instead of copying data from one place to another (slow).

- Consistency

  The use of handles and data containers is consistent across the HP TestCore data types and across the platforms upon which HP TestExec SL runs. This means that once you understand handles and data containers, you have access to a broad range of features that work alike in various environments, languages, and compilers.

- Data hiding

  APIs and data containers hide the details of their contents. This makes it easy for you to work with rich data types like ranges and switching states, which do not exist as standard data types in programming languages, without knowing or caring about their internal structures. For example, instead of being concerned with what is stored in a range or how it is stored, you simply create a data container that contains range data and use predefined functions to access it.

  Data hiding also makes it possible for Hewlett-Packard to enhance HP TestCore's functionality and performance without changing the

external appearance of data or the means by which that data is accessed. This means that code you write today remains compatible with future versions of HP TestExec SL.

- Cross-platform compatibility

  Having data types defined by HP TestCore makes your code portable across platforms. Suppose that instead of using HP TestCore data types, you used standard data types provided by a programming language. As you moved from compiler to compiler or platform to platform, the characteristics of your data could change. For example, an integer might be a 16-bit data type on one platform and a 32-bit data type on another. Code developed for one platform or language might use a different calling convention—i.e., the order in which parameters passed to functions are pushed onto the stack—than code developed elsewhere. Or, differences across platforms could cause problems when saving or retrieving data in files because the representation of data changed with the platform.

  Instead of requiring you to deal with these portability issues, HP TestCore data types mean that the designers of HP TestExec SL deal with them. Typically, you can move action routines from one platform to another, recompile them into a new DLL or shared library there, and reuse them as-is.

How Do Parameter Blocks Use Data Containers?

Thus far, we have discussed using handles to access individual data items in their containers. However, HP TestExec SL also supports “parameter blocks,” which are groups of named parameters in which each parameter has a value. Each action has a parameter block associated with it. Given that a parameter block is a collection of data containers, how do you manipulate the data inside a parameter block?

HP TestCore API functions that manipulate parameter blocks include “Pb” in their names. For example `UtaPbSetInt32()` sets the value of a named 32-bit integer in a parameter block. As with many of the other functions, this one uses a handle, except this time it is the handle to a parameter block

instead of the handle to an individual data container. A call to the function might look like this:

```
long lMyInt = 5;
char MyParameter[] = “Voltage”;
UtaPbSetInt32(hMyParameterBlock, MyParameter, lMyInt);
```

where

hMyParameterBlock is the handle to the parameter block of interest

MyParameter is the name of a parameter inside the parameter block

As shown in this example, setting the value of data in parameter blocks is straightforward. Retrieving data from parameter blocks offers more options. In many cases, you can either retrieve a handle to a data container (parameter) in a parameter block and use it to access the data, or you can directly access the data via a pointer.

Suppose a parameter block named *MyParmBlock* exists, its handle is *hMyParmBlock*, and the parameter block contains a parameter named *MyParm* associated with a data container that contains a 32-bit integer number. An example of using a handle to access the data might look like this:

```
// assign variable for handle to data type
HUTAINT32 hMyData;
// assign variable to hold returned value
long lMyInt32;

// get handle to container that contains a
// 32-bit integer number
hMyData = UtaPbGetInt32(hMyParmBlock, "MyParm");

// get value from data container
lMyInt32 = UtaInt32GetValue(hMyData);
```

Function `UtaInt32GetValue()` has an optional parameter in which you can pass a pointer to the data in the data container. Accessing the data via this method might look like this:

```
// assign variable for handle to UTA data type
HUTAINT32 hMyData;
long lMyInt32; // variable to hold returned value

hMyData = UtaPbGetInt32(
   hMyParmBlock,
   "MyParm",
   &lMyInt32
   );
// lMyInt32 now contains 32-bit integer value
```

For simple data types, such as integers, reals, and strings, it usually is simplest to use a pointer to return the value. For more complicated data types, such as complex, and range data, returning the handle can be advantageous because those data types have many API functions that let you manipulate them via handles.

Which Data Types Does the HP TestCore API Support?

Overview of the Data Types

Many of the data types supported by HP TestExec SL are familiar types, such as integers, reals, strings, and arrays. However, some of the data types—for example, ranges and states—are special, "test-oriented" data types. The next several topics describe these potentially unfamiliar data types.

Data Types Associated with Switching

A switching state describes a collection of switching elements and their positions. For example, suppose the switching elements—relays and such—in your switching hardware were set up to make the necessary connections prior to making a measurement. If you could take a "snapshot" of that set of connections, you could think of it as a switching state.

A switching state is created by merging one or more switching paths, which are higher-level abstractions that describe a series of connections, into a switching state. For example, the switching state needed to make a measurement might contain several paths. One path might connect a power supply to the UUT, another might connect the UUT's output to a frequency counter, etc. Collectively, these paths define the switching state.

The word "State" appears in the names of HP TestCore API functions used to create and manipulate switching states. For example, function `UtaStateCreate()` creates a data container that contains a switching state. Similarly, the word "Path" appears in the names of functions used to manipulate switching paths. For example, function `UtaPathConnect()` sets up a switching path.

For more information about switching, see "Using C Actions to Control Switching Paths" in Chapter 3 of the *Using HP TestExec SL* book.

Complex Data

A complex number describes a vector quantity. It has two components, magnitude and direction, that are referred to as its “real” and “imaginary” components, respectively. An example of a complex number is 5 + 3*i*, where 5 is the real component and 3*i* is the imaginary component.

The names of HP TestCore API functions used to create and manipulate complex numbers contain the word “Complex.” For example, function `UtaComplexCreate()` creates a data container that contains a complex number.

Point Data

Point data is a pair of numbers that define X,Y coordinates. For example, a point value of 3,4 means that X = 3 and Y = 4.

The names of HP TestCore API functions used to create and manipulate points contain the word “Point.” For example, function `UtaPointCreate()` creates a data container that contains a point.

Range Data

A range is a means of storing data that has a beginning, an end, and an incremental step size. One example of this is frequency sweep data. There

are several different models or ways of viewing range data. You can view a range as any of the following sets of characteristics:

start, **stop**, **step size**	The beginning and ending points of the range define its span or overall size. Because it defines the size of the incremental values in the range, step size determines how many incremental values the range has. Example: A range starts at 1 and stops at 10. If its step size is 1.8, it has 5.56 incremental values.
start, **stop**, **# of points**	The beginning and ending points of the range define its span or overall size. The number of points in the range determines how many incremental values it has, and the size of those increments. Example: A range starts at 1 and stops at 10. If its number of points is 5, it has 4 incremental values whose sizes are 2.25 each. (Note that start and stop are included in the number of points.)

center, span, step size	The span defines the overall size of the range around its center value. Because it defines the size of the incremental values in the range, step size determines how many incremental values the range has. Example: The center of a range is 6 and its span is 5. Thus, its maximum value is 11 (6 + 5) and its minimum value is 1 (6 - 5). If its step size is 1.67, it has 6 incremental values.
center, span, # of points	The span defines the overall size of the range around its center value. The number of points in the range determines how many incremental values it has, and the size of those increments. Example: The center of a range is 6 and its span is 5. Thus, its maximum value is 11 (6 + 5) and its minimum value is 1 (6 - 5). If its number of points is 7, it has 6 incremental values whose sizes are 1.67 each. (Note that the maximum and minimum values are included in the number of points.)

The names of HP TestCore API functions used to create and manipulate ranges contain the word "Range." For example, function `UtaRangeCreate()` creates a data container that contains a range and its parameters let you specify the starting value, ending value, and number of points for the range.

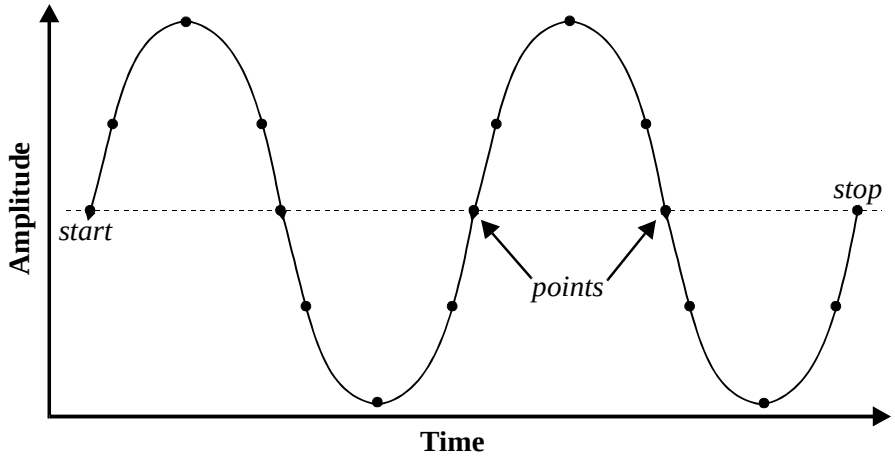
Although `UtaRangeCreate()` requires you to use the "start, stop, # of points" model when creating the range—i.e., its parameters are *dStart*, *dStop*, and *iNumPoints*—other HP TestCore API functions let you view the other models of the same data. For example, `UtaRangeGetSpan()` returns the span value from a range created using `UtaRangeCreate()`.

Note

Because the number of points is an integer, rounding may occur when you specify a step size.

Waveform Data

As shown below, a waveform typically is a plot of amplitude versus time.



Points are individual "samples" of the waveform's amplitude at specific times. The locations of points at intersections of the X (time) and Y (amplitude) axes define the shape of the waveform. The greater the number of points, the greater the potential resolution when representing a waveform.

In HP TestExec SL, waveform data—i.e., data that describes a particular waveform—is stored as a data type that contains:

- Range data (described earlier) to store the values of start, stop, and number of points
- An array of 64-bit real numbers to store the amplitude of each of the points with respect to time

The Y axis (amplitude) for the waveform is determined by the value of the points stored in the array. The X axis (time) is derived from the values of start, stop, and number of points.

The names of HP TestCore API functions used to create and manipulate waveforms contain the word "Waveform." For example, function `UtaWaveformCreate()`creates a data container that contains a waveform.

2

The C Action Development API Reference

This chapter describes the C Action Development API, whose functions let you use a C/C++ compiler to develop action routines. In many respects, this API is a general-purpose "tool kit" you can use to interact with various types of data.

See Chapter 3 in the *Using HP TestExec SL* book for more information.

Functions for Manipulating Data in Parameter Blocks

The functions described in this section let you work with data stored by name in data containers that reside in parameter blocks.

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaPbGetReal64()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains a 64-bit real number.

```
HUTAREAL64 UtaPbGetReal64(
    HUTAPB hParameterBlock,      // handle to a parameter  block
    LPCSTR lpszName,             // name of parameter in block
    UtaReal64 *lpdValue          // pointer to data in data container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

**lpdValue*
Optional. A pointer to the data in the data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

If desired, you can directly access the value in the data container. If you pass the pointer to a 64-bit real variable in **lpdValue*, the value in the data container is immediately returned to that variable when this function is called. If you omit the **lpdValue* parameter or use it to pass a NULL pointer when calling this function, no action is taken on **lpdValue*.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the value is useful when speed and simplicity are most important.

Example

```
// EXAMPLE OF ACCESSING DATA DIRECTLY VIA A POINTER
// Note: This is the recommended method for simple data types.
HUTAREAL64 hMyData; // handle to 64-bit real data
double dMyReal64; // variable to hold returned value
hMyData = UtaPbGetReal64(hMyParmBlock, "MyParm", &dMyReal64);
// dMyReal64 now contains 64-bit real value

// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a 64-bit real parameter named MyParm
HUTAREAL64 hMyData; // handle to 64-bit real data
double dMyReal64; // variable to hold returned value
// get handle to data container that contains a 64-bit real number
hMyData = UtaPbGetReal64(hMyParmBlock, "MyParm");
// get value from data container
dMyReal64 = UtaReal64GetValue(hMyData);
```

See also

UtaPbSetReal64()
UtaPbGetInt32()

UtaPbSetReal64()

This function sets the value of a 64-bit real number in a data container that contains a parameter found by name in a specified parameter block.

```
HUTAREAL64 UtaPbSetReal64(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName,            // name of parameter in parameter block
    UtaReal64 dValue            // desired value of data in data container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

dValue
The value to which the data in the data container is to be set.

Return Value

If *lpszName* is found in the parameter block, a handle to the data container is returned for subsequent use and the value of the number is set to *dValue*. Otherwise, this function returns NULL to indicate the specified parameter was not found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains 64-bit real
// data in a parameter (data container) named Real64Data
HUTAREAL64 hMyUtaReal64;
double dValue = 2.54;
// Write new value to parameter named Real64Data
hMyUtaReal64 = UtaPbSetReal64(
   hMyParmBlock,
   "Real64Data",
   dValue
   ;
```

See also

UtaPbGetReal64()
UtaPbSetInt32()

UtaPbGetInt32()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains a 32-bit integer number.

HUTAINT32 UtaPbGetInt32(

HUTAPB *hParameterBlock*,	// handle to a parameter block
LPCSTR *lpszName*,	// name of parameter in parameter block
UtaInt32 **lplValue*	// pointer to data in data container
);	

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

**lplValue*
: Optional. A pointer to the data in the data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

If desired, you can directly access the value in the data container. If you pass the pointer to a 32-bit integer variable in **lpdValue*, the value in the data container is immediately returned to that variable when this function is called. If you omit the **lpdValue* parameter or use it to pass a NULL pointer when calling this function, no action is taken on **lpdValue*.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the value is useful when speed and simplicity are most important.

Example

```
// EXAMPLE OF ACCESSING DATA DIRECTLY VIA A POINTER
// Note: This is the recommended method for simple data types.
HUTAINT32 hMyData; // handle to 32-bit integer data
long lMyInt32; // variable to hold returned value
hMyData = UtaPbGetInt32(hMyParmBlock, "MyParm", &lMyInt32);
// lMyInt32 now holds 32-bit integer value.

// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a 32-bit integer parameter named MyParm
HUTAINT32 hMyData; // handle to integer data
long lMyInt32; // variable to hold returned value
// get handle to data container that contains a 32-bit integer number
hMyData = UtaPbGetInt32(hMyParmBlock, "MyParm");
// get value from data container
lMyInt32 = UtaInt32GetValue(hMyData);
```

See also

UtaPbSetInt32()
UtaPbGetReal64()

UtaPbSetInt32()

This function sets the value of a 32-bit integer number in a data container that contains a parameter found by name in a specified parameter block.

```
HUTAINT32 UtaPbSetInt32(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName,            // name of parameter in parameter block
    UtaInt32 lValue             // desired value of data in data container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

lValue
The value to which the data in the data container in the parameter block is to be set.

Return Value

If *lpszName* is found in the parameter block, a handle to the 32-bit integer number in the data container is returned and the value of the number is set to *iValue*. Otherwise, this function returns NULL to indicate the specified parameter was not found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains 32-bit integer
// data in a parameter (data container) named Int32Data
HUTAINT32 hMyUtaInt32;
long lValue = 25;
// Write new value to parameter named IntData
hMyUtaInt32 = UtaPbSetInt32(
   hMyParmBlock,
   "Int32Data",
   lValue
   );
```

See also

UtaPbGetInt32()
UtaPbSetReal64()

UtaPbGetString()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains a string.

HUTASTRING UtaPbGetString(

HUTAPB *hParameterBlock*,	// handle to a parameter block
LPCSTR *lpszName*,	// name of parameter in parameter block
LPSTR *lpszBuffer*,	// buffer to receive characters from string
int *iMax*	// max # of chars to be copied
);	

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
: The name of a parameter associated with a data container in the parameter block.

lpszBuffer
: Optional. A buffer used to receive characters copied from the string data container. Defaults to NULL.

iMax
: Optional. The maximum number of characters to be copied from the string. Defaults to zero.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this functions returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

If desired, you can directly access the value in the data container. If you pass the pointer to a character buffer in *lpszBuffer*, the value in the data container is immediately returned to that buffer when this function is called. If you omit the *lpszBuffer* parameter or use it to pass a NULL pointer when calling this function, no action is taken on lpszBuffer.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the value is useful when speed and simplicity are most important.

If *iMax* is provided and is non-zero, then a maximum of *iMax* bytes (char) is copied to the buffer. If *iMax* is zero, the entire string is copied without regard for the size of *lpszBuffer.*

Example

```
// EXAMPLE OF DIRECTLY ACCESSING DATA & COPYING SPECIFIED # OF CHARS
// Note: This is the recommended method for simple data types.
HUTASTRING hMyData; // handle to string data
char chMyString[20]; // variable to hold returned value
hMyData = UtaPbGetString(hMyParmBlock, "MyParm", chMyString, 10);
// chMyString now contains 10 characters from string in data container
```

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a string parameter named MyParm
HUTASTRING hMyData; // handle to string data
char chMyString[20]; // variable to hold returned value
// get handle to data container that contains a string
hMyData = UtaPbGetString(hMyParmBlock, "MyParm");
// get value from data container
chMyString = UtaStringGetValue(hMyData);
```

See also

UtaPbSetString()

UtaPbSetString()

This function sets the value of a string in a data container that contains a parameter found by name in a specified parameter block.

HUTASTRING UtaPbSetString(

HUTAPB *hParameterBlock*,	// handle to a parameter block
LPCSTR *lpszName*,	// name of parameter in parameter block
LPCSTR *lpszValue*	// desired value of string in data container

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

lpszValue
The value to which the string in the data container in the parameter block is to be set.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this functions returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The string will expand or contract to hold the all of the characters in *lpszValue*.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains string
// data in a parameter (data container) named StringData
HUTASTRING hMyUtaString;
char chValue[] = "Hello";
// Write new value to parameter named StringData
hMyUtaString = UtaPbSetString(
   hMyParmBlock,
   "StringData",
   chValue
   );
```

See also

UtaPbGetString()

UtaPbGetPath()

This function retrieves the handle to a switching path object found by name as a parameter in a specified parameter block. Use this function when the

parameter is a switching path. The handle can then be used in the `UtaPathConnect()` and `UtaPathDisconnect()` functions.

```
HUTAPATH UtaPbGetPath(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName             // name of parameter in parameter block
);
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a switching path object in the parameter block.

Return Value
If *lpszName* is found in the parameter block, this function returns a handle to the switching path object. Otherwise, it function returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

Initial switching conditions usually are set up in the test via the Switching Action Editor instead of programmatically.

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path hOriginalState
```

```
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

See also

UtaPathConnect()
UtaPathDisconnect()

UtaPbGetComplex()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains data whose type is complex.

HUTACOMPLEX UtaPbGetComplex(

HUTAPB *hParameterBlock*,	// handle to a parameter block
LPCSTR *lpszName*,	// name of parameter in block
UtaReal64 **lpdReal*,	// pointer to real component in data container
UtaReal64 **lpdImag*	// pointer to imaginary component in data container

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
: The name of a parameter associated with a data container in the parameter block.

**lpdReal*
: Optional. A pointer to the real component in the data container in the parameter block. Defaults to NULL.

**lpdImag*
: Optional. A pointer to the imaginary component in the data container in the parameter block. Defaults to NULL.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

If desired, you can directly access the values in the data container. If you pass the pointers to 64-bit real variables in *lpdReal* and *lpdImag*, the values in the data container are immediately returned to those variables when this function is called. If you omit the *lpdReal* and *lpdImag* parameters or use them to pass NULL pointers when calling this function, no action is taken on them.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the value is useful when speed and simplicity are most important. The handle can be used to access the individual fields via the `UtaComplex...()` functions.

Example

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Note: This is the recommended method for complicated data types.
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a complex parameter named MyParm
HUTACOMPLEX hMyData; // handle to complex data
double dMyReal, dMyImag; // variables to hold returned values
// get handle to data container that contains complex data
hMyData = UtaPbGetComplex(hMyParmBlock, "MyParm");
// get values from data container
dMyReal = UtaComplexGetReal(hMyData); // get real value
dMyImag = UtaComplexGetImag(hMyData); // get imaginary value

// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
HUTACOMPLEX hMyData; // handle to complex data
double dMyReal, dMyImag; // variables to hold returned values
hMyData = UtaPbGetComplex(hMyParmBlock, "MyParm", &MyReal, &MyImag);
// MyReal now has value of real component & MyImag now has value
// of imaginary component of complex data.
```

See also

various UtaComplex...() functions
UtaPbSetComplex()

UtaPbSetComplex()

This function sets the values of the "real" and "imaginary" components of a complex data type in a data container that contains a parameter found by name in a specified parameter block.

```
HUTACOMPLEX UtaPbSetComplex(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName,            // name of parameter in parameter
                                // block
    UtaReal64 dReal,            // real component in data container
    UtaReal64 dImag             // imaginary component in data
                                // container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

dReal
The 64-bit real value to which the real component in the data container in the parameter block is to be set.

dImag
The 64-bit real value to which the imaginary component in the data container in the parameter block is to be set.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container and sets the values of its "real" and "imaginary" components to the values of *dReal* and *dImag*, respectively. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains complex
// data in a parameter (data container) named ComplexData
HUTACOMPLEX hMyUtaComplex;
double dReal = 1, dImaginary = -2;
// Write new values to parameter named ComplexData
hMyUtaComplex = UtaPbSetComplex(
   hMyParmBlock,
   "ComplexData",
   dReal,
   dImaginary
                );
```

See also

UtaPbGetComplex()

UtaPbGetRange()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains data whose type is range.

HUTARANGE UtaPbGetRange(

HUTAPB *hParameterBlock*,	// handle to a parameter block
LPCSTR *lpszName*,	// name of parameter in parameter block
UtaReal64 **lpdStart*,	// beginning value of range in data container
UtaReal64 **lpdStop*,	// ending value of range in data container
UtaInt16 **lpiPoints*	// # of points in range in data container

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

**lpdStart*
Optional. A pointer to the starting value of a range in the data container in the parameter block. Defaults to NULL.

**lpdStop*
Optional. A pointer to the ending value of a range in the data container in the parameter block. Defaults to NULL.

**lpiPoint*
Optional. A pointer to the number of points in a range in the data container in the parameter block. Defaults to NULL.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

If desired, you can directly access the values in the data container. If you pass the pointers to 64-bit real variables in *lpdStart*, *lpdStop* and *lpdPoints*, the values in the data container are immediately returned to those variables when this function is called. If you omit the *lpdStart*, *lpdStop* and *lpdPoints* parameters or use them to pass NULL pointers when calling this function, no action is taken on them.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the value is useful when speed and simplicity are most important. The handle retrieved by this function can be used to access individual fields via the `UtaRange()` function.

Example

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Note: This is the recommended method for complicated data types.
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a range parameter named MyParm
HUTARANGE hMyData; // handle to range data
double dMyStart, dMyStop; // variables to hold returned values
long lMyPoints; // variable to hold returned value
// get handle to data container that contains range data
hMyData = UtaPbGetRange(hMyParmBlock, "MyParm");
// get values from data container
dMyStart = UtaRangeGetStart(hMyData); // get start value
dMyStop = UtaRangeGetStop(hMyData); // get stop value
lMyPoints = UtaRangeGetNumPoints(hMyData); // get # of points value

// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
HUTARANGE hMyData; // handle to range data
double dMyStart, dMyStop; // variables to hold returned values
long lMyPoints; // variable to hold returned value
hMyData = UtaPbGetRange(
   hMyParmBlock,
   "MyParm",
   &dMyStart,
   &dMyStop,
   &lMyPoints
   );
// dMyStart now has value of start, dMyStop now has value of stop
// & lMyPoints now has value of # of points in range data.
```

See also

various UtaRange...() functions
UtaPbSetRange()

UtaPbSetRange()

This function sets the values of the "start", "stop" and "points" components of range data in a data container that contains a parameter found by name in a specified parameter block.

```
HUTARANGE UtaPbSetRange(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName,            // name of parameter in parameter block
    UtaReal64 dStart,           // desired beginning value for range in
                                // data container
    UtaReal64 dStop,            // desired ending value for range in data
                                // container
    UtaInt16 iPoints            // desired # of points in range in data
                                // container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

dStart
The 32-bit real value to which the starting value for the range data in the data container is to be set.

dStop
The 32-bit real value to which the ending value for the range data in the data container is to be set.

iPoints
The 16-bit integer value to which the number of points in the range data in the data container is to be set.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains range
// data in a parameter (data container) named RangeData
HUTARANGE hMyUtaRange;
double dStart = 1, dStop = 10;
long lPoints = 5;
// Write new values to parameter named RangeData
hMyUtaRange = UtaPbSetRange(
   hMyParmBlock,
   "RangeData",
   dStart,
   dStop,
   lPoints
   );
```

See also

UtaPbGetRange()

UtaPbGetPoint()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains data whose type is point.

```
HUTAPOINT UtaPbGetPoint(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName,            // name of parameter in parameter block
    UtaReal64 *lpdX,            // pointer to X component in data
                                // container
    UtaReal64 *lpdY             // pointer to Y component in data
                                // container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

**lpdX*
Optional. A pointer to the X component of the point data in the data container in the parameter block. Defaults to NULL.

**lpdY*
Optional. A pointer to the Y component of the point data in the data container in the parameter block. Defaults to NULL.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

If desired, you can directly access the value in the data container. If you pass the pointers to 64-bit real variables in **lpdX* and **lpdY*, the values in the data container are immediately returned to these variables when this function is called. If you omit the **lpdX* and **lpdY* parameters or use them to pass a NULL pointer when calling this function, no action is taken on **lpdX* and **lpdY*.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the value is useful when speed and simplicity are most important. The handle retrieved by this function can be used to access individual fields via the `UtaComplex()` function.

Example

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Note: This is the recommended method for complicated data types.
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a point parameter named MyParm
HUTAPOINT hMyData; // handle to point data
double dMyX, dMyY; // variables to hold returned values
// get handle to data container that contains point data
hMyData = UtaPbGetPoint(hMyParmBlock, "MyParm");
// get values from data container
dMyX = UtaPointGetX(hMyData); // get X value
dMyY = UtaPointGetY(hMyData); // get Y value

// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
HUTAPoint hMyData; // handle to point data
double dMyX, dMyY; // variables to hold returned values
hMyData = UtaPbGetPoint(
   hMyParmBlock,
   "MyParm",
   &dMyX,
   &dMyY
   );
// dMyX now has X value and dMyY now has Y value of point data.
```

See also

various UtaComplex...() functions
UtaPbSetPoint()

UtaPbSetPoint()

This function sets the values for the X and Y parts of a point data type in a data container that contains a parameter found by name in a specified parameter block.

```
HUTAPOINT UtaPbSetPoint(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName,            // name of parameter in parameter block
    UtaReal64 dX,               // desired value for X component in data
                                // container
    UtaReal64 dY                // desired value for Y component in data
                                // container
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

dX
The 64-bit real value to which the X component of the point data in a data container in the parameter block is to be set.

dY
The 64-bit real value to which the Y component of the point data in a data container in the parameter block is to be set.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container and sets X and Y to *dX* and *dY.* Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains point
// data in a parameter (data container) named PointData
HUTAPOINT hMyUtaPoint;
double dX = 1, dY = 5;
// Write new values to parameter named PointData
hMyUtaPoint = UtaPbSetPoint(
   hMyParmBlock,
   "PointData",
   dX,
   dY,
   );
```

See also

various UtaPoint...() functions
UtaPbGetPoint()

UtaPbGetReal64Array()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the the data container contains an array of 64-bit real numbers.

HUTAR64ARR UtaPbGetReal64Array(

HUTAPB *hParameterBlock*, // handle to a parameter block

LPCSTR *lpszName* // name of parameter in parameter block

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

Return Value
If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The API functions prefixed with "UtaR64Arr" described elsewhere are used to get and retrieve values via the handle returned by this function.

Example

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains an array parameter named MyParm
HUTAR64ARR hMyData; // handle to 64-bit real array data
double dMyReal64;
// get handle to data container that contains array data
hMyData = UtaPbGetReal64Array(hMyParmBlock, "MyParm");
// use handle to return value from element 3
dMyReal64 = UtaR64ArrGetAt1(hMyData, 3);
```

See also

various `UtaR64Arr...()` functions

UtaPbGetR64Arr()

This function is an alternate form of `UtaPbGetReal64Array()`. See the description of that function for information about this function's return value and parameters.

UtaPbGetInt32Array()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains an array of 32-bit integer numbers.

```
HUTAI32ARR UtaPbGetInt32Array(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName             // name of parameter in parameter block
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The API functions prefixed with "UtaI32Arr" described elsewhere are used to get and retrieve values via the handle returned by this function.

Example

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains an array parameter named MyParm
HUTAI32ARR hMyData; // handle to 32-bit integer array data
long lMyInt32;
// get handle to data container that contains array data
hMyData = UtaPbGetInt32Array(hMyParmBlock, "MyParm");
// use handle to return value from element 3
lMyInt32 = UtaI32ArrGetAt1(hMyData, 3);
```

See also

various `UtaI32Arr...()` functions

UtaPbGetI32Arr()

This function is an alternate form of `UtaPbGetInt32Array()`. See the description of that function for information about this function's return value and parameters.

UtaPbGetStringArray()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains an array of strings.

HUTASTRARR UtaPbGetStringArray(

HUTAPB *hParameterBlock*, // handle to a parameter block

LPCSTR *lpszName* // name of parameter in parameter block

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName

The name of a parameter associated with a data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The API functions prefixed with "UtaStrArr" described elsewhere are used to get and retrieve values via the handle returned by this function.

Example

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a string parameter named MyParm
HUTASTRARR hMyData; // handle to string array data
char chMyString[20];
// get handle to data container that contains string array data
hMyData = UtaPbGetStringArray(hMyParmBlock, "MyParm");
```

See also

various `UtaStrArr...()` functions

UtaPbGetStrArr()

This function is an alternate form of `UtaPbGetStringArray()`. See the description of that function for information about this function's return value and parameters.

UtaPbGetPointArray()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains an array of point data.

HUTAPTARR UtaPbGetPointArray(

HUTAPB *hParameterBlock*, // handle to a parameter block

LPCSTR *lpszName* // name of parameter in parameter block

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The API functions prefixed with "UtaPtArr" described elsewhere are used to retrieve or update values via the handle returned by this function.

Example

```
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a point array parameter named MyParm
HUTAPTARR hMyData; // handle to point array data
// get handle to data container that contains point array data
hMyData = UtaPbGetPointArray(hMyParmBlock, "MyParm");
```

See also

various `UtaPtArr...()` functions

UtaPbGetPtArr()

This function is an alternate form of `UtaPbGetPointArray()`. See the description of that function for information about this function's return value and parameters.

UtaPbGetRangeArray()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains an array of range data.

```
HUTARNGARR UtaPbGetRangeArray(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName             // name of parameter in parameter block
);
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The API functions prefixed with "UtaRngArr" described elsewhere are used to retrieve or update values via the handle returned by this function.

Example

```
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a range array parameter named MyParm
HUTARNGARR hMyData; // handle to range array data
// get handle to data container that contains range array data
hMyData = UtaPbGetRangeArray(hMyParmBlock, "MyParm");
```

See also

various `UtaRngArr...()` functions

UtaPbGetRngArr()

This function is an alternate form of `UtaPbGetRangeArray()`. See the description of that function for information about this function's return value and parameters.

UtaPbGetWaveform()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains waveform data.

HUTAWAVEFORM UtaPbGetWaveform(

HUTAPB *hParameterBlock*, // handle to a parameter block

LPCSTR *lpszName* // name of parameter in parameter block

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName

The name of a parameter associated with a data container in the parameter block.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

The API functions prefixed with "UtaWaveform" described elsewhere are used to retrieve or update values via the handle returned by this function.

Example

```
// Example assumes that:
// - parameter block MyParmBlock exists & its handle is hMyParmBlock
// - MyParmBlock contains a waveform parameter named MyParm
HUTAWAVEFORM hMyData; // handle to waveform data
// get handle to data container that contains waveform data
hMyData = UtaPbGetWaveform(hMyParmBlock, "MyParm");
```

See also

various `UtaWaveform...()` functions

UtaPbGetInst()

This function retrieves the handle to a data container that contains a parameter found by name in a specified parameter block. Use this function when the data container contains instrument data.

HUTAINST UtaPbGetInst(

HUTAPB *hParameterBlock*, // handle to a parameter block

LPCSTR *lpszName*, // name of parameter in parameter block

UTAINT32 **lpViSession* // pointer to a VXIplug&play ViSession

);

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a parameter associated with a data container in the parameter block.

**lpViSession*
Optional pointer to a VXI*plug&play* ViSession. When using VXI*plug&play* instrument drivers, the *ViSession* is returned by passing in a memory address. VXI*plug&play* drivers will need to be passed this *ViSession*. If NULL (which is the default value) is passed in as the address, nothing is passed back.

Return Value

If a parameter whose name matches *lpszName* is found in the parameter block, this function returns a handle to the data container. Otherwise, it returns a NULL value to indicate the specified parameter was not found in the parameter block.

Remarks

Using the optional **lpViSession* parameter is a shortcut that combines the functionality of `UtaInstGetViSession()` with this function; i.e., in a single function call you can get the handle to the instrument and the identifier of its ViSession. The shortcut syntax looks like this:

```
HUTAINST hInstrument;
unsigned int ViSession; // variable to store ID of ViSession
hInstrument = UtaPbGetInst(hParameterBlock, "InstrName", &ViSession);
```

Example

```
void UTADLL ProgramPowerSupply (HUTAPB hParameterBlock)
{
// Action routine that programs an HP 66312 power supply.
// Example assumes that parameter block contains three parameters:
//   Voltage - type Real64
//   Current - type Real64
//   PowerSupply - type Inst
```

```
// Assign miscellaneous variables
HUTAREAL64 hData;
ViStatus ErrorCodes;
HUTAINST hInstrument;

// Get value of voltage from parameter block
hData = UtaPbGetReal64(hParameterBlock, "Voltage");
double dVolt = UtaReal64GetValue(hData);

// Get value of current from parameter block
hData = UtaPbGetReal64(hParameterBlock, "Current");
double dCurr = UtaReal64GetValue(hData);

// Get the ViSession identifier from the parameter block
hInstrument = UtaPbGetInst(hParameterBlock, "PowerSupply");
long lViSession = UtaInstGetViSession(hInstrument);

// Set the voltage & current, and turn on the output
ErrorCodes = hp66312_voltCurrOutp (lViSession, dVolt, dCurr);

...(optional code that checks ErrorCodes for power supply errors)

return;
}
```

See also

UtaInstGetViSession()

Functions for Locating Data in Parameter Blocks

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaPbFindId()

This function returns the ID of a data container that contains the data for a parameter found by name in a specified parameter block.

```
IDUTAPARM UtaPbFindId(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszName             // name of parameter in parameter block
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lpszName
The name of a data container in the parameter block.

Return Value

If the parameter is found in the parameter block, this function returns the ID of its data container. Otherwise, it returns NULL to indicate the specified parameter was not found in the parameter block.

Remarks

Each parameter has both an ID and a name. The ID is an integer value that uniquely identifies the location of its data container within the specified parameter block.

IDs are consecutive numbers with values between 1 and the size of the parameter block, which you can use `UtaPbGetSize()` to determine. Given the ID and the size of the parameter block, you can use `UtaPbGetData()` in a "for" loop that starts at 1 and ends at the parameter block's size to browse or "walk" a parameter block and examine all of the data in it.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains a
// parameter (data container) named MyData
IDUTAPARM ID;
char chString[40];
if (UtaPbFindId(hMyParmBlock, "MyData") != NULL) // if the parm exists
   {
   ID = UtaPbFindId(hMyParmBlock, "MyData"); // get the ID
   sprintf(chString, "The ID is %d", ID);
   AfxMessageBox(chString, MB_OK);
   }
else
   AfxMessageBox("Parameter not found in block!", MB_OK);
```

See also

UtaPbGetParmName()

UtaPbGetParmName()

This function returns the name of a parameter if its ID is found in a specified parameter block.

LPCSTR UtaPbGetParmName(

HUTAPB *hParameterBlock*, // handle to a parameter block

IDUTAPARM *idParameter* // ID of parameter in parameter block

);

Parameters

hParameterBlock
The handle to a parameter block.

idParameter
The ID of a parameter to be found in the parameter block.

Return Value

If a parameter whose ID matches *idParameter* is found in the parameter block, this function returns the name of the parameter. Otherwise, it returns a NULL value to indicate the specified ID was not found in the parameter block.

Remarks

Each parameter has both an ID and a name. The ID is an integer value that uniquely identifies the location of its data container within the specified parameter block.

IDs are consecutive numbers with values between 1 and the size of the parameter block, which you can use `UtaPbGetSize()` to determine. Given the ID and the size of the parameter block, you can use `UtaPbGetData()` in a "for" loop that starts at 1 and ends at the parameter block's size to browse or "walk" a parameter block and examine all of the data in it.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// that its handle is hMyParmBlock, and that it contains a
// parameter (data container) named MyData whose Id is 5
LPCSTR lpcstrParmName;
char chString[40];
if (UtaPbGetParmName(hMyParmBlock, 5) != NULL) // if the ID exists
   {
   lpcstrParmName = UtaPbGetParmName(hMyParmBlock, 5); // get the name
   sprintf(chString, "Name of parameter is %s", lpcstrParmName);
   AfxMessageBox(chString, MB_OK);
   }
else
   AfxMessageBox("Parameter not found in block!");
```

See also

UtaPbFindId()

UtaPbGetSize()

This function returns the number of parameters in a specified parameter block.

```
int UtaPbGetSize(
    HUTAPB hParameterBlock      // handle to a parameter block
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

Return Value

Returns a 32-bit integer value that reports how many parameters were found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists,
// and that its handle is hMyParmBlock
long lNumberOfParameters;
char chString[40];
lNumberOfParameters = UtaPbGetSize(hMyParmBlock); // return # of parms
sprintf(chString, "Number of parameters = %ld", lNumberOfParameters);
AfxMessageBox(chString, MB_OK);
```

UtaPbFindData()

This function searches a specified parameter block for a parameter with a given name.

```
HUTADATA UtaPbFindData(
    HUTAPB hParameterBlock,     // handle to a parameter block
    LPCSTR lpszParmName         // name of parameter in parameter block
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

lszParmName
The name of a parameter to be searched for in the parameter block.

Return Value

If the parameter is found in the parameter block, this function returns the handle to the data container in which the parameter's data resides. Other, it returns a NULL value to indicate the desired parameter was not found in the parameter block.

Example

```
// Example assumes that parameter block MyParmBlock already exists
// and that its handle is hMyParmBlock
HUTADATA hDataContainer;
if (UtaPbFindData(hMyParmBlock, "MyData) != NULL)
   hDataContainer = UtaPbFindData(hMyParmBlock, "MyData));
```

UtaPbGetData()

This function searches for a given parameter ID in a specified parameter block.

```
HUTADATA UtaPbGetData(
    HUTAPB hParameterBlock,     // handle to a parameter block
    IDUTAPARM idParm            // ID of a parameter in parameter block
    );
```

Parameters

hParameterBlock
The handle to a parameter block.

idParm
The ID of a parameter to be searched for in the parameter block.

Return Value

If the specified ID is found in the parameter block, this function returns the handle to the data container that contains the data associated with the ID. Otherwise, the function returns a NULL value to indicate the ID was not found in the parameter block.

Remarks

Each parameter has both an ID and a name. The ID is an integer value that uniquely identifies the location of its data container within the specified parameter block.

IDs are consecutive numbers with values between 1 and the size of the parameter block, which you can use `UtaPbGetSize()` to determine. Given the ID and the size of the parameter block, you can use `UtaPbGetData()` in a “for” loop that starts at 1 and ends at the parameter block’s size to browse or “walk” a parameter block and examine all of the data in it.

Example

```
// Example assumes that parameter block MyParmBlock already exists
// and that its handle is hMyParmBlock
HUTADATA hDataContainer;
IDUTAPARM ID = 5;
if (UtaPbGetData(hMyParmBlock, ID) != NULL)
   hDataContainer = UtaPbGetData(hMyParmBlock, ID)); // get the handle
```

See also

UtaPbFindId()

UtaTableRegFindData()

This function searches by name for data stored in one or all symbol tables.

HUTADATA UtaTableRegFindData(

LPCSTR *lpszParmEntryName*, // name of data to be found

BOOL *bEntryMatchCase*, // sets case for data name

LPCSTR *lpszTableName*, // name of table to search

BOOL *bTableMatchCase* // sets case for table name

);

Parameters

lpszParmEntryName
Name of the data to be searched for in symbol table(s).

bEntryMatchCase
Optional. If set to TRUE, matches data names regardless of whether names are upper- or lower-case. Defaults to TRUE.

lpszTableName
Optional. Name of symbol table to search for the specified data. Default is to search all symbol tables.

bTableMatchCase

Optional. If set to TRUE, matches symbol table names regardless of whether names are upper- or lower-case. Defaults to TRUE.

Return Value

If this function finds the data of interest, it returns the handle to the data container in which the data is stored. Otherwise, it returns a NULL value to indicate the data was not found in the specified symbol table(s).

Remarks

This function defaults to searching all of the symbol tables. Optionally, you can give it the name of a specific symbol table to search. Also, you can optionally specify case sensitivity. This function is case-sensitive, and by default matches on both upper- and lower-case.

Example

```
// Example searches for named data in all symbol tables.
// Search is not case-sensitive.
HUTADATA hData;
if (UtaTableRegFindData("MyParm") != NULL)
   {
   hData=UtaTableRegFindData("MyParm");
   // hData now has handle to data container in a symbol table
   AfxMessageBox("Found the data!");
   }
else
   AfxMessageBox("Unable to find the data!");
```

```
// Example searches for named data only in TestStepLocals symbol
// table. Search is case-sensitive for both name of data & name
// of symbol table.
HUTADATA hData;
if (UtaTableRegFindData(
     "MyParm",
     FALSE,
     "TestStepLocals",
     FALSE) != NULL)
   {
   hData=UtaTableRegFindData(
     "MyParm",
     FALSE,
     "TestStepLocals",
     FALSE);
   // hData now has handle to data container in a symbol table
   AfxMessageBox("Found the data!");
   }
else
   AfxMessageBox("Unable to find the data!");
```

Functions for Manipulating Data in Data Containers

The functions described in this section let you work with data stored in data containers that do not reside in parameter blocks.

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaReal64Create()

This function creates a new data container that contains a 64-bit real number, assigns a value to the number, and then returns a handle to the newly created container.

```
HUTAREAL64 UtaReal64Create(
    UtaReal64 dValue          // desired value of data in data container
);
```

Parameters

dValue
A value to be assigned to the newly created data container.

Return Value
Returns a handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be

recovered, which means you will have a long-term “memory leak” that can cause unstable operation of your test system.

Example

```
double dMyReal64;
HUTAREAL64 hMyReal64Data; // declare variable for 64-bit real data
hMyReal64Data = UtaReal64Create(3.25); // create & assign a value
   ...(do something)
dMyReal64 = UtaReal64GetValue(hMyReal64Data); // return current value
dMyReal64 += 8.32; // Add 8.32 to value of MyReal64
UtaReal64SetValue(hMyReal64Data, dMyReal64); // set to new value
   ...(do something)
UtaDataRelease(hMyReal64Data); // delete data container & free memory
```

See also

UtaReal64GetValue()
UtaReal64SetValue()
UtaDataCopy()
UtaDataRelease()

UtaReal64GetValue()

This function returns the 64-bit real value stored in a specified data container.

UtaReal64 UtaReal64GetValue(

HUTAREAL64 *hData* // handle to a data container

);

Parameters

hData
The handle to a data container that contains a 64-bit real number.

Return Value

Returns the 64-bit real value stored in the specified data container.

Remarks

If your compiler does not support returning 64-bit real numbers through a C function name compatible with Microsoft compilers, use function `UtaReal64GetDataPtr()` instead.

Example

```
double dMyReal64;
HUTAREAL64 hMyReal64Data; // declare variable for 64-bit real data
hMyReal64Data = UtaReal64Create(3.25); // create & assign a value
   ...(do something)
dMyReal64 = UtaReal64GetValue(hMyReal64Data); // return current value
dMyReal64 += 8.32; // Add 8.32 to value of MyReal64
UtaReal64SetValue(hMyReal64Data, dMyReal64); // set to new value
   ...(do something)
UtaDataRelease(hMyReal64Data); // delete data container & free memory
```

See also

UtaReal64Create()
UtaReal64SetValue()
UtaReal64GetDataPtr() if needed

UtaReal64SetValue()

This function updates the value of a 64-bit real number stored in a specified data container.

void UtaReal64SetValue(

HUTAREAL64 *hData*, // handle to a data container

UtaReal64 *value* // desired value of data in data container

);

Parameters

hData
The handle to a data container that contains a 64-bit real number.

value
A 64-bit real value to replace the existing value in the data container.

Return Value

(none)

Example

```
double dMyReal64;
HUTAREAL64 hMyReal64Data; // declare variable for 64-bit real data
hMyReal64Data = UtaReal64Create(3.25); // create & assign a value
   ...(do something)
dMyReal64 = UtaReal64GetValue(hMyReal64Data); // return current value
dMyReal64 += 8.32; // Add 8.32 to value of MyReal64
UtaReal64SetValue(hMyReal64Data, dMyReal64); // set to new value
   ...(do something)
UtaDataRelease(hMyReal64Data); // delete data container & free memory
```

See also

UtaReal64Create()
UtaReal64GetValue()

UtaReal64GetDataPtr()

This function returns a pointer to a 64-bit real number in a specified data container.

UtaPtrReal64 UtaReal64GetDataPtr(

HUTAREAL64 *hData* // handle to a data container

);

Parameters

hData
The handle to a data container that contains a 64-bit real number.

Return Value

Returns a pointer to the data stored in the specified data container.

Remarks

Use this function if your compiler does not support returning 64-bit real numbers through a C function that is compatible with Microsoft compilers.

Example

```
HUTAREAL64 hMyReal64Data; // declare variable for 64-bit real data
UtaPtrReal64 pMyReal64;
pMyReal64 = &MyReal64;
hMyReal64Data = UtaReal64Create(3.25); // create & assign a value
   ...(do something)
pMyReal64 = UtaReal64GetDataPtr(hMyReal64Data); // get pointer
*pMyReal64 += 8; // Add 8 to value of pMyReal64
   ...(do something)
UtaDataRelease(hMyReal64Data); // delete data container & free memory
```

See also

UtaReal64Create()
UtaReal64SetValue()

UtaInt32Create()

This function creates a new data container that contains a 32-bit integer number and returns a handle to the newly created data container.

HUTAINT32 UtaInt32Create(

UtaInt32 *value* // desired value for data in data container

);

Parameters

value

A value for the 32-bit integer number in the newly created data container.

Return Value

Returns a handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term "memory leak" that can cause unstable operation of your test system.

Example

```
long lMyInt32;
HUTAINT32 hMyInt32Data; // declare variable for 32-bit integer
hMyInt32Data = UtaInt32Create(3); // create & assign a value
   ...(do something)
lMyInt32 = UtaInt32GetValue(hMyInt32Data); // return current value
lMyInt32 += 8; // Add 8 to value of MyReal64
UtaInt32SetValue(hMyInt32Data, lMyInt32); // set to new value
   ...(do something)
UtaDataRelease(hMyInt32Data); // delete data container & free memory
```

See also

UtaInt32GetValue()
UtaInt32SetValue()
UtaDataCopy()
UtaDataRelease()

UtaInt32GetValue()

This function returns the value of a 32-bit integer number stored in a specified data container.

UtaInt32 UtaInt32GetValue(

HUTAINT32 *hData* // handle to a data container

);

Parameters

hData
The handle to a data container that contains a 32-bit integer number.

Return Value
Returns the value of a 32-bit integer value from the specified data container.

Example

```
long lMyInt32;
HUTAINT32 hMyInt32Data; // declare variable for 32-bit integer
hMyInt32Data = UtaInt32Create(3); // create & assign a value
   ...(do something)
lMyInt32 = UtaInt32GetValue(hMyInt32Data); // return current value
lMyInt32 += 8; // Add 8 to value of MyReal64
UtaInt32SetValue(hMyInt32Data, lMyInt32); // set to new value
   ...(do something)
UtaDataRelease((HUTADATA)hMyInt32Data); // delete data container
```

See also

UtaInt32Create()
UtaInt32GetDataPtr()
UtaInt32SetValue()

UtaInt32SetValue()

This function updates the value of a 32-bit integer number stored in a specified data container.

void UtaInt32SetValue(

HUTAINT32 *hData***,** // handle to a data container

UtaInt32 *value* // desired value for data in data container

);

Parameters

hData
The handle to a data container that contains a 32-bit integer number.

value
A 32-bit integer value to which the data in the data container should be updated.

Return Value

(none)

Example

```
long lMyInt32;
HUTAINT32 hMyInt32Data; // declare variable for 32-bit integer
hMyInt32Data = UtaInt32Create(3); // create & assign a value
   ...(do something)
lMyInt32 = UtaInt32GetValue(hMyInt32Data); // return current value
lMyInt32 += 8; // Add 8 to value of MyReal64
UtaInt32SetValue(hMyInt32Data, lMyInt32); // set to new value
   ...(do something)
UtaDataRelease((HUTADATA)hMyInt32Data); // delete data container
```

See also

UtaInt32Create()
UtaInt32GetValue()

UtaInt32GetDataPtr()

This function returns a pointer to a 32-bit integer number stored in a specified data container.

UtaPtrInt32 UtaInt32GetDataPtr(

HUTAINT32 *hData* // handle to a data container

);

Parameters

hData
The handle to a data container that contains a 32-bit integer number.

Return Value

Returns a 32-bit integer pointer to the data stored in the specified data container.

Remarks

Use this function if your compiler does not support returning 32-bit integer numbers through a C function that is compatible with Microsoft compilers.

Example

```
HUTAINT32 hMyInt32Data; // declare variable for 32-bit integer
UtaPtrInt32 pMyInt32;
pMyInt32 = &MyInt32;
hMyInt32Data = UtaInt32Create(5); // create & assign a value
   ...(do something)
pMyInt32 = UtaPtrInt32GetDataPtr(hMyInt32Data); // get pointer
*pMyInt32 += 8; // Add 8 to value of pMyInt32
   ...(do something)
UtaDataRelease((HUTADATA)hMyInt32Data); // delete data container
```

See also

UtaInt32Create()

UtaStringCreate()

This function creates a new data container that contains string data, stores a specified value in it, and returns a handle to the newly created data container.

HUTASTRING UtaStringCreate(

LPCSTR *value* // desired value for string in data container

);

Parameters

value
A value for the data in the newly created data container.

Return Value

Returns the handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term "memory leak" that can cause unstable operation of your test system.

Example

```
LPCSTR chMyString[10];
HUTASTRING hMyStringData; // declare variable for string
hMyStringData = UtaStringCreate("Monday"); // create & assign a value
   ...(do something)
chMyString = UtaStringGetValue(hMyStringData); // return current value
chMyString = strcpy(chMyString,"Tuesday"); // Change chMyString
UtaStringSetValue(hMyStringData, chMyString); // set to new value
   ...(do something)
UtaDataRelease((HUTADATA)hMyStringData); // delete data container
```

See also

UtaStringGetValue()
UtaStringSetValue()
UtaDataCopy()
UtaDataRelease()

UtaStringGetValue()

This function returns the value of a string stored in a specified data container.

LPCSTR UtaStringGetValue(

HUTASTRING *hData* // handle to a data container

);

Parameters

hData
The handle to a data container that contains a string.

Return Value

Returns the value of the data stored in the specified data container.

Remarks

Do not hold onto a LPCSTR returned by this function across any function call that changes the value of the string, such as `UtaStringSetValue()` or `UtaPbSetString()`. If you do, the value of the LPCSTR pointer may become invalid. Instead, use the LPCSTR pointer immediately, as shown in the example below.

Example

```
char chTempString[20];
LPCSTR lpcstrMyString;
HUTASTRING hMyStringData; // declare variable for string
hMyStringData = UtaStringCreate("Monday"); // create & assign a value
   ...(do something)
lpcstrMyString = UtaStringGetValue(hMyStringData); // return value
strcpy(chTempString,"Tuesday"); // Change chTempString
UtaStringSetValue(hMyStringData, chTempString); // set to new value
   ...(do something)
UtaDataRelease((HUTADATA)hMyStringData); // delete data container
```

See also

UtaStringSetValue()

UtaStringSetValue()

This function updates the value of a string stored in a specified data container.

```
void UtaStringSetValue(
    HUTASTRING hData,     // handle to a data container
    LPCSTR value          // desired value for string in data container
    );
```

Parameters

hData
The handle to a data container that contains a string.

value
A value to which the string data in the data container should be set.

Return Value

(none)

Example

```
char chTempString[20];
LPCSTR lpcstrMyString;
HUTASTRING hMyStringData; // declare variable for string
hMyStringData = UtaStringCreate("Monday"); // create & assign a value
   ...(do something)
lpcstrMyString = UtaStringGetValue(hMyStringData); // return value
strcpy(chTempString,"Tuesday"); // Change chTempString
UtaStringSetValue(hMyStringData, chTempString); // set to new value
   ...(do something)
UtaDataRelease((HUTADATA)hMyStringData); // delete data container
```

See also

UtaStringGetValue()

UtaR64ArrCreate()

This function creates a new data container that contains a single-dimensional array of 64-bit real numbers and returns the handle to the newly created data container.

```
HUTAR64ARR UtaR64ArrCreate(
    UtaInt16 lowerBound,        // desired lower boundary of array
    UtaInt16 upperBound         // desired upper boundary of array
);
```

Parameters

lowerBound
The lower boundary for elements in the array.

upperBound
The upper boundary for elements in the array.

Return Value

Returns the handle to the newly created data container.

Remarks

The size of the array is determined by the difference between *lowerBound* and *upperBound*. For compatibility with C, always specify *lowerBound* as 0.

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term “memory leak” that can cause unstable operation of your test system.

Example

```
double dMyReal64;
HUTAR64ARR hMyReal64Array; // declare variable (object) for array
hMyReal64Array = UtaR64ArrCreate(0, 9); // create array of 10 elements
UtaR64ArrSetAt1(2,4.5); // write value of 4.5 to element 2 of array
   ...(do something)
dMyReal64 = UtaR64ArrGetAt1(hMyReal64Array, 5); // value of element 5
   ...(do something)
UtaDataRelease((HUTADATA)hMyReal64Array); // delete data container
```

See also

UtaR64ArrGetAt1()
UtaR64ArrSetAt1()
UtaDataCopy()
UtaDataRelease()

UtaR64ArrGetBuffer()

This function returns a pointer to the start of an array of 64-bit real numbers stored in a specified data container.

UtaPtrReal64 UtaR64ArrGetBuffer(

HUTAR64ARR *hData* // handle to a data container

);

Parameters

hData
The handle to a data container that contains an array of 64-bit real numbers.

Return Value

Returns a pointer to the first element of the array stored in the data container.

Example

```
HUTAR64ARR hMyReal64Array; // declare variable (object) for array
UtaPtrReal64 pReal64Array[10];
pReal64Array = &Real64Array[];
hMyReal64Array = UtaR64ArrCreate(0, 9); // create array of 10 elements
   ...(do something)
// get pointer to array
pStartOfArray = UtaR64ArrGetBuffer(hMyReal64Array);
   ...(do something)
UtaDataRelease((HUTADATA)hMyReal64Array); // delete data container
```

UtaR64ArrGetAt1()

This function returns the value of an element in a single-dimensional array of 64-bit real numbers in a specified data container.

UtaReal64 UtaR64ArrGetAt1(

HUTAR64ARR *hR64Array*, // handle to a data container

UtaInt16 *iIndex1* // index of element in the array

);

Parameters

hR64Array
The handle to a data container that contains a single-dimensional array of 64-bit real numbers.

iIndex1
The index of an individual element in the array.

Return Value

Returns a 64-bit real value from the array element specified by *iIndex1*.

Remarks

Use function `UtaR64ArrGetBuffer()` to access a pointer to the C array.

Example

```
double dMyReal64;
HUTAR64ARR hMyReal64Array; // declare variable (object) for array
hMyReal64Array = UtaR64ArrCreate(0, 9); // create array of 10 elements
UtaR64ArrSetAt1(2,4.5); // write value of 4.5 to element 2 of array
   ...(do something)
dMyReal64 = UtaR64ArrGetAt1(hMyReal64Array, 5); // value of element 5
   ...(do something)
UtaDataRelease((HUTADATA)hMyReal64Array); // delete data container
```

See also

UtaR64ArrGetBuffer()
UtaR64ArrSetAt1()

UtaR64ArrSetAt1()

This function updates the value of an array element in a specified data container when the data container contains a single-dimensional array of 64-bit real numbers.

void UtaR64ArrSetAt1(

HUTAR64ARR *hR64Array*,	// handle to a data container
UtaInt16 *iIndex1*,	// index of an element in the array
UtaReal64 *dValue*	// desired value of the element at the index

);

Parameters

hR64Array
The handle to a data container that contains a single-dimensional array of 64-bit real numbers.

iIndex1
The index identifying which array element to update.

dValue
The 64-bit real value to which the array element should be set.

Return Value

(none)

Remarks

Use function `UtaR64ArrGetBuffer()` to access a pointer to the C array.

Example

```
double dMyReal64;
HUTAR64ARR hMyReal64Array; // declare variable (object) for array
hMyReal64Array = UtaR64ArrCreate(0, 9); // create array of 10 elements
UtaR64ArrSetAt1(2,4.5); // write value of 4.5 to element 2 of array
   ...(do something)
dMyReal64 = UtaR64ArrGetAt1(hMyReal64Array, 5); // value of element 5
   ...(do something)
UtaDataRelease((HUTADATA)hMyReal64Array); // delete data container
```

See also

UtaR64ArrGetAt1()
UtaR64ArrGetBuffer()

UtaR64ArrGetAt2()

This function returns the value of an array element in a specified data container when the data container contains a two-dimensional array of 64-bit real numbers.

UtaReal64 UtaR64ArrGetAt2(

HUTAR64ARR *hR64Array*,	// handle to a data container
UtaInt16 *iIndex1*,	// index of an element in a row in the array
UtaInt16 *iIndex2*	// index of an element in a column in the // array

);

Parameters

hR64Array
The handle to a data container that contains a two-dimensional array of real numbers.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

Return Value

Returns from the data container the value of a 64-bit real number stored in the element identified by the indices.

Example

```
// Example assumes that two-dimensional array MyReal64Array2 exists
// and that its handle is hMyReal64Array2.
double dMyReal64;
long lRow = 5, lColumn = 6;
dMyReal64 = UtaR64ArrGetAt2(hMyReal64Array2, lRow, lColumn);
// dMyReal64 now has value from element whose coordinates are 5, 6
```

See also

UtaR64ArrSetAt2()

UtaR64ArrSetAt2()

This function updates the value of an array element in a specified data container when the data container contains a two-dimensional array of 64-bit real numbers.

```
void UtaR64ArrSetAt2(
    HUTAR64ARR hR64Array,     // handle to a data container
    UtaInt16 iIndex1,         // index of an element in a row in the array
    UtaInt16 iIndex2,         // index of an element in a column in the
                              // array
    UtaReal64 dValue          // desired value of the element at the
                              // index
    );
```

Parameters

hR64Array
The handle to a data container that contains a two-dimensional array of 64-bit real numbers.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

dValue
A 64-bit real value to be written to the element identified by the indices.

Return Value

(none)

Example

```
// Example assumes that two-dimensional array MyReal64Array2 exists
// and that its handle is hMyReal64Array2.
double dValue = 12.5;
long lRow = 5, lColumn = 2;
UtaR64ArrSetAt2(hMyReal64Array2, lRow, lColumn, dValue);
// Element whose coordinates are 5, 2 now has value of 12.5
```

See also

UtaR64ArrGetAt2()

UtaI32ArrCreate()

This function creates a data container that contains a single-dimensional array of 32-bit integer numbers and returns a handle to the newly created data container.

HUTAI32ARR Utal32ArrCreate(

UtaInt16 *lowerBound*, // lower boundary of array

UtaInt16 *upperBound* // upper boundary of array

);

Parameters

lowerBound
The lower boundary for elements in the array.

upperBound
The upper boundary for elements in the array.

Return Value

Returns a handle to the newly created data container.

Remarks

The size of the array is determined by the difference between *lowerBound* and *upperBound*. For compatibility with C, always specify *lowerBound* as 0.

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term "memory leak" that can cause unstable operation of your test system.

Example

```
long lMyInt32;
HUTAI32ARR hMyInt32Array; // declare variable (object) for array
hMyInt32Array = UtaI32ArrCreate(0, 9); // create array of 10 elements
UtaI32ArrSetAt1(2,4); // write value of 4 to element 2 of array
   ...(do something)
lMyInt32 = UtaI32ArrGetAt1(hMyInt32Array, 5); // value of element 5
   ...(do something)
UtaDataRelease((HUTADATA)hMyInt32Array); // delete data container
```

See also

UtaI32ArrGetAt1()
UtaI32ArrSetAt1()
UtaDataCopy()
UtaDataRelease()

UtaI32ArrGetBuffer()

This function returns a pointer to the start of an array of 32-bit integer numbers in a specified data container.

UtaPtrInt32 UtaI32ArrGetBuffer(

HUTAI32ARR *hData* // handle to a data container

);

Parameters

hData
A handle to a data container that contains an array of 32-bit integer numbers.

Return Value

Returns a pointer to the first element of the array stored in the data container.

Example

```
// Example assumes that hData is handle to existing data container that
// contains an array of 32-bit integer numbers.
UtaPtrInt32 pDataPointer;
pDataPointer = UtaI32ArrGetBuffer(hData);
```

See also

UtaI32ArrGetAt1()
UtaI32ArrSetAt1()

UtaI32ArrGetAt1()

This function returns the value of an element in a single-dimensional array of 32-bit integer numbers in a specified data container.

```
UtaInt32 UtaI32ArrGetAt1(
    HUTAI32ARR hI32Array,      // handle to a data container
    UtaInt16 iIndex1           // index of an element in the array
    );
```

Parameters

hI32Array
The handle to a data container that contains a single-dimensional array of 32-bit integer numbers.

iIndex1
The index of an element whose value is to be returned from the array.

Return Value

Returns a 32-bit integer value from an element in the array.

Remarks

Use function `UtaI32ArrGetBuffer()` to access a pointer to the C array.

Example

```
long lMyInt32;
HUTAI32ARR hMyInt32Array; // declare variable (object) for array
hMyInt32Array = UtaI32ArrCreate(0, 9); // create array of 10 elements
UtaI32ArrSetAt1(2,4); // write value of 4 to element 2 of array
   ...(do something)
lMyInt32 = UtaI32ArrGetAt1(hMyInt32Array, 5); // value of element 5
   ...(do something)
UtaDataRelease((HUTADATA)hMyInt32Array); // delete data container
```

See also

UtaI32GetBuffer()
UtaI32ArrSetAt1()

UtaI32ArrSetAt1()

This function updates the value of an array element in a specified data container when the data container contains a single-dimensional array of 32-bit integer numbers.

void UtaI32ArrSetAt1(

HUTAI32ARR *hI32Array*, // handle to a data container

UtaInt16 *iIndex1*, // index of an element in the array

UtaInt32 *nValue* // desired value of the element at the index

);

Parameters

hI32Array

The handle to a data container that contains a single-dimensional array of 32-bit integer numbers.

iIndex1
The index of an element in an array in the data container.

nValue
A 32-bit integer value to be written to the element at iIndex1 in the array in the data container.

Return Value

(none)

Example

```
long lMyInt32;
HUTAI32ARR hMyInt32Array; // declare variable (object) for array
hMyInt32Array = UtaI32ArrCreate(0, 9); // create array of 10 elements
UtaI32ArrSetAt1(2,4); // write value of 4 to element 2 of array
   ...(do something)
lMyInt32 = UtaI32ArrGetAt1(hMyInt32Array, 5); // value of element 5
   ...(do something)
UtaDataRelease((HUTADATA)hMyInt32Array); // delete data container
```

See also

UtaI32ArrGetAt1()

UtaI32ArrGetAt2()

This function returns the value of an array element in a data container that contains a two-dimensional array of 32-bit integer numbers.

UtaInt32 UtaI32ArrGetAt2(

HUTAI32ARR *hI32Array*,	// handle to an array data container
UtaInt16 *iIndex1*,	// index of an element in a row in the array
UtaInt16 *iIndex2*	// index of an element in a column in the // array

);

Parameters

hI32Array
The handle to a data container that contains a two-dimensional array of 32-bit integer numbers.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

Return Value

Returns a 32-bit integer value from an element in the array.

Example

```
// Example assumes that two-dimensional array MyInt32Array2 exists
// and that its handle is hMyInt32Array2.
long lMyInt32;
long lRow = 5, lColumn = 6;
dMyInt32 = UtaI32ArrGetAt2(hMyInt32Array2, lRow, lColumn);
// dMyInt32 now has value from element whose coordinates are 5, 6
```

See also

UtaI32ArrSetAt2()

UtaI32ArrSetAt2()

This function updates the value of an array element in a specified data container that contains a single-dimensional array of 64-bit real numbers.

```
void UtaI32ArrSetAt2(
    HUTAI32ARR hI32Array,     // handle to a data container
    UtaInt16 iIndex1,         // index of an element in a row in the array
    UtaInt16 iIndex2,         // index of an element in a column in the
                              // array
    UtaInt32 nValue           // desired value of element at the indices
    );
```

Parameters

hI32Array
The handle to a data container that contains a single-dimensional array of 64-bit real numbers.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

nValue
A 32-bit integer value to be written to an element in the array identified by the indices.

Return Value

(none)

Example

```
// Example assumes that two-dimensional array MyInt32Array2 exists
// and that its handle is hMyInt32Array2.
long lValue = 15, lRow = 5, lColumn = 2;
UtaI32ArrSetAt2(hMyInt32Array2, lRow, lColumn, lValue);
// Element whose coordinates are 5, 2 now has value of 15
```

See also

UtaI32ArrGetAt2()

UtaComplexCreate()

This function creates a data container that holds data whose type is complex, sets the values of the real and imaginary components of the complex data, and returns a handle to the newly created data container.

```
HUTACOMPLEX UtaComplexCreate(
    UtaReal64 dReal,        // real component in data container
    UtaReal64 dImag         // imaginary component in data container
);
```

Parameters

dReal
A 64-bit real value for the real component in the new data container.

dImag
A 64-bit real value for the imaginary component in the new data container.

Return Value

Returns a handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term “memory leak” that can cause unstable operation of your test system.

Example

```
double dReal = 5, dImaginary = -6, dTemp1, dTemp2;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(dReal, dImaginary); // create data
   ...(do something)
dTemp1 = UtaComplexGetReal(hMyComplexData); // get real value
dTemp2 = UtaComplexGetImag(hMyComplexData); // get imaginary value
dTemp1 += 3; // add 3 to value of Temp1
dTemp2 += -1; // add -1 to value of Temp2
UtaComplexSetReal(hMyComplexData, dTemp1); // write new real value
UtaComplexSetImag(hMyComplexData, dTemp2); // write new imaginary value
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetReal()
UtaComplexGetImag()
UtaComplexGetValues()
UtaComplexSetReal()
UtaComplexSetImag()
UtaComplexSetValues()
UtaDataCopy()
UtaDataRelease()

UtaComplexGetValues()

This function returns the real and imaginary components of complex data stored in a specified data container.

void UtaComplexGetValues(

HUTACOMPLEX *hComplex*,	// handle to a data container
UtaReal64 **lpdReal*,	// pointer to real component
UtaReal64 **lpdImag*	// pointer to imaginary component

);

Parameters

hComplex
The handle to a data container that contains complex data.

**lpdReal*
A pointer to the 64-bit real value of the real component in the data container. Defaults to NULL.

**lpdImag*
A pointer to the 64-bit real value of the imaginary component in the data container. Defaults to NULL.

Return Value

(none)

Remarks

Returns 64-bit real values to the addresses specified in **lpdReal* and **lpdImag*. To individually return the values of the complex data's real and imaginary components, use `UtaComplexGetReal()` and `UtaComplexGetImag()`.

Example

```
double dReal, dImag;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(4, -8); // create data
   ...(do something)
UtaComplexGetValues(hMyComplexData, &dReal, &dImag); // get values
// dReal now has value of real component & dImag now has value of
// imaginary component of complex data
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetReal()
UtaComplexGetImag()
UtaComplexSetReal()
UtaComplexSetImag()
UtaComplexSetValues()

UtaComplexSetValues()

This function updates the real and imaginary components of complex data stored in a specified data container.

```
void UtaComplexSetValues(
    HUTACOMPLEX hComplex,     // handle to a data container
    UtaReal64 dReal,          // desired value of real component
    UtaReal64 dImag           // desired value of imaginary component
    );
```

Parameters

hComplex
The handle to a data container that contains complex data.

dReal
A 64-bit real value to be written to the real component in the data container.

dImag
A 64-bit real value to be written to the imaginary component in the data container.

Return Value

(none)

Remarks

The 64-bit real values are specified in *dReal* and *dImag*. To individually set the values of the complex data's real and imaginary components, use `UtaComplexSetReal()` and `UtaComplexSetImag()`.

Example

```
double dReal = 5, dImag = -6, dTemp1, dTemp2;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(dReal, dImag); // create data
   ...(do something)
dTemp1 = UtaComplexGetReal(hMyComplexData); // get real value
dTemp2 = UtaComplexGetImag(hMyComplexData); // get imaginary value
dTemp1 += 3; // add 3 to value of Temp1
dTemp2 += -1; // add -1 to value of Temp2
UtaComplexSetValues(hMyComplexData, dTemp1, dTemp2); // write values
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetReal()
UtaComplexGetImag()
UtaComplexGetValues()
UtaComplexSetReal()
UtaComplexSetImag()

UtaComplexGetReal()

This function returns the value of the real component of complex data stored in a specified data container.

UtaReal64 UtaComplexGetReal(

HUTACOMPLEX *hComplex* // handle to a data container

);

Parameters

hComplex
The handle to a data container that contains complex data.

Return Value

Returns the 64-bit real value of the real component in the data container.

Remarks

To simultaneously return the values of the real and imaginary components, use `UtaComplexGetValues()`.

Example

```
double dReal = 5, dImag = -6, dTemp1, dTemp2;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(dReal, dImag); // create data
   ...(do something)
dTemp1 = UtaComplexGetReal(hMyComplexData); // get real value
dTemp2 = UtaComplexGetImag(hMyComplexData); // get imaginary value
dTemp1 += 3; // add 3 to value of dTemp1
dTemp2 += -1; // add -1 to value of dTemp2
UtaComplexSetReal(hMyComplexData, dTemp1); // write new real value
UtaComplexSetImag(hMyComplexData, dTemp2); // write new imaginary value
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetImag()
UtaComplexGetValues()
UtaComplexSetReal()
UtaComplexSetImag()
UtaComplexSetValues()

UtaComplexGetImag()

This function returns the value of the imaginary component of complex data stored in a specified data container.

UtaReal64 UtaComplexGetImag(

HUTACOMPLEX *hComplex* // handle to a data container

);

Parameters

hComplex
The handle to a data container that contains complex data.

Return Value

Returns the 64-bit real value of the imaginary component of complex data stored in the data container.

Remarks

To simultaneously return the values of the real and imaginary components, use `UtaComplexGetValues()`.

Example

```
double dReal = 5, dImag = -6, dTemp1, dTemp2;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(Real, Imaginary); // create data
   ...(do something)
dTemp1 = UtaComplexGetReal(hMyComplexData); // get real value
dTemp2 = UtaComplexGetImag(hMyComplexData); // get imaginary value
dTemp1 += 3; // add 3 to value of dTemp1
dTemp2 += -1; // add -1 to value of dTemp2
UtaComplexSetReal(hMyComplexData, dTemp1); // write new real value
UtaComplexSetImag(hMyComplexData, dTemp2); // write new imaginary value
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetReal()
UtaComplexGetValues()
UtaComplexSetReal()
UtaComplexSetImag()
UtaComplexSetValues()

UtaComplexSetReal()

This function updates the value of the real component of complex data stored in a specified data container.

```
void UtaComplexSetReal(
    HUTACOMPLEX hComplex,     // handle to a data container
    UtaReal64 dReal           // desired value for the real component
);
```

Parameters

hComplex
The handle to a data container that contains complex data.

dReal
A 64-bit real value to be written to the real component of the complex data in the data container.

Return Value

(none)

Remarks

The 64-bit real value of the real component is passed into *dReal*. To simultaneously set the values of the real and imaginary components, use `UtaComplexSetValues()`.

Example

```
double dReal = 5, dImag = -6, dTemp1, dTemp2;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(dReal, dImag); // create data
   ...(do something)
dTemp1 = UtaComplexGetReal(hMyComplexData); // get real value
dTemp2 = UtaComplexGetImag(hMyComplexData); // get imaginary value
dTemp1 += 3; // add 3 to value of dTemp1
dTemp2 += -1; // add -1 to value of dTemp2
UtaComplexSetReal(hMyComplexData, dTemp1); // write new real value
UtaComplexSetImag(hMyComplexData, dTemp2); // write new imaginary value
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetReal()
UtaComplexGetImag()
UtaComplexGetValues()
UtaComplexSetImag()
UtaComplexSetValues()

UtaComplexSetImag()

This function updates the value of the imaginary component of complex data stored in a specified data container.

```
void UtaComplexSetImag(
   HUTACOMPLEX hComplex,    // handle to a data container
   UtaReal64 dImag          // desired value for the imaginary
                            // component
);
```

Parameters

hComplex
The handle to a data container that contains complex data.

dImag
A 64-bit real value to be written to the imaginary component of complex data in the data container.

Return Value

(none)

Remarks

The 64-bit real value of the imaginary component is passed into *dImag*. To simultaneously set the values of the real and imaginary components, use `UtaComplexSetValues()`.

Example

```
double dReal = 5, dImag = -6, dTemp1, dTemp2;
HUTACOMPLEX hMyComplexData; // assign var. (object) for complex data
hMyComplexData = UtaComplexCreate(dReal, dImag); // create data
   ...(do something)
dTemp1 = UtaComplexGetReal(hMyComplexData); // get real value
dTemp2 = UtaComplexGetImag(hMyComplexData); // get imaginary value
dTemp1 += 3; // add 3 to value of dTemp1
dTemp2 += -1; // add -1 to value of dTemp2
UtaComplexSetReal(hMyComplexData, dTemp1); // write new real value
UtaComplexSetImag(hMyComplexData, dTemp2); // write new imaginary value
   ...(do something)
UtaDataRelease((HUTADATA)hMyComplexData); // delete data container
```

See also

UtaComplexGetReal()
UtaComplexGetImag()
UtaComplexGetValues()
UtaComplexSetReal()
UtaComplexSetValues()

UtaPointCreate()

This function creates a new data container containing data whose type is point, sets the values of the point data's X and Y components, and returns a handle to the newly created data container.

```
HUTAPOINT UtaPointCreate(
    UtaReal64 dX,          // desired value for X data
    UtaReal64 dY           // desired value for Y data
    );
```

Parameters

dX

A 64-bit real value for the X data component of point data in the data container.

dY

A 64-bit real value for the Y data component of point data in the data container container.

Return Value

Returns a handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term "memory leak" that can cause unstable operation of your test system.

Example

```
double dX = 5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
dTemp1 = UtaPointGetX(hMyPointData); // get X value
dTemp2 = UtaPointGetY(hMyPointData); // get Y value
dTemp1 += 3; // add 3 to value of X
dTemp2 += 1; // add 1 to value of Y
UtaPointSetX(hMyPointData, dTemp1); // write new X value
UtaPointSetY(hMyPointData, dTemp2); // write new Y value
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetX()
UtaPointGetY()
UtaPointGetValues()
UtaPointSetX()
UtaPointSetY()
UtaPointSetValues()
UtaDataCopy()
UtaDataRelease()

UtaPointGetValues()

This function returns the values of the X and Y components of point data stored in a specifed data container.

void UtaPointGetValues(

HUTAPOINT *hPoint*,	// handle to a data container
UtaReal64 **lpdX*,	// pointer to X data component
UtaReal64 **lpdY*	// pointer to Y data component

);

Parameters

hPoint
The handle to a data container that contains point data.

**lpdX*
A pointer to a 64-bit real value of the X component of the point data in the data container. Defaults to NULL.

**lpdY*
A pointer to a 64-bit real value of the Y component of the point data in the data container. Defaults to NULL.

Return Value

(none)

Remarks

Returns 64-bit real values to the addresses specified in *lpdX* and *lpdY.* To individually return the values of the point data's X and Y components, use `UtaPointGetX()` and `UtaPointGetY()`.

Example

```
double dX = 5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
UtaPointGetValues(hMyPointData, &dTemp1, &dTemp2);
// dTemp1 now has X value & dTemp2 has Y value of point data
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetX()
UtaPointGetY()

UtaPointSetValues()

This function updates the values of the X and Y components of point data stored in a specified data container.

```
void UtaPointSetValues(
    HUTAPOINT hPoint,     // handle to a data container
    UtaReal64 dX,         // desired value for X data component
    UtaReal64 dY          // desired value for Y data component
);
```

Parameters

hPoint
The handle to a data container that contains point data.

dX
A 64-bit real value to be written to the X component of the point data in the data container.

dY
A 64-bit real value to be written to the Y component of the point data in the data container.

Return Value

(none)

Remarks

The 64-bit real values are specified in *dX* and *dY*. To individually set the values of the point data's X and Y components, use `UtaPointSetX()` and `UtaPointSetY()`.

Example

```
double dX =5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
dTemp1 = UtaPointGetX(hMyPointData); // get X value
dTemp2 = UtaPointGetY(hMyPointData); // get Y value
dTemp1 += 3; // add 3 to value of X
dTemp2 += 1; // add 1 to value of Y
UtaPointSetValues(hMyPointData, dTemp1, dTemp2); // write new values
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetX()
UtaPointGetY()
UtaPointGetValues()
UtaPointSetX()
UtaPointSetY()

UtaPointGetX()

This function returns the value of the X data component of point data in a specified data container.

UtaReal64 UtaPointGetX(

HUTAPOINT *hPoint* // handle to a data container

);

Parameters

hPoint
The handle to a data container that contains point data.

Return Value

Returns the 64-bit real value of the X component of the point data in the data container.

Remarks

To simultaneously return the values of the X and Y components, use `UtaPointGetValues()`.

Example

```
double dX = 5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
dTemp1 = UtaPointGetX(hMyPointData); // get X value
dTemp2 = UtaPointGetY(hMyPointData); // get Y value
dTemp1 += 3; // add 3 to value of X
dTemp2 += 1; // add 1 to value of Y
UtaPointSetX(hMyPointData, dTemp1); // write new X value
UtaPointSetY(hMyPointData, dTemp2); // write new Y value
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetY()
UtaPointGetValues()
UtaPointSetX()
UtaPointSetY()
UtaPointSetValues()

UtaPointGetY()

This function returns the value of the Y component of point data stored in a specified data container.

UtaReal64 UtaPointGetY(

HUTAPOINT *hPoint* // handle to a data container

);

Parameters

hPoint
The handle to a data container that contains point data.

Return Value

Returns the 64-bit real value of the Y component of the point data in the data container.

Remarks

To simultaneously return the values of the X and Y components, use `UtaPointGetValues()`.

Example

```
double dX = 5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
dTemp1 = UtaPointGetX(hMyPointData); // get X value
dTemp2 = UtaPointGetY(hMyPointData); // get Y value
dTemp1 += 3; // add 3 to value of X
dTemp2 += 1; // add 1 to value of Y
UtaPointSetX(hMyPointData, dTemp1); // write new X value
UtaPointSetY(hMyPointData, dTemp2); // write new Y value
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetX()
UtaPointGetValues()
UtaPointSetX()
UtaPointSetY()
UtaPointSetValues()

UtaPointSetX()

This function updates the value of the X component of point data stored in a specified data container.

```
void UtaPointSetX(
    HUTAPOINT hPoint,       // handle to a data container
    UtaReal64 dX            // desired value for the X data component
    );
```

Parameters

hPoint
The handle to a data container that contains point data.

dX
A 64-bit real value to be written to the X component of the point data in the data container.

Return Value

(none)

Remarks

To simultaneously set the values of the X and Y components, use `UtaPointSetValues()`.

Example

```
double dX = 5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
dTemp1 = UtaPointGetX(hMyPointData); // get X value
dTemp2 = UtaPointGetY(hMyPointData); // get Y value
dTemp1 += 3; // add 3 to value of X
dTemp2 += 1; // add 1 to value of Y
UtaPointSetX(hMyPointData, dTemp1); // write new X value
UtaPointSetY(hMyPointData, dTemp2); // write new Y value
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetX()
UtaPointGetY()
UtaPointGetValues()
UtaPointSetY()
UtaPointSetValues()

UtaPointSetY()

This function updates the value of the Y component of point data stored in a specified data container.

void UtaPointSetY(

HUTAPOINT *hPoint*,	// handle to a data container
UtaReal64 *dY*	// desired value for the Y data component

);

Parameters

hPoint
The handle to a data container that contains point data.

dY

A 64-bit real value to be written to the Y component of the point data in the data container.

Return Value

(none)

Remarks

To simultaneously set the values of the X and Y components, use `UtaPointSetValues()`.

Example

```
double dX = 5, dY = 6, dTemp1, dTemp2;
HUTAPOINT hMyPointData; // assign variable (object) for point data
hMyPointData = UtaPointCreate(dX, dY); // create data
   ...(do something)
dTemp1 = UtaPointGetX(hMyPointData); // get X value
dTemp2 = UtaPointGetY(hMyPointData); // get Y value
dTemp1 += 3; // add 3 to value of X
dTemp2 += 1; // add 1 to value of Y
UtaPointSetX(hMyPointData, dTemp1); // write new X value
UtaPointSetY(hMyPointData, dTemp2); // write new Y value
   ...(do something)
UtaDataRelease((HUTADATA)hMyPointData); // delete data container
```

See also

UtaPointGetX()
UtaPointGetY()
UtaPointGetValues()
UtaPointSetX()
UtaPointSetValues()

UtaRangeCreate()

This function creates a data container containing data whose type is range, sets initial values for the range data's components, and returns a handle to the newly created data container.

```
HUTARANGE UtaRangeCreate(
    UtaReal64 dStart,       // beginning value for a range
    UtaReal64 dStop,        // desired ending value for a range
    UtaInt16 iNumPoints     // desired # of points between beginning &
                            // ending values
);
```

Parameters

dStart
The starting value or point in a range of values.

dStop
The ending value or point in a range of values.

iNumPoints
The number of incremental values or points in a range of values.

Return Value

Returns a handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term "memory leak" that can cause unstable operation of your test system.

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints; // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()
UtaDataCopy()
UtaDataRelease()

UtaRangeGetValues()

This function retrieves the values of a range stored inside a specified data container.

```
void UtaRangeGetValues(
    HUTARANGE hRange,       // handle to a data container
    UtaReal64 *lpdStart,    // pointer to starting value for a range
    UtaReal64 *lpdStop,     // pointer to ending value for a range
    UtaInt16 *lpiPoints     // pointer to # of points in a range
    );
```

Parameters

hRange
The handle to a data container that contains range data.

**lpdStart*
A pointer to the starting value or point in a range of values. Defaults to NULL.

**lpdStop*
A pointer to the ending value or point in a range of values. Defaults to NULL.

**lpiPoints*
A pointer to the number of incremental values or points in a range of values. Defaults to NULL.

Return Value

(none)

Remarks

If you pass the pointers to 64-bit real variables in **lpdStart* and **lpdStop* and the pointer to an integer variable in **lpiPoints*, the values in the data container are immediately returned to these variables when this function is

called. If you omit these parameters or use them to pass NULL pointers when calling this function, no action is taken on the pointer variables.

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
UtaRangeGetValues(hMyRangeData, &dTemp1, &dTemp2, &lTemp3);
// dTemp1 now has Start value, dTemp2 has Stop value & lTemp3 has
// value of # of points for range data
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeGetCenter()

This function returns the center value for a range stored in a specified data container.

UtaReal64 UtaRangeGetCenter(

HUTARANGE *hRange* // handle to a data container

);

Parameters

hRange
The handle to a data container that contains range data.

Return Value

Returns the 64-bit real value of the center of a range.

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2, dTemp3;
int lPoints = 5;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
dTemp3 = UtaRangeGetCenter(hMyRangeData); // get Center value
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 5; // add 5 to value of Stop
dTemp3 += 2; // add 2 to value of Center
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetCenter(hMyRangeData, dTemp3); // write new Center value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeGetSpan()

This function returns the span value for a range stored in a specified data container.

```
UtaReal64 UtaRangeGetSpan(
    HUTARANGE hRange        // handle to a data container
);
```

Parameters

hRange
The handle to a data container that contains range data.

Return Value

Returns the 64-bit real value of the span of a range.

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp;
int lPoints = 5;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp = UtaRangeGetSpan(hMyRangeData); // get Span value
dTemp += 2; // add 2 to value of Span
UtaRangeSetSpan(hMyRangeData, dTemp); // write new Span value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeGetStart()

This function returns the starting value for a range stored in a specified data container.

UtaReal64 UtaRangeGetStart(

HUTARANGE *hRange* // handle to a data container

);

Parameters

hRange
The handle to a data container that contains range data.

Return Value

Returns the 64-bit real value of the start of a range.

Remarks

To simultaneously return the values of Start and Stop, use `UtaRangeGetValues()`.

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints; // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()

UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeGetStop()

This function returns the ending value for a range stored in a specified data container.

```
UtaReal64 UtaRangeGetStop(
    HUTARANGE hRange          // handle to a data container
);
```

Parameters

hRange
The handle to a data container that contains range data.

Return Value

Returns the 64-bit real value of the end of a range.

Remarks

To simultaneously return the values of Start and Stop, use `UtaRangeGetValues()`.

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints; // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeGetStep()

This function returns the value of the step size for a range stored in a specified data container.

UtaReal64 UtaRangeGetStep(

HUTARANGE *hRange* // handle to a data container

);

Parameters

hRange
The handle to a data container that contains range data.

Return Value

Returns the 64-bit real value of the step size of a range.

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp;
int lPoints = 5;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp = UtaRangeGetStep(hMyRangeData); // get Step value
dTemp += 1; // add 1 to value of Step
UtaRangeSetStep(hMyRangeData, dTemp); // write new Step value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()

UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeGetNumPoints()

This function returns the number of points for a range stored in a specified data container.

```
UtaInt16 UtaRangeGetNumPoints(
    HUTARANGE hRange          // handle to a data container
);
```

Parameters

hRange
The handle to a data container that contains range data.

Return Value

Returns a 16-bit integer value for the number of points in the data container.

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints(hMyRangeData); // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeSetValues()

This function updates the values of a range stored in a specified data container.

```
void UtaRangeSetValues(
    HUTARANGE hRange,          // handle to a data container
    UtaReal64 dStart,          // desired starting value for a range
    UtaReal64 dStop,           // desired ending value for a range
    UtaInt16 iPoints           // desired # of points in a range
    );
```

Parameters

hRange
The handle to a data container that contains range data.

dStart
The starting value or point in a range of values.

dStop
The ending value or point in a range of values.

iPoints
The number of incremental values or points in a range of values.

Return Value

(none)

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dNewStart = 2, dNewStop = 20;
int lPoints = 5, lNewPoints = 10;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
UtaRangeSetValues(hMyRangeData, dNewStart, dNewStop, lNewPoints);
// Values of Start, Stop & # of points now updated in range data
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()

UtaRangeSetCenter()

This function updates the center value for a range stored in a specified data container.

void UtaRangeSetCenter(

HUTARANGE *hRange*, // handle to a data container

UtaReal64 *dCenter* // desired center value for a range

);

Parameters

hRange
The handle to a data container that contains range data.

dCenter
The center value in a range of points.

Return Value

(none)

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

```
Example
double dStart = 1, dStop = 10, dTemp1, dTemp2, dTemp3;
int lPoints = 5;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
dTemp3 = (int)UtaRangeGetCenter(hMyRangeData); // get Center value
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 5; // add 5 to value of Stop
dTemp3 += 2; // add 2 to value of Center
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetCenter(hMyRangeData, dTemp3); // write new Center value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()

UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeSetSpan()

This function updates the span value for a range stored in a specified data container.

```
void UtaRangeSetSpan(
    HUTARANGE hRange,          // handle to a data container
    UtaReal64 dSpan            // desired span value for a range
    );
```

Parameters

hRange
The handle to a data container that contains range data.

dSpan
The span value in a range of points.

Return Value

(none)

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp;
int lPoints = 5;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp = UtaRangeGetSpan(hMyRangeData); // get Span value
dTemp += 2; // add 2 to value of Span
UtaRangeSetSpan(hMyRangeData, dTemp); // write new Span value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeSetStart()

This function updates the beginning value for a range stored in a specified data container.

void UtaRangeSetStart(

HUTARANGE *hRange*, // handle to a data container

UtaReal64 *dStart* // desired beginning value for a range

);

Parameters

hRange
The handle to a data container that contains range data.

dStart
A 64-bit real value to be written to the data container as the new beginning value for the range.

Return Value

(none)

Remarks

To simultaneously set the values of Start and Stop, use `UtaRangeSetValues()`.

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints; // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()

UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeSetStop()

This function updates the ending value for a range stored in a specified data container.

```
void UtaRangeSetStop(
    HUTARANGE hRange,        // handle to a data container
    UtaReal64 dStop          // desired ending value for a range
    );
```

Parameters

hRange
The handle to a data container that contains range data.

dStop
A 64-bit real value to be written to the data container as the new ending value for the range.

Return Value

(none)

Remarks

To simultaneously set the values of Start and Stop, use `UtaRangeSetValues()`.

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints; // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaRangeSetStep()

This function updates the step size value for a range stored in a specified data container.

```
void UtaRangeSetStep(
    HUTARANGE hRange,        // handle to a data container
    UtaReal64 dStep          // desired step size for a range
    );
```

Parameters

hRange
The handle to a data container that contains range data.

dStep
A 64-bit real value to be written to the data container as the new step size for the range.

Return Value

(none)

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp;
int lPoints = 5;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp = UtaRangeGetStep(hMyRangeData); // get Step value
dTemp += 1; // add 1 to value of Step
UtaRangeSetStep(hMyRangeData, dTemp); // write new Step value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetNumPoints()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetValues()

UtaRangeSetNumPoints()

This function updates the number of points for a range stored in a specified data container.

```
void UtaRangeSetNumPoints(
    HUTARANGE hRange,       // handle to a data container
    UtaInt16 iPoints        // desired # of points in a range
    );
```

Parameters

hRange
The handle to a data container that contains range data.

iPoints
A 16-bit integer value written to update the number of points for range data in the data container.

Return Value

(none)

Remarks

There are several different models or ways of viewing range data; see "Which Data Types Does the HP TestCore API Support?" in Chapter 1 for more information.

Example

```
double dStart = 1, dStop = 10, dTemp1, dTemp2;
int lPoints = 5, lTemp3;
HUTARANGE hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(dStart, dStop, lPoints); // create data
   ...(do something)
dTemp1 = UtaRangeGetStart(hMyRangeData); // get Start value
dTemp2 = UtaRangeGetStop(hMyRangeData); // get Stop value
lTemp3 = (int)UtaRangeGetNumPoints; // get # of Points
dTemp1 += 3; // add 3 to value of Start
dTemp2 += 1; // add 1 to value of Stop
lTemp3 += 4; // add 4 to value of Points
UtaRangeSetStart(hMyRangeData, dTemp1); // write new Start value
UtaRangeSetStop(hMyRangeData, dTemp2); // write new Stop value
UtaRangeSetNumPoints(hMyRangeData, lTemp3); // write new Points value
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
```

See also

UtaRangeGetStart()
UtaRangeGetStop()
UtaRangeGetNumPoints()
UtaRangeGetCenter()
UtaRangeGetSpan()
UtaRangeGetStep()
UtaRangeGetValues()
UtaRangeSetStart()
UtaRangeSetStop()
UtaRangeSetCenter()
UtaRangeSetSpan()
UtaRangeSetStep()
UtaRangeSetValues()

UtaWaveformCreate()

This function creates a new data container that contains a waveform and returns a handle to the newly created data container.

HUTAWAVEFORM UtaWaveformCreate(

UtaInt16 *nPoints* // desired # of points in a waveform

);

Parameters

nPoints

A 16-bit integer value that specifies the desired number of points for waveform data being created in the data container.

Return Value

Returns a handle to the newly created data container.

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically. If you do use this function to create data containers, we recommend that you eventually use `UtaDataRelease()` to delete any data containers that you create. Otherwise, the memory used by data containers will not be recovered, which means you will have a long-term “memory leak” that can cause unstable operation of your test system.

See Chapter 1 for an overview of waveform data.

Example

```
HUTAWAVEFORM hMyWaveformData; // declare variable for waveform data
hMyWaveformData = UtaWaveformCreate(10); // create & assign # of points
   ...(more code)
UtaDataRelease((HUTADATA)hMyWaveformData); // delete data container
```

See also

UtaWaveformGetBuffer()
UtaWaveformGetStart()

UtaWaveformGetStop()
UtaWaveformGetNumPoints()
UtaWaveformSetStart()
UtaWaveformSetStop()
UtaWaveformGetAt()
UtaWaveformSetAt()
UtaDataCopy()
UtaDataRelease()

UtaWaveformGetBuffer()

This function returns a pointer to the start of an array in a specified data container that contains waveform data.

```
UtaPtrReal64 UtaWaveformGetBuffer(
    HUTAWAVEFORM hWaveform      // handle to a data container
);
```

Parameters

hWaveform
The handle to a data container that contains an array of 64-bit real numbers that store the values of points for waveform data.

Return Value

Returns a pointer to the first element of the array stored in the data container.

Example

```
// Example assumes that waveform data already exists & its handle is
// hMyWaveformData
UtaPtrReal64 pStartOfArray; // declare variable for pointer to buffer
pStartOfArray = UtaWaveformGetBuffer(hMyWaveformData); // get pointer
```

UtaWaveformGetStart()

This function returns the starting value for a waveform stored in a specified data container.

UtaReal64 UtaWaveformGetStart(

HUTAWAVEFORM *hWaveform* // handle to a data container

);

Parameters

hWaveform
The handle to a data container that contains waveform data.

Return Value

Returns the 64-bit real value of the start of a waveform.

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
double dStart; // declare variable to hold returned value of start
dStart = UtaWaveformGetStart(hMyWaveformData); // get value of start
```

See also

UtaWaveformGetStop()
UtaWaveformGetNumPoints()
UtaWaveformSetStart()
UtaWaveformSetStop()

UtaWaveformGetStop()

This function returns the ending value for a waveform stored in a specified data container.

```
UtaReal64 UtaWaveformGetStop(
    HUTAWAVEFORM hWaveform      // handle to a data container
);
```

Parameters

hWaveform
The handle to a data container that contains waveform data.

Return Value

Returns the 64-bit real value of the end of a waveform.

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
double dStop; // declare variable to hold returned value of stop
dStop = UtaWaveformGetStop(hMyWaveformData);
```

See also

UtaWaveformGetStart()
UtaWaveformGetNumPoints()
UtaWaveformSetStart()
UtaWaveformSetStop()

UtaWaveformGetNumPoints()

This function returns the number of points for a waveform stored in a specified data container.

UtaInt16 UtaWaveformGetNumPoints(

HUTAWAVEFORM *hWaveform* // handle to a data container

);

Parameters

hWaveform
The handle to a data container that contains waveform data.

Return Value

Returns a 16-bit integer value for the number of points in the waveform.

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
int nNumPoints; // declare variable to hold returned # of points
nNumPoints = UtaWaveformGetNumPoints(hMyWaveformData);
```

See also

UtaWaveformGetStart()
UtaWaveformGetStop()
UtaWaveformGetNumPoints()
UtaWaveformSetStart()
UtaWaveformSetStop()

UtaWaveformSetStart()

This function updates the beginning value for a waveform stored in a specified data container.

```
void UtaWaveformSetStart(
    HUTAWAVEFORM hWaveform,     // handle to a data container
    UtaReal64 dStart            // desired beginning value for a
                                // waveform
    );
```

Parameters

hWaveform
The handle to a data container that contains waveform data.

dStart
A 64-bit real value to be written to the data container as the new beginning value for the waveform.

Return Value

(none)

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
double dStart = 5; // variable for value of start
UtaWaveformSetStart(hMyWaveformData, dStart); // set start to 5
```

See also

UtaWaveformGetStart()
UtaWaveformGetStop()
UtaWaveformGetNumPoints()
UtaWaveformSetStop()

UtaWaveformSetStop()

This function updates the ending value for a waveform stored in a specified data container.

```
void UtaWaveformSetStop(
    HUTAWAVEFORM hWaveform,     // handle to a data container
    UtaReal64 dStop             // desired ending value for a
                                // waveform
    );
```

Parameters

hWaveform
The handle to a data container that contains waveform data.

dStop
A 64-bit real value to be written to the data container as the new ending value for the waveform.

Return Value

(none)

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
double dStop = 25; // variable for value of stop
UtaWaveformSetStop(hMyWaveformData, dStop); // set stop to 25
```

See also

UtaWaveformGetStart()
UtaWaveformGetStop()
UtaWaveformGetNumPoints()
UtaWaveformSetStart()

UtaWaveformGetAt()

This function returns the value of a specified element in an array in waveform data stored in a specified data container.

```
UtaReal64 UtaWaveformGetAt(
    HUTAWAVEFORM hWaveform,     // handle to a data container
    UtaInt16 nIndex             // index of an element in the array
    );
```

Parameters

hWaveform
The handle to a data container that contains an array of 64-bit real numbers that store the values of points for waveform data.

nIndex
The index of an element whose value is to be returned from the array.

Return Value

Returns a 64-bit real value from an element in the array.

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
double dValue; // variable to hold returned value of element at index
int nIndex = 5; // variable for index of element in array
dValue = UtaWaveformGetAt(hMyWaveformData, nIndex); // get the value
```

See also

UtaWaveformSetAt()

UtaWaveformSetAt()

This function updates the value of a specified element in an array in waveform data stored in a specified data container.

```
void UtaWaveformSetAt(
    HUTAWAVEFORM hWaveform,     // handle to a data container
    UtaInt16 nIndex,            // index of an element in the array
    UtaReal64 dValue            // desired value of the element at the
                                // index
    );
```

Parameters

hWaveform

The handle to a data container that contains an array of 64-bit real numbers that store the values of points for waveform data.

nIndex

The index of an element in an array in the data container.

dValue

A 64-bit real value to be written to an element at the specified index in the array in the data container.

Return Value

(none)

Remarks

See Chapter 1 for an overview of waveform data.

Example

```
// Example assumes that waveform data already exists in data container
// & its handle is hMyWaveformData
int nIndex = 5; // variable for index of element in array & its value
double dValue = 3; // variable for value of specified element in array
UtaWaveformSetAt(hMyWaveformData, nIndex, dValue); // set the value
```

See also

UtaWaveformGetAt()

UtaInstGetViSession()

This function returns the identifier of VXI*plug&play* ViSession in instrument data stored in a specified data container.

UTAInt32 UtaInstGetViSession(

HUTAINST *hInstrument* // handle to a data container

);

Parameters

hInstrument
The handle to a data container that contains instrument data.

Return Value

Returns a 32-bit integer that is the identifier for a ViSession.

Example

```
void UTADLL ProgramPowerSupply (HUTAPB hParameterBlock)
{
// Action routine that programs an HP 66312 power supply.
// Example assumes that parameter block contains three parameters:
//   Voltage - type Real64
//   Current - type Real64
//   PowerSupply - type Inst
```

```
// Assign miscellaneous variables
HUTAREAL64 hData;
ViStatus ErrorCodes;
HUTAINST hInstrument;

// Get value of voltage from parameter block
hData = UtaPbGetReal64(hParameterBlock, "Voltage");
double dVolt = UtaReal64GetValue(hData);

// Get value of current from parameter block
hData = UtaPbGetReal64(hParameterBlock, "Current");
double dCurr = UtaReal64GetValue(hData);

// Get the ViSession identifier from the parameter block
hInstrument = UtaPbGetInst(hParameterBlock, "PowerSupply");
long lViSession = UtaInstGetViSession(hInstrument);

// Set the voltage & current, and turn on the output
ErrorCodes = hp66312_voltCurrOutp (lViSession, dVolt, dCurr);

...(optional code that checks ErrorCodes for power supply errors)

return;
}
```

See also

UtaPbGetInst()

Functions for Copying & Releasing Data in Data Containers

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaDataCopy()

This function creates a new data container, copies data in a given data container to the newly created data container, and returns a handle to the newly created data container.

HUTADATA UtaDataCopy(

HUTADATA *hData* // handle to a data container

);

Parameters

hData

The handle to an existing data container that is to be duplicated and whose data is to be copied into the duplicate data container.

Return Value

Returns a handle to the newly created, duplicate data container.

Example

```
// Example shows copying range data, but other data types can be copied
HUTADATA hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(1, 10, 5); // create data container
   ...(do something)
HUTADATA hNewRangeData = UtaDataCopy(hMyRangeData); // make a copy
   ...(do something)
UtaDataRelease((HUTADATA)hMyRangeData); // delete data container
UtaDataRelease((HUTADATA)hNewRangeData); // delete data container
```

See also

UtaDataRelease()

UtaDataRelease()

This function frees memory used by data stored in the specified data container.

void UtaDataRelease(

HUTADATA *hData* // handle to a data container

);

Parameters

hData
The handle to a data container whose memory is to be freed for reuse.

Return Value

(none)

Remarks

Because this "generic" function is used to delete all types of data containers, you must cast the data type being released to HP TestCore's generic data type, HUTADATA.

Example

```
// Example shows releasing range data, but other data types can be
// released
HUTADATA hMyRangeData; // assign variable (object) for range data
hMyRangeData = UtaRangeCreate(1, 10, 5); // create data container
   ...(do something)
HUTADATA hNewRangeData = UtaDataCopy(hMyRangeData); // make a copy
   ...(do something)
// delete data containers & free memory. Notice use of cast operator.
UtaDataRelease((HUTADATA)hMyRangeData);
UtaDataRelease((HUTADATA)hNewRangeData);
```

See also

UtaDataCopy()

Functions for Manipulating Switching Paths

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaPathConnect()

This function establishes a switching path based on a switching path stored in a specified switching path object.

```
void UtaPathConnect(
    HUTAPATH hPath,          // handle to a switching path object
    BOOL bWait               // wait/no wait for a change of state
    );
```

Parameters

hPath

The handle to an object that contains switching path data.

bWait

Specifies whether or not this function waits for switching elements in the switching path to change state before returning control to the calling program. TRUE is wait, FALSE is do not wait. Defaults to TRUE.

Return Value

(none)

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

See also

UtaPathDisconnect()

UtaPathDisconnect()

This function disconnects (opens) all the switching elements in a switching path stored in a specified switching path object.

void UtaPathDisconnect(

HUTAPATH *hPath*, // handle to a switching path object

BOOL *bWait* // wait/no wait for change of state

);

Parameters

hPath

The handle to an object that contains switching path data.

bWait

Specifies whether or not this function waits for the switching elements in the switching path to change state before returning control to the calling program. TRUE is wait, FALSE is do not wait. Defaults to TRUE.

Return Value

(none)

Example

```
HUTAPATH hPath;
// Get the parameter specifying the path
hPath = UtaPbGetPath (hParameterBlock, "DcvPathLow");
// Close the path
UtaPathConnect (hPath);
// Take a measurement
// ...(do something)
// Open the Path
UtaPathDisconnect (hPath);
```

See also

UtaPathConnect()

UtaPathWait()

This function is used to wait for a `UtaPathConnect()` or `UtaPathDisconnect()` function to complete when its *bWait* flag is set to FALSE.

```
void UtaPathWait(
    HUTAPATH hPath          // handle to a data container
);
```

Parameters

hPath

The handle to an object that contains switching path data.

Return Value

(none)

Remarks

This function returns control to the calling program as soon as all of the switching elements have finished changing their state.

Example

```
// SIMPLE EXAMPLE
UtaPathConnect (hPath, FALSE);
...(do something else while waiting)
UtaPathWait (hPath);
// Ensures that path will be closed.

// MORE COMPLEX EXAMPLE that lets 3 paths be established at once
// and does other tasks while waiting for all of them to complete.
UtaPathConnect (hPath1, FALSE);
UtaPathConnect (hPath2, FALSE);
UtaPathConnect (hPath3, FALSE);
...(do something else while waiting)
UtaPathWait (hPath1);
UtaPathWait (hPath2);
UtaPathWait (hPath3);
// Ensures that all paths will be closed.
```

See also

UtaPathConnect()
UtaPathDisconnect()

UtaStateCreate()

This function allocates memory and creates a switching state into which switching paths can be merged.

HUTASTATE UtaStateCreate();

Parameters

(none)

Return Value

The handle to a switching state.

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

See also

UtaStateRelease()

UtaStateRelease()

This function deletes a switching state in a specified data container and any memory allocated to it.

void UtaStateRelease(

HUTASTATE *hState* // handle to a data container

);

Parameters

hState

The handle to a data container that contains switching state data that is to be deleted and whose memory is to be freed for reuse.

Return Value

(none)

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching state
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

See also

UtaStateCreate()

UtaStateMergeState()

This function merges a switching state stored in a specified data container with a second switching state stored in another data container.

```
void UtaStateMergeState(
    HUTASTATE hState,          // handle to first data container
    HUTASTATE hStateToMerge    // handle to second data container
    );
```

Parameters

hState
The handle to a data container that contains switching state data.

hStateToMerge
The handle to a second data container whose switching state is to be merged with the first.

Return Value

(none)

Example

```
// Example assumes states State1 & State2 exist & the handles to
// their data containers are hState1 & hState2
UtaStateMergeState(hState1, hState2); // merge State2 with State1
```

UtaStateMergePathState()

This function merges a switching path stored in a specified switching data object into a switching state stored in a data container.

```
void UtaStateMergePathState(
    HUTASTATE hState,    // handle to data container that contains switching
                         // state
    HUTAPATH hPath       // handle to object that contains switching path
);
```

Parameters

hState
The handle to a data container that contains switching state data.

hPath
The handle to an object that contains switching path data.

Return Value

(none)

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

UtaStateUpdate()

This function updates the positions of switching elements in a switching state stored in a data container so it matches the current states of those switching elements in the hardware. In other words, this function reads the hardware and updates the switching state from it.

void UtaStateUpdate(

HUTASTATE *hState* // handle to a data container

);

Parameters

hState

The handle to a data container that contains switching state data.

Return Value

(none)

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its initial, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

UtaStateClear()

This function sets to their default positions all of the switching elements in a switching state stored in a specified data container.

```
void UtaStateClear(
    HUTASTATE hState          // handle to a data container
    );
```

Parameters

hState
The handle to a data container that contains switching state data.

Return Value

(none)

Remarks

When this function updates simple relays to their default state, it opens them.

This function does not affect the hardware; i.e., it clears the switching state without clearing the hardware. Use `UtaStateReset()` to clear the hardware specified in a switching state.

Example

```
// Example assumes switching state State1 exists & the handle to its
// data container is hState1
UtaStateClear(hState1); // clear state but do not reset hardware
```

See also

UtaStateReset()

UtaStateRecall()

This function sets switching elements in hardware to the positions for those switching elements specified in a switching state stored in a specified data container. In other words, this function reads a switching state and then updates the hardware from it.

```
void UtaStateRecall(
    HUTASTATE hState,      // handle to a data container
    BOOL bWait             // wait/no wait for change of state
    );
```

Parameters

hState
The handle to a data container that contains switching state data.

bWait

Specifies whether or not this function waits for the switching elements in the path to close before returning control to the calling program. TRUE is wait, FALSE is do not wait. Defaults to TRUE.

Return Value

(none)

Example

```
// The following example temporarily stores the state of the switching
// hardware, adds to the state of the switching hardware a new path
// previously stored in a parameter block, and subsequently restores
// the switching hardware to its original state. It assumes path data
// already exists in a parameter named NewPath in a parameter block.
HUTASTATE hOriginalState; // variable for handle to switching state
HUTAPATH hPath; // variable for handle to switching path
hOriginalState = UtaStateCreate(); // create empty switching state
hPath = UtaPbGetPath(hParameterBlock, "NewPath"); // get path data
UtaStateMergePathState(hOriginalState, hPath); // define state's scope
UtaStateUpdate(hOriginalState); // store current state of hardware
UtaPathConnect(hPath); // set hardware to path retrieved from NewPath
// Do tasks while new path is in effect
...
...(make a measurement, etc.)
...
// restore the hardware to its original, stored state
UtaStateRecall(hOriginalState);
UtaStateRelease(hOriginalState); // free memory used by state object
```

UtaStateReset()

This function resets to their default positions all of the switching elements in a switching state stored in a specified data container.

```
void UtaStateReset(
    HUTASTATE hState,      // handle to a data container
    BOOL bWait             // wait/no wait for change of state
    );
```

Parameters

hState
The handle to a data container that contains switching state data.

bWait
Specifies whether or not this function waits for the switching elements in the switching path to change state before returning control to the calling program. TRUE is wait, FALSE is do not wait. Defaults to TRUE.

Return Value

(none)

Remarks

When this function updates simple relays to their default state, it opens them.

This function resets the hardware specified in the switching state data. Use `UtaStateClear()` to clear a switching state without resetting the hardware.

Example

```
// Examples assume switching state State1 exists & the handle to its
// data container is hState1

// Reset hardware defined in state hState1. Wait for hardware to change
// state before continuing.
UtaStateReset(hState1);

// Reset hardware defined in state hState1. Do not wait for hardware
// to change state before continuing.
UtaStateReset(hState1, FALSE);
```

See also

UtaStateClear()

UtaStateWait()

This function waits for the switching elements to complete their change of state before returning control to the calling program.

```
void UtaStateWait(
    HUTASTATE hState          // handle to a switching state
);
```

Parameters

hState
The handle to a data container that contains switching state data.

Return Value

(none)

Remarks

Use this function when FALSE is passed as the *bWait* parameter to the `UtaStateRecall()` and `UtaStateReset()` functions.

Example

```
// Example assumes switching state data is stored in State1, whose
// handle is hState1
UtaStateRecall(hState1); // set hardare to state defined by State1
UtaStateWait(hState1); // wait for switching elements to change state
```

See also

UtaStateRecall()
UtaStateReset()

Functions for Waiting (timer control)

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaTimerCreate()

This function creates a data container that contains a timer and returns a handle to the newly created data container. Timers usually are used to ensure that a DUT or source has settled before continuing.

```
HUTATIMER UtaTimerCreate();
```

Parameters

(none)

Return Value

Returns a handle to the newly created data container.

Remarks

If you use function this function to create a timer, you must eventually use `UtaTimerRelease()` to release the memory it uses or you will have a memory leak.

Example

```
// Example allows 100 msec. of settling time for a power supply
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
// wait for power supply to settle
UtaTimerSet(hMyTimer, 100000); // set delay in usec. & start counting
UtaTimerWait(hMyTimer); // wait for timer to time out
// power supply has settled now, okay to continue
...(do something)
UtaTimerRelease(hMyTimer); // delete data container & free memory
```

See also

UtaTimerGetTimeLeft()
UtaTimerReset()
UtaTimerSet()
UtaTimerWait()
UtaTimerRelease()

UtaTimerGetTimeLeft()

This function returns the amount of time remaining on a timer stored in a specified data container.

UTAUSECS UtaTimerGetTimeLeft(

HUTATIMER *hTimer* // handle to a data container

);

Parameters

hTimer
The handle to a data container that contains a timer.

Return Value

Returns, in microseconds, the amount of time remaining on a timer. Or, returns 0 if the timer has gone off; i.e., "timed out."

Remarks

Use function `UtaTimerCreate()` to create data containers that contain timers. Use `UtaTimerGetElapsedTime()` to return the amount of time elapsed on a timer.

Example

```
// Example allows at least 100 msec. of settling time for a power
// supply while simultaneously doing another task. The timer is
// programmed for longer than needed to set up the power supply,
// and the UtaTimerGetTimeLeft() function is used to make sure
// the minimum necessary interval has passed before continuing.
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
// wait for power supply to settle
UtaTimerSet(hMyTimer, 1000000); // set delay in usec. & start counting
...(do something, such as set up an instrument)
while (UtaTimerGetTimeLeft(hMyTimer) > 100000)
  UtaTimerWait(hMyTimer); // wait for timer to time out
// power supply has settled now, okay to continue
UtaTimerRelease(hMyTimer); // delete data container & free memory
```

See also

UtaTimerCreate()
UtaTimerGetElapsedTime()
UtaTimerSet()
UtaTimerWait()
UtaTimerRelease()

UtaTimerWait()

This function waits the amount of time specified in a timer stored in a specified data container.

void UtaTimerWait(

HUTATIMER *hTimer* // handle to a data container

);

Parameters

hTimer
The handler to a data container that contains a timer.

Return Value

(none)

Remarks

Use function `UtaTimerCreate()` to create data containers that contain timers.

Example

```
// Example allows 100 msec. of settling time for a power supply
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
// wait for power supply to settle
UtaTimerSet(hMyTimer, 100000); // set delay in usec. & start counting
UtaTimerWait(hMyTimer); // wait for timer to time out
// power supply has settled now, okay to continue
UtaTimerRelease(hMyTimer); // delete data container & free memory
```

See also

UtaTimerCreate()
UtaTimerGetTimeLeft()
UtaTimerSet()
UtaTimerRelease()

UtaTimerSet()

This function sets the delay for a timer stored in a data container and starts the timer counting down.

void UtaTimerSet(

HUTATIMER *hTimer*, // handle to a data container

UTAUSECS *uSecsDelay* // # of microseconds delay in timer

);

Parameters

hTimer
The handle to a data container that contains a timer.

uSecsDelay
The delay in microseconds before the timer times out.

Return Value

(none)

Remarks

Use function `UtaTimerCreate()` to create timers. Use function `UtaTimerWait()` to wait until the timer has finished counting down to zero.

Example

```
// Example allows 100 msec. of settling time for a power supply
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
// wait for power supply to settle
UtaTimerSet(hMyTimer, 100000); // set delay in usec. & start counting
UtaTimerWait(hMyTimer); // wait for timer to time out
// power supply has settled now, okay to continue
UtaTimerRelease(hMyTimer); // delete data container & free memory
```

See also

UtaTimerCreate()
UtaTimerGetTimeLeft()
UtaTimerWait()
UtaTimerRelease()
UtaTimerReset()

UtaTimerRelease()

This function releases memory associated with a timer created by function `UtaTimerCreate()`.

```
void UtaTimerRelease(
    HUTATIMER hTimer            // handle to a data container
);
```

Parameters

hTimer
The handle to a data container containing a timer whose memory is to be freed for reuse.

Return Value

(none)

Remarks

If you use function `UtaTimerCreate()` to create a timer, you must eventually use this function to release the memory it uses or you will have a memory leak.

Example

```
// Example allows 100 msec. of settling time for a power supply
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
// wait for power supply to settle
UtaTimerSet(hMyTimer, 100000); // set delay in usec. & start counting
UtaTimerWait(hMyTimer); // wait for timer to time out
// power supply has settled now, okay to continue
UtaTimerRelease(hMyTimer); // delete data container & free memory
```

See also

UtaTimerCreate()
UtaTimerGetTimeLeft()
UtaTimerSet()
UtaTimerWait()

UtaTimerGetElapsedTime()

This function returns the amount of time elapsed on a timer stored in a specified data container.

UTAUSECS UtaTimerGetElapsedTime(

HUTATIMER *hTimer* // handle to a data container

);

Parameters

hTimer
The handle to a data container that contains a timer.

Return Value

Returns, in microseconds, the amount of time elapsed on a timer. Or, returns zero if the timer has gone off; i.e., "timed out."

Remarks

The elapsed time is the interval between the last `UtaTimerSet()` or `UtaTimerReset()` function and when this function is invoked.

Use function `UtaTimerCreate()` to create data containers that contain timers. Use `UtaTimerGetTimeLeft()` to return the amount of time remaining on a timer.

Example

```
// Example allows at least 100 msec. of settling time for a power
// supply while simultaneously doing another task. The timer is
// programmed for longer than needed to set up the power supply,
// and the UtaTimerGetTimeLeft() function is used to make sure
// the minimum necessary interval has passed before continuing.
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
// wait for power supply to settle
UtaTimerSet(hMyTimer, 1000000); // set delay in usec. & start counting
...(do something, such as set up an instrument)
while (UtaTimerGetElapsedTime(hMyTimer) <= 100000)
   UtaTimerWait(hMyTimer); // wait for timer to time out
// power supply has settled now, okay to continue
UtaTimerRelease(hMyTimer); // delete data container & free memory
```

See also

UtaTimerCreate()
UtaTimerGetTimeLeft()
UtaTimerSet()
UtaTimerWait()
UtaTimerRelease()

UtaTimerReset()

This function resets to zero the elapsed time for a timer stored in a data container and starts the timer counting down.

void UtaTimerSet(

HUTATIMER *hTimer* // handle to a data container

);

Parameters

hTimer
The handle to a data container that contains a timer.

Return Value

(none)

Remarks

Use function `UtaTimerCreate()` to create timers.

Example

```
HUTATIMER hMyTimer; // declare variable (object) to hold a timer
hMyTimer = UtaTimerCreate(); // create a timer
...(code that sets up the power supply)
UtaTimerSet(hMyTimer, 100000); // set delay in usec. & start counting
UtaTimerWait(hMyTimer); // wait for timer to time out
...(something happens that makes it desirable to reset the timer)
UtaTimerReset(hMyTimer); // reset the timer
UtaTimerWait(hMyTimer); // wait for timer to time out
...(do something)
UtaTimerRelease(hMyTimer ); // delete data container & free memory
```

See also

UtaTimerCreate()
UtaTimerSet()
UtaTimerGetElapsedTime()
UtaTimerGetTimeLeft()
UtaTimerWait()
UtaTimerRelease()

Functions for Interacting with Arrays

Unlike other API functions that let you manipulate arrays that contain specific types of data—e.g., `UtaI32ArrSetAt1()` for arrays of 32-bit integers or `UtaR64ArrCreate()` for arrays of 64-bit real numbers—the functions listed in this section let you interact with all types of arrays.

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaArrayGetSize()

This function returns a value that indicates the size (number of elements) of an array stored in a data container.

```
UtaInt16 UtaArrayGetSize(
    HUTADATA hArray              // handle to a data container
);
```

Parameters

hArray
The handle to a data container that contains an array.

Return Value

Returns the size (number of elements) of the array.

Remarks

Use the cast operator as necessary with data whose type is HUTADATA, which is the "generic" handle to an HP TestCore data type.

Example

```
HUTAI32ARR hArray; // assign variable (object) for array data
long lSize;
char chMessage[40];
hArray = UtaI32ArrCreate(0,9); // create an array with 10 elements
// Note use of cast operator in the line below
lSize = UtaArrayGetSize((HUTADATA)hArray); // return size of the array
sprintf(chMessage, "Size of array = %ld", lSize);
AfxMessageBox(chMessage, MB_OK); // display the size of the array
// delete data container & free memory. Note use of cast operator.
UtaDataRelease((HUTADATA)hArray);
```

See also

UtaArrayGetNumDimensions()
UtaArrayGetLowerBound()
UtaArrayGetUpperBound()

UtaArrayGetNumDimensions()

This function returns a value that indicates how many dimensions an array stored in a data container has.

UtaInt16 UtaArrayGetNumDimensions(

HUTADATA *hArray* // handle to a data container

);

Parameters

hArray
The handle to a data container that contains an array.

Return Value

Returns the number of dimensions in an array; i.e., 1 if the array is single-dimensioned, or greater than 1 if the array is multi-dimensional.

Remarks

Use the cast operator as necessary with data whose type is HUTADATA, which is the "generic" handle to an HP TestCore data type.

Example

```
HUTAI32ARR hArray; // assign variable (object) for array data
long lDimensions;
char chMessage[40];
hArray = UtaI32ArrCreate(0,9); // create a single-dimensional array
// Return # of dimensions of the array. Note use of the cast operator.
lDimensions = UtaArrayGetNumDimensions((HUTADATA)hArray);
sprintf(chMessage, "Num. of dimensions in array = %ld", lDimensions);
AfxMessageBox(chMessage, MB_OK); // display array's # of dimensions
// delete data container & free memory. Note use of cast operator.
UtaDataRelease((HUTADATA)hArray);
```

See also

UtaArrayGetSize()
UtaArrayGetLowerBound()
UtaArrayGetUpperBound()

UtaArrayGetLowerBound()

This function returns a value that indicates the lower boundary for an index into the elements of an array stored in a data container.

UtaInt16 UtaArrayGetLowerBound(

HUTADATA *hArray*, // handle to a data container

UtaInt16 *iDimension* // dimension whose boundary is returned

);

Parameters

hArray
The handle to a data container that contains an array.

iDimension

The dimension whose boundary value is to be returned. Defaults to zero, which is a single-dimensioned array. Enter a non-zero value to specify a dimension in a multi-dimensional array.

Return Value

Returns the value of the lower boundary of an array.

Remarks

Use the cast operator as necessary with data whose type is HUTADATA, which is the "generic" handle to an HP TestCore data type.

Example

```
HUTAI32ARR hArray; // assign variable (object) for array data
long lLowerBoundary;
char chMessage[40];
hArray = UtaI32ArrCreate(2,9); // create a single-dimensional array
// Return lower boundary of array. Notice use of cast operator.
lLowerBoundary = UtaArrayGetLowerBound((HUTADATA)hArray);
sprintf(chMessage, "Lower boundary in array = %ld", lLowerBoundary);
AfxMessageBox(chMessage, MB_OK); // display # of dimensions in array
// delete data container & free memory. Notice use of cast operator.
UtaDataRelease((HUTADATA)hArray);
```

See also

UtaArrayGetUpperBound()

UtaArrayGetUpperBound()

This function returns a value that indicates the upper boundary for an index into the elements of an array stored in a data container.

```
UtaInt16 UtaGetUpperBound(
    HUTADATA hArray,      // handle to a data container
    UtaInt16 iDimension   // dimension whose boundary is returned
    );
```

Parameters

hArray
The handle to a data container that contains an array.

iDimension
The dimension whose boundary value is to be returned. Defaults to zero, which is a single-dimensioned array. Enter a non-zero value to specify a dimension in a multi-dimensional array.

Return Value

Returns the value of the upper boundary of an array.

Remarks

Use the cast operator as necessary with data whose type is HUTADATA, which is the "generic" handle to an HP TestCore data type.

Example

```
HUTAI32ARR hArray; // assign variable (object) for array data
long lUpperBoundary;
char chMessage[40];
hArray = UtaI32ArrCreate(2,9); // create a single-dimensional array
// Return upper boundary of array. Note use of cast operator.
lUpperBoundary = UtaArrayGetUpperBound((HUTADATA)hArray);
sprintf(chMessage, "Upper boundary in array = %ld", lUpperBoundary);
AfxMessageBox(chMessage, MB_OK); // display # of dimensions in array
// delete data container & free memory. Note use of cast operator.
UtaDataRelease((HUTADATA)hArray);
```

See also

UtaArrayGetLowerBound()

UtaArrayGetAt1()

This function returns a handle to an element in a single-dimensional array stored in a data container.

```
HUTADATA UtaArrayGetAt1(
    HUTADATA hArray,        // handle to a data container
    UtaInt16 iIndex1        // index of element in array
    );
```

Parameters

hArray
The handle to a data container that contains a single-dimensioned array.

iIndex1
The index of an element in an array.

Return Value

Returns a handle to an element in an array.

Remarks

Use the cast operator as necessary with data whose type is HUTADATA, which is the "generic" handle to an HP TestCore data type.

Example

```
HUTAI32ARR hArray;
HUTADATA hMyData;
int nCounter;
long lMyLong;
char chMessage[40];
hArray = UtaI32ArrCreate(0,9); // create a single-dimensional array
for (nCounter = 0; nCounter < 10; nCounter++)
  UtaI32ArrSetAt1(hArray, nCounter, nCounter); // set values in array
// Note use of cast operators below.
hMyData = UtaArrayGetAt1((HUTADATA)hArray, 5);
lMyLong = UtaInt32GetValue((HUTAINT32)hMyData);
sprintf(chMessage, "Value of element 5 = %ld", lMyLong);
AfxMessageBox(chMessage, MB_OK); // display the value of the element
```

See also

UtaArrayGetAt2()

UtaArrayGetAt2()

This function returns a handle to an element in a two-dimensional array stored in a data container.

HUTADATA UtaArrayGetAt2(

HUTADATA *hData*, // handle to a data container

UtaInt16 *iIndex1*, // index of element in row of array

UtaInt16 *iIndex2* // index of element in column of array

);

Parameters

hArray
The handle to a data container that contains a two-dimensional array.

Return Value

Returns the handle to an element in an array.

Remarks

Use the cast operator as necessary with data whose type is HUTADATA, which is the "generic" handle to an HP TestCore data type.

Example

```
// Example assumes a two-dimensional array of 32-bit integers exists,
// and that its handle is hArray.
HUTADATA hMyData;
long lMyLong;
char chMessage[40];
// Note use of cast operators below.
hMyData = UtaArrayGetAt2((HUTADATA)hArray, 5, 6);
lMyLong = UtaInt32GetValue((HUTAINT32)hMyData);
sprintf(chMessage, "Value of element 5,6 = %ld", lMyLong);
AfxMessageBox(chMessage, MB_OK); // display the value of the element
```

See also

UtaArrayGetAt1()

UtaPtArrGetAt1()

This function returns the handle to a data container that contains point data stored in an element in a one-dimensional array whose handle is specified.

HUTAPOINT UtaPtArrGetAt1(

HUTAPTARR *hPtArray*,	// handle to an array
UtaInt16 *iIndex1*,	// index of element in the array
UtaReal64 **lpdX*,	// pointer to X value of point in array
UtaReal64 **lpdY*	// pointer to Y value of point in array

);

Parameters

hPtArray
The handle to an array of data whose type is point.

iIndex1
The index of an element in the array.

**lpdX*
A pointer to a 64-bit real value of the X component of point data in the array. Defaults to NULL.

**lpdY*
A pointer to a 64-bit real value of the Y component of point data in the array. Defaults to NULL.

Return Value

The handle to a data container containing point data in an element in the array.

Remarks

If desired, you can directly access the values of the point data in the data container. If you pass the pointers to 64-bit real variables in **lpdX* and **lpdY,* the X and Y values in the data container are immediately returned to those variables when this function is called. If you omit the **lpdX* or **lpdY* parameters or use them to pass a NULL pointer when calling this function, no action is taken on **lpdX* or **lpdY.*

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the values is useful when speed and simplicity are most important.

Example

```
// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
// Example assumes the Action Definition Editor was used to define an
// action whose parameter block has a parameter named "MyParm" that
// contains a one-dimensional array of point data.
HUTAPTARR hMyData; // handle to array of point data
// get handle to data container that contains array of point data
hMyData = UtaPbGetPointArray(hMyParmBlock, "MyParm");
HUTAPOINT hMyPointData; // declare variable for handle to point data
int lIndex = 0; // Specify which array element to retrieve values from
double dXValue, dYValue;
// Get a handle & values from an element in the array
hMyPointData = UtaPtArrGetAt1(hMyData, lIndex, &dXValue, &dYValue);
char chMessage[40];
sprintf(chMessage, "Value of X = %f and Y = %f", dXValue, dYValue);
AfxMessageBox(chMessage, MB_OK); // display the X & Y values

// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes the Action Definition Editor was used to define an
// action whose parameter block has a parameter named "MyParm" that
// contains a one-dimensional array of point data.
HUTAPTARR hMyData; // handle to array of point data
// get handle to data container that contains array of point data
hMyData = UtaPbGetPointArray(hMyParmBlock, "MyParm");
HUTAPOINT hMyPointData; // declare variable for handle to point data
int lIndex = 0; // Specify which array element to retrieve values from
double dXValue, dYValue;
// Get the handle to an element in the array
hMyPointData = UtaPtArrGetAt1(hMyData, lIndex);
// Use the handle to retrieve the X & Y values
dXValue = UtaPointGetX(hMyPointData);
dYValue = UtaPointGetY(hMyPointData);
char chMessage[40];
sprintf(chMessage, "Value of X = %f and Y = %f", dXValue, dYValue);
AfxMessageBox(chMessage, MB_OK); // display the X & Y values
```

See also

UtaPtArrSetAt1()

UtaPtArrGetAt2()

This function returns the handle to a data container that contains point data stored in an element in a two-dimensional array whose handle is specified.

```
HUTAPOINT UtaPtArrGetAt2(
    HUTAPTARR hPtArray,     // handle to an array
    UtaInt16 iIndex1,       // index of element in a row in the array
    UtaInt16 iIndex2,       // index of element in a column in the array
    UtaReal64 *lpdX,        // pointer to X value of point in array
    UtaReal64 *lpdY         // pointer to Y value of point in array
    );
```

Parameters

hPtArray
The handle to an array of data whose type is point.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

**lpdX*
A pointer to a 64-bit real value of the X component of point data in the array. Defaults to NULL.

**lpdY*
A pointer to a 64-bit real value of the Y component of point data in the array. Defaults to NULL.

Return Value

The handle to a data container containing point data in an element in the array.

Remarks

If desired, you can directly access the values of the point data in the data container. If you pass the pointers to 64-bit real variables in **lpdX* and **lpdY*, the X and Y values in the data container are immediately returned to those variables when this function is called. If you omit the **lpdX* or **lpdY* parameters or use them to pass a NULL pointer when calling this function, no action is taken on **lpdX* or **lpdY*.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the values is useful when speed and simplicity are most important.

Example

```
// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
// Example assumes an action exists whose parameter block has a
// parameter named "MyParm" that contains a two-dimensional array of
// point data.
HUTAPTARR hMyData; // handle to array of point data
// get handle to data container that contains array of point data
hMyData = UtaPbGetPointArray(hMyParmBlock, "MyParm");
HUTAPOINT hMyPointData; // declare variable for handle to point data
int lRow = 0, lColumn = 0; // Specify element to retrieve values from
double dXVal, dYVal;
// Get a handle & values from an element in the array
hMyPointData = UtaPtArrGetAt2(hMyData, lRow, lColumn, &dXVal, &dYVal);
char chMessage[40];
sprintf(chMessage, "Value of X = %f and Y = %f", dXVal, dYVal);
AfxMessageBox(chMessage, MB_OK); // display the X & Y values
```

See also

UtaPtArrSetAt2()

UtaPtArrSetAt1()

This function updates the X and Y values of point data stored in a data container in an element in a one-dimensional array whose handle is specified.

```
void UtaPtArrSetAt1(
    HUTAPTARR hPtArray,     // handle to an array
    UtaInt16 iIndex1,       // index of an element in the array
    HUTAPOINT hValue        // handle to point data to be written to
                            // element
);
```

Parameters

hPtArray
The handle to an array of points.

iIndex1
The index of an element in the array.

hValue
The handle to point data whose values are to be written to the specified element in the array.

Return Value

(none)

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically.

Example

```
// Example assumes the Action Definition Editor was used to define an
// action whose parameter block has a parameter named "MyParm" that
// contains a one-dimensional array of point data.
int nIndex = 2; // Specify which element in array
HUTAPTARR hMyData; // handle to array of point data
// get handle to data container that contains array of point data
hMyData = UtaPbGetPointArray(hMyParmBlock, "MyParm");
HUTAPOINT hMyPointData; // declare variable for handle to point data
hMyPointData = UtaPointCreate(5,25); // create point & set X, Y values
UtaPtArrSetAt1(hMyData, lIndex, hMyPointData); // write point data
// Get a handle & values from an element in the array
double dXValue, dYValue;
hMyPointData = UtaPtArrGetAt1(hMyData, nIndex, &dXValue, &dYValue);
char chMessage[40];
sprintf(chMessage, "Value of X = %f and Y = %f", dXValue, dYValue);
AfxMessageBox(chMessage, MB_OK); // display the X & Y values
```

See also

UtaPtArrGetAt1()

UtaPtArrSetAt2()

This function updates the X and Y values of point data stored in a data container in an element in a two-dimensional array whose handle is specified.

void UtaPtArrSetAt2(

HUTAPTARR *hPtArray*,	// handle to an array
UtaInt16 *iIndex1*,	// index of element in a row in the array
UtaInt16 *iIndex2*,	// index of element in a column in the array
HUTAPOINT *hValue*	// handle to point data to be written to element

);

Parameters

hPtArray
The handle to an array of points.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

hValue
The handle to point data whose values are to be written to the specified element in the array.

Return Value

(none)

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically.

Example

```
// Example assumes that an action’s parameter block has a parameter
// named "MyParm" that contains a two-dimensional array of point data.
int nRow = 1, nColumn = 2; // Specify an element in the array
HUTAPTARR hMyData; // handle to array of point data
// get handle to data container that contains array of point data
hMyData = UtaPbGetPointArray(hMyParmBlock, "MyParm");
HUTAPOINT hMyPointData; // declare variable for handle to point data
hMyPointData = UtaPointCreate(5,25); // create point & set X, Y values
UtaPtArrSetAt2(hMyData, nRow, nColumn, hMyPointData); // write data
// Get a handle & values from an element in the array
double dXVal, dYVal;
hMyPointData = UtaPtArrGetAt2(hMyData, nRow, nColumn, &dXVal, &dYVal);
char chMessage[40];
sprintf(chMessage, "Value of X = %f and Y = %f", dXVal, dYVal);
AfxMessageBox(chMessage, MB_OK); // display the X & Y values
```

See also

UtaPtArrGetAt2()

UtaRngArrGetAt1()

This function returns the handle to a data container that contains range data stored in an element in a one-dimensional array whose handle is specified.

```
HUTARANGE UtaRngArrGetAt1(
    HUTARNGARR hRngArray,     // handle to an array
    UtaInt16 iIndex1,         // index of an element in the array
    UtaReal64 *lpdStart,      // pointer to the beginning value for a
                              // range
    UtaReal64 *lpdStop,       // pointer to the ending value for a range
    UtaInt16  *lpiPoints      // pointer to the # of points in a range
    );
```

Parameters

hRngArray
The handle to an array of data whose type is range.

iIndex1
The index of an element in the array.

**lpdStart*
A pointer to a 64-bit real value of the start of the range. Defaults to NULL.

**lpdStop*
A pointer to a 64-bit real value of the end of the range. Defaults to NULL.

**lpiPoints*
A pointer to a 16-bit integer value of the number of points in the range. Defaults to NULL.

Return Value

The handle to a data container containing range data in an element in the array.

Remarks

If desired, you can directly access the values of the range data in the data container. If you pass the pointers to 64-bit real variables in **lpdStart* and **lpdStop* and the pointer to a 16-bit integer variable in **lpiPoints*, the beginning, ending, and point values in the data container are immediately returned to those variables when this function is called. If you omit the **lpdStart*, **lpdStop*, or **lpiPoints* parameters or use them to pass a NULL pointer when calling this function, no action is taken on **lpdStart*, **lpdStop* or **lpiPoints*.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the values is useful when speed and simplicity are most important.

Example

```
// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
// Example assumes the Action Definition Editor was used to define an
// action whose parameter block has a parameter named "MyParm" that
// contains a one-dimensional array of range data.
HUTARNGARR hMyData; // handle to array of range data
// get handle to data container that contains array of range data
hMyData = UtaPbGetRangeArray(hMyParmBlock, "MyParm");
HUTARANGE hMyRangeData; // declare variable for handle to range data
int nIndex = 0; // Specify which array element to retrieve values from
double dStart, dStop;
int nPoints;
// Get a handle & values from an element in the array
hMyRangeData = UtaRngArrGetAt1(
   hMyData,
   nIndex,
   &dStart,
   &dStop,
   &nPoints
   );
char chMessage[40];
sprintf(chMessage, "Start = %f, Stop = %f, Points = %d", dStart, dStop,
   nPoints);
AfxMessageBox(chMessage, MB_OK); // display the range values
```

```
// EXAMPLE OF ACCESSING DATA VIA A HANDLE
// Example assumes the Action Definition Editor was used to define an
// action whose parameter block has a parameter named "MyParm" that
// contains a one-dimensional array of range data.
HUTARNGARR hMyData; // handle to array of range data
// get handle to data container that contains array of range data
hMyData = UtaPbGetRangeArray(hMyParmBlock, "MyParm");
HUTARANGE hMyRangeData; // declare variable for handle to range data
int nIndex = 0; // Specify which array element to retrieve values from
double dStart, dStop;
int nPoints;
// Get the handle to an element in the array
hMyRangeData = UtaRngArrGetAt1(hMyData, nIndex);
// Retrieve the values from the handle
dStart = UtaRangeGetStart(hMyRangeData);
dStop = UtaRangeGetStop(hMyRangeData);
nPoints = UtaRangeGetNumPoints(hMyRangeData);
char chMessage[40];
sprintf(chMessage, "Start = %f, Stop = %f, Points = %d", dStart, dStop,
  nPoints);
AfxMessageBox(chMessage, MB_OK); // display the range values
```

See also

UtaRngArrSetAt1()

UtaRngArrGetAt2()

This function returns the handle to a data container that contains point data stored in an element in a two-dimensional array whose handle is specified.

HUTARANGE UtaRngArrGetAt2(

HUTARNGARR *hRngArray*,	// handle to an array
UtaInt16 *iIndex1*,	// index of an element in a row in the // array
UtaInt16 *iIndex2*,	// index of an element in a column in the // array
UtaReal64 **lpdStart*,	// pointer to beginning value for a range

```
    UtaReal64 *lpdStop,          // pointer to ending value for a range
    UtaInt16  *lpiPoints         // pointer to # of points in range data
    );
```

Parameters

hRngArray
The handle to an array of data whose type is range.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

**lpdStart*
A pointer to a 64-bit real value of the start of the range. Defaults to NULL.

**lpdStop*
A pointer to a 64-bit real value of the end of the range. Defaults to NULL.

**lpiPoints*
A pointer to a 16-bit integer value of the number of points in the range. Defaults to NULL.

Return Value

The handle to a data container containing range data in an element in the array.

Remarks

If desired, you can directly access the values of the range data in the data container. If you pass the pointers to 64-bit real variables in **lpdStart* and **lpdStop* and the pointer to a 16-bit integer variable in **lpiPoints*, the beginning, ending, and point values in the data container are immediately returned to those variables when this function is called. If you omit the **lpdStart*, **lpdStop*, or **lpiPoints* parameters or use them to pass a NULL pointer when calling this function, no action is taken on **lpdStart*, **lpdStop* or **lpiPoints*.

Retrieving the handle can be useful if you expect to use it for additional data manipulations, while directly returning the values is useful when speed and simplicity are most important.

Example

```
// EXAMPLE OF ACCESSING DATA DIRECTLY VIA POINTERS
// Example assumes that an action exists whose parameter block has a
// parameter named "MyParm" that contains a two-dimensional array of
// range data.
HUTARNGARR hMyData; // handle to array of range data
// get handle to data container that contains array of range data
hMyData = UtaPbGetRangeArray(hMyParmBlock, "MyParm");
HUTARANGE hMyRangeData; // declare variable for handle to range data
int nRow = 0, nColumn = 1; // Specify element to retrieve values from
double dStart, dStop;
int nPoints;
// Get a handle & values from an element in the array
hMyRangeData = UtaRngArrGetAt2(
   hMyData,
   nRow,
   nColumn,
   &dStart,
   &dStop,
   &nPoints
   );
char chMessage[40];
sprintf(chMessage, "Start = %f, Stop = %f, Points = %d", dStart, dStop,
   nPoints);
AfxMessageBox(chMessage, MB_OK); // display the range values
```

See also

UtaRngArrSetAt2()

UtaRngArrSetAt1()

This function updates the values of range data stored in a data container in an element in a one-dimensional array whose handle is specified.

```
void UtaRngArrSetAt1(
    HUTARNGARR hRngArray,     // handle to an array
    UtaInt16 iIndex1,         // index of an element in the array
    HUTARANGE hValue          // handle to range data to be written to
                              // element
    );
```

Parameters

hRngArray
The handle to an array of data whose type is range.

iIndex1
The index of an element in the array.

hValue
The handle to range data whose values are to be written to the specified element in the array.

Return Value

(none)

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically.

Example

```
// Example assumes the Action Definition Editor was used to define an
// action whose parameter block has a parameter named "MyParm" that
// contains a one-dimensional array of range data.
int nIndex = 2; // Specify which element in array
HUTARNGARR hMyData; // handle to array of range data
// get handle to data container that contains array of range data
hMyData = UtaPbGetRangeArray(hMyParmBlock, "MyParm");
HUTARANGE hMyRangeData; // declare variable for handle to range data
hMyRangeData = UtaRangeCreate(8,24,4); // create range & set values
UtaRngArrSetAt1(hMyData, lIndex, hMyRangeData); // write range data
double dStart, dStop;
int nPoints;
// Get a handle & values from an element in the array
hMyRangeData = UtaRngArrGetAt1(
  hMyData,
  nIndex,
  &dStart,
  &dStop,
  &nPoints
  );
char chMessage[40];
sprintf(chMessage, "Start = %f, Stop = %f, Points = %d", dStart, dStop,
  nPoints);
AfxMessageBox(chMessage, MB_OK); // display the range values
```

See also

UtaRngArrGetAt1()

UtaRngArrSetAt2()

This function updates the values of range data stored in a data container in an element in a two-dimensional array whose handle is specified.

```
void UtaPtArrSetAt2(
    HUTARNGARR hRngArray,     // handle to an array
    UtaInt16 iIndex1,         // index of element in a row in the array
    UtaInt16 iIndex2,         // index of element in a column in the
                              // array
    HUTARANGE hValue          // handle to range data to be written to
                              // element
);
```

Parameters

hRngArray
The handle to an array of data whose type is range.

iIndex1
The index of an element in a row in the array.

iIndex2
The index of an element in a column in the array.

hValue
The handle to range data whose values are to be written to the specified element in the array.

Return Value

(none)

Remarks

Normally, data is passed in through parameter blocks created using the Action Definition Editor, and need not be created programmatically.

Example

```
// Example assumes an action exists whose parameter block has a
// parameter named "MyParm" that contains a two-dimensional array of
// range data.
int nRow = 2, nColumn = 1; // Specify which element in array
HUTARNGARR hMyData; // handle to array of range data
// get handle to data container that contains array of range data
hMyData = UtaPbGetRangeArray(hMyParmBlock, "MyParm");
HUTARANGE hMyRangeData; // declare variable for handle to range data
hMyRangeData = UtaRangeCreate(8,24,4); // create range & set values
UtaRngArrSetAt2(hMyData, nRow, nColumn, hMyRangeData); // write data
double dStart, dStop;
int nPoints;
// Get a handle & values from an element in the array
hMyRangeData = UtaRngArrGetAt2(
   hMyData,
   nRow,
   nColumn,
   &dStart,
   &dStop,
   &nPoints
   );
char chMessage[40];
sprintf(chMessage, "Start = %f, Stop = %f, Points = %d", dStart, dStop,
   nPoints);
AfxMessageBox(chMessage, MB_OK); // display the range values
```

See also

UtaRngArrGetAt2()

Functions for Tracing During Testplan Execution

The API functions described in this section let actions send messages to HP TestExec SL's Trace window as a testplan executes.

UtaTrace()

This function is used to send a user-defined message to HP TestExec SL's Trace window in the default stream of trace information from an action.

```
void UtaTrace(
    LPCSTR lpszMessage    // pointer to a string sent to the Trace
                          // window
);
```

Parameters

lpszMessage
The pointer to a string that contains a message to be sent to the Trace window when this function is called.

Return Value

(none)

Remarks

This function is optional.

Any formatting of your message string must be handled outside this function. Also, this function does not let you control which stream of trace information your message appears in. If desired, you can use `UtaTraceEx()` to format a message string, specify the name of the trace stream in which it appears, and send the message string in a single function.

Example

```
...(code inside an action routine)
// Send a message to the Trace window if tracing is
// enabled for a test that contains this action
UtaTrace("This message sent to the Trace window\n");
...(more code inside an action routine)
```

See also

UtaTraceEx()

UtaTraceEx()

This function is used to format, specify the trace stream for, and send a user-defined message to HP TestExec SL's Trace window from an action.

```
void UtaTraceEx(
    LPCSTR lpszStreamName,    // pointer to name of stream in which
                              // message appears
    LPCSTR pFormat            // pointer to format
    );
```

Parameters

lpszStreamName

Pointer to a string that contains the name of the stream of trace information to which the message should be sent when this function is called. Specify "" (null string) to have the message sent to the default trace stream.

pFormat

Pointer to a string that contains a formatted message to be sent to the Trace window when this function is called. You can use formatting codes that are valid for the `printf()` function in C to format the message contained in this string.

Return Value

(none)

Remarks

This function is optional.

Example

```
// Send a message to the Trace window if tracing is enabled for a test
// that contains this action. Message is sent to default trace stream.
UtaTraceEx ("", "Current value is %d\n", nValue);

// To illustrate how UtaTraceEx() simplifies formatting, the previous
// example might look like this if it used UtaTrace() instead:
char szMessageString[20];
sprintf (szMessageString, "Current value is %d\n", nValue);
UtaTrace(szMessageString);

// Send a message to the Trace window if tracing is enabled for a test
// that contains this action. Message is sent to user-specified trace
// stream.
UtaTraceEx ("MyTraceStream", "Current value is %d\n", nValue);
```

See also

UtaTrace()

Functions for User-Defined Messages

The API functions described in this section let actions, operator interfaces, and hardware handlers send user-defined messages to one another. These messages provide a means of communicating across processes, such as between HP TestExec SL and an operator interface written in Visual Basic.

UtaSendUserDefinedMessage()

This function is used to broadcast a message, and an identifier of the type of message, to all potential listeners. Potential listeners include actions in tests, operator interfaces, and hardware handlers.

```
void UtaSendUserDefinedMessage(
    long lID,                   // identifier of the type of message
    LPCSTR lpszMessage          // pointer to the message's contents
    );
```

Parameters

lID
: An identifier that listeners can use to determine if this message is intended for them.

lpszMessage
: Pointer to a string that contains the message broadcast to listeners when this function is called.

Return Value

(none)

Remarks

Unlike `UtaSendUserDefinedQuery()`, this function does not wait for a response; i.e., it simply broadcasts the message.

Example

```
// Code in action routine written in C...
// An ID of 2 is a pass/fail status message for a voltage measurement
if (Voltage > 5)
   UtaSendUserDefinedMessage(2, "passed");
else
   UtaSendUserDefinedMessage(2, "failed");
// More code in action routine...
```

See also

UtaSendUserDefinedQuery()

UtaSendUserDefinedQuery()

This function is used to broadcast a message, and an identifier of the type of message, to all potential listeners. It then waits a specified amount of time (or until an operator abort is seen) for a response from a listener. Potential listeners include actions in tests, operator interfaces, and hardware handlers.

LPCSTR UtaSendUserDefinedQuery(

long *lID*,	// identifier of the type of message
LPCSTR *lpszMessage*,	// pointer to the message's contents
double *secsTimeout*	// number of seconds to wait for response
);	

Parameters

lID
An identifier that identifies which listener should respond to this query.

lpszMessage
Pointer to a string that contains the message broadcast to listeners when this function is called.

secsTimeout
Number of seconds to wait for a response from a listener.

Return Value

Returns a pointer to a string that contains a response from a listener. When a response is detected, you should immediately move to a local variable the value returned .

Remarks

Use `UtaSendUserDefinedResponse()` to respond to this query.

If this function does not receive a response, it returns NULL. Thus, you probably will want to check for NULL before assuming the response was valid.

Example

```
// Code in action routine written in C...
// ID is 4 and timeout value is 2 seconds
if (UtaSendUserDefinedQuery(4, "Waiting for a response", 2) != NULL)
   ...code that does some task if response is received
else
   // exceeded time-out value or an error occurred
// More code in action routine...
```

See also

UtaSendUserDefinedMessage()
UtaSendUserDefinedResponse()

UtaSendUserDefinedResponse()

This function is used to broadcast a user-defined message to a specified listener who is awaiting a response.

void UtaSendUserDefinedRespone(

long *lID*, // identifier of the type of message

LPCSTR *lpszMessage* // pointer to the message's contents

);

Parameters

lID
An identifier that listeners can use to determine if this message is intended for them.

lpszMessage
Pointer to a string that contains the message broadcast to listeners when this function is called.

Return Value

(none)

Remarks

Use `UtaSendUserDefinedQuery()` to send a query to which this function provides a response.

Example

```
// Code in hardware handler written in C
void UTADLL AdviseUserDefinedMessage(HUTAHWMOD hModule,
                                     HUTAPB hParameterBlock,
                                     LPVOID pUserInitData,
                                     long lID)
                                     LPCSTR lpszMessage)
  {
  if (lID == 2)
     ...get status of switch via some I/O strategy
     if (SwitchClosed == 1)
       (UtaSendUserDefinedResponse(2, "Yes");
     else
       (UtaSendUserDefinedResponse(2, "No");
  }
```

See also

UtaSendUserDefinedQuery()

3

The Hardware Handler Function & API Reference

This chapter describes the API functions and user-written functions that HP TestExec SL can use to control hardware modules via a hardware handler.

For more information, see Chapter 2 in the *Customizing HP TestExec SL* book.

Functions Used in a Hardware Handler

This section describes the functions whose implementation code you write when creating a hardware handler, some of which contain calls to the Hardware Handler API.

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

Mandatory General-Purpose Functions

The functions described in this section must appear in all hardware handlers. They are useful in various kinds of hardware handlers.

Init()

This function initializes or "opens" a hardware module. Code that you write to implement this function should do whatever is needed to initialize the hardware module.

Parameters

```
LPVOID UTADLL Init(
    HUTAHWMOD hModule,          // handle to hardware module
    HUTAPB hParameterBlock      // handle to parameter block
);
```

hModule
: Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
: Handle to a parameter block for a structure. The parameter block whose handle is passed into this function (which is declared with the

`DeclareParms()` function) should contain data unique to this instance of the hardware module, such as its location and configuration.

Return Value

A pointer to `void` that points to a structure. You can cast this pointer to another data type as needed. You can return a pointer to user-defined initialization data—such as a handle to the I/O session, VXI*plug&play* ViSession, or SICL session—you wish to pass as the `pUserInitData` parameter into other functions in your hardware handler.

Remarks

This function must appear in all hardware handlers.

When running a testplan, HP TestExec SL calls this function to initialize each instance of a hardware module that uses this hardware handler.

This function lets you create—i.e., "new"—a structure of your choosing and pass it as the `pUserInitData` parameter in other functions, such as `GetPosition()` and `SetPosition()`. You might use this function to create a structure to hold transient data used by a function in your hardware handler. Later, you could use “delete” in the `Close()` function to reclaim the memory used by the structure.

Prior to version 2.00 of HP TestExec SL, this function was called `BindParms()`. Although HP TestExec SL will accept either name, `Init()` is the preferred name for future use.

This function also resets the module when called.

Example

```
LPVOID UTADLL Init (HUTAHWMOD hModule, HUTAPB hParameterBlock)
  {
  PMATParmStruct* p;
  p = new PMATParmStruct;
  // Bind all required values into structure
  p->pinInst = UtaPbGetInst(hParameterBlock, "PinInstrument");
  return((LPVOID)p);
  }
```

See also

Close()

Close()

This function closes a hardware module opened with the `Init()` function. Code that you write to implement this function should do whatever is needed to close the hardware module, such as freeing or deleting any memory associated with a structure created with the `Init()` function.

```
void UTADLL Close(
    HUTAHWMOD hModule,          // handle to hardware module
    HUTAPB hParameterBlock,     // handle to parameter block
    LPVOID pUserInitData        // optional for enhanced functionality
    );
```

Parameters

hModule

Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock

Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData

Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

Return Value

(none)

Remarks

This function must appear in all hardware handlers.

When running a testplan, HP TestExec SL calls this function for each instance of a hardware module that uses this hardware handler. This function is called when it is time to close the hardware module, such as when HP TestExec SL exits or when the system configuration changes.

Note that HP TestExec SL only calls this function if the hardware module was opened by a successful—i.e., no exceptions were raised—call to the `Init()` function.

Prior to version 2.00 of HP TestExec SL, this function was called Un`bindParms()`. Although the software will accept either name, `Close()` is the preferred name for future use.

Example

```
void UTADLL Close(
   HUTAHWMOD hModule,
   HUTAPB hParameterBlock,
   LPVOID pUserInitData)
   {
   delete pUserInitData;
   pUserInitData = NULL;
   }
```

See also

Init()

Reset()

This function resets a hardware module and lets you return the amount of time it will take to finish resetting, if any. Code that you write to implement this function should do whatever is needed to reset the hardware module to whatever you want its default state to be.

```
UTAUSECS UTADLL Reset(
   HUTAHWMOD hModule,        // handle to hardware module
   HUTAPB hParameterBlock,   // handle to parameter block
   LPVOID pUserInitData,     // optional for enhanced functionality
   );
```

Parameters

hModule

Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock

Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData

Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

Return Value

Number of microseconds needed until all the switching elements are reset.

Remarks

This function must appear in all hardware handlers.

When running a testplan, HP TestExec SL calls this function to reset hardware as needed, such as when a new testplan is loaded or when recovery from an error is required.

Note that to prevent "hot switching," HP TestExec SL resets instruments before resetting hardware handlers.

Example

```
UTAUSECS UTADLL Reset(
   HUTAHWMOD hModule,
   HUTAPB hParameterBlock,
   LPVOID pUserInitData)
   {
   // Hardware handler code, which might examine bit in
   // VXIbus register or format correct string and send to
   // a switching module. It might also keep a cache of the
   // current state based on calls to SetPosition
   // function.
   return TIME_TO_RESET;
   }
```

DeclareParms()

This function is used to declare parameters that HP TestExec SL passes to the DLL containing the hardware handler. These parameters tell the DLL which hardware module is being used—which is essential when the same DLL is used with more than one module of the same type—and, in some cases, supply additional information needed by the DLL.

Code that you write to implement this function should call the `UtaHwModDeclareParm()` function to declare any parameters needed to distinguish one instance of the hardware module from another, such as the module's HP-IB address, VXIbus slot number, etc. Also, you can use this function to pass configuration parameters, such as a parameter that chooses between 2x8 and 4x4 multiplexer configurations in a switching module.

Parameters

void UTADLL DeclareParms(

HUTAHWMOD *hModule*, // handle to hardware module

HUTAPBDEF *hParmBlockDef* // handle to parameter block definition

);

hModule
Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParmBlockDef
Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the switching module, such as its location and configuration.

Return Value

(none)

Remarks

This function must appear in all hardware handlers.

This function uses a call to the `UtaHwModDeclareParm()` API.

Valid data types for parameters, and their closest C equivalents, are:

Data Type	C Equivalent
CUtaReal64	64-bit real
CUtaInt32	32-bit integer
CUtaString	string
CUtaComplex	complex (imaginary, real)
CUtaInst	instrument data structure

CUtaRange	range (start, stop, steps)
CUtaPoint	point (x, y)
CUtaReal64Array	64-bit array of reals
CUtaInt32Array	32-bit array of integers
CUtaStringArray	array of strings
CUtaPointArray	array of points
CUtaRangeArray	array of ranges

This function is called when you use the Switching Topology Editor to edit switching topology, and not when running testplans.

Example

```
void UTADLL DeclareParms(HUTAHWMOD hModule, HUTAPBDEF hParmBlockDef)
  {
  //
  // Code will declare any parameters needed by the module
  // in order to talk to the module later. This example specifies
  // that users needs to provide the name of the hardware module
  // to be looked up in a driver configuration file. Later drivers
  // will be called to open and close relays using the name
  // provided by users.
  UtaHwModDeclareParm (
    hModule, // Passed into DeclareParms above
    hParmBlockDef, // Passed into DeclareParms above
    "InstrumentName", // Parameter name to present to user
    "CUtaString", // String data type
    "Please provide the name of the switching module to use");
  // Declare any other parameters
  }
```

See also

UtaHwModDeclareParm()

Mandatory Switching-Specific Functions

The functions described in this section must appear in all hardware handlers that control switching hardware—i.e., "switching handlers"—but are not needed in other kinds of hardware handlers.

DeclareNodes()

This function is used to define all of the nodes and adjacencies—i.e., adjacent nodes that can be connected via a switching element—for a switching module. Code that you write to implement this function should call the `UtaHwModDeclareNode()` and `UtaHwModDeclareAdjacent()` API functions to declare nodes and adjacencies, respectively, in the switching module.

Parameters

```
void UTADLL DeclareNodes(
  HUTAHWMOD hModule,          // handle to hardware module
  HUTAPB hParameterBlock      // handle to parameter block
  );
```

hModule
: Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
: Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

Return Value

(none)

Remarks

This function must appear in all hardware handlers that control switching hardware.

This function uses calls to the `UtaHwModDeclareNode()` and `UtaHwModDeclareAdjacent()` APIs.

Be aware that user parameters are seldom used by `DeclareNodes()`. When user parameters are used, they usually contain information about how to talk to the module. However, some modules can be set up in multiple configurations, such as a 2x8 or a 4x4 matrix. If so, the hardware handler must know this to declare the proper set of nodes and adjacencies in the topology.

This function is called when you use the Switching Topology Editor to edit switching topology, and not when running testplans.

Example

```
void UTADLL DeclareNodes(HUTAHWMOD hModule, HUTAPB hParameterBlock)
  {
  // Code will declare all needed connectors and internal nodes to
  // describe the topology of the card.
  // The example has two connectors on the switch card; Connector A and
  // Connector B. Each Connector has four pins. The card is 4x4 matrix
  // with 16 relays. Each pin on Connector A can get to any pin on
  //Connector B by closing a single relay.
  UtaHwModDeclareNode(
    hModule,
    "ConnectorA.1",
    "The first pin of the connector A",
    NULL);
  UtaHwModDeclareNode(
    hModule,
    "ConnectorA.2",
    "The second pin of the connector A",
    NULL);

  ...(more declarations)

  UtaHwModDeclareNode(
    hModule,
    "ConnectorB.3",
    "The third pin of the connector B",
    NULL);
  UtaHwModDeclareNode(
    hModule,
    "ConnectorB.4",
    "The fourth pin of the connector B",
    NULL);
```

```
// Declare the adjacencies.  This could be written more concisely
// with loops but is written linearly for clarity.
UtaHwModDeclareAdjacent(
   hModule,
   "ConnectorA.1",
   "ConnectorB.1",
   1, 1); // Relay 1 closed
UtaHwModDeclareAdjacent(
   hModule,
   "ConnectorA.1",
   "ConnectorB.2",
   2, 1); // Relay 2 closed
...(more declarations)
UtaHwModDeclareAdjacent(
   hModule,
   "ConnectorA.4",
   "ConnectorB.3",
   15, 1); // Relay 15 closed
UtaHwModDeclareAdjacent(
   hModule,
   "ConnectorA.4",
   "ConnectorB.4",
   16, 1); // Relay 16 closed
}
```

GetPosition()

This function returns the position of a specified switching element. Code that you write to implement this function should return the current position of a switching element, such as a relay, as it exists in the switching module.

You can query the switching module for the position or, if there is significant overhead when talking to the card, cache the positions of the switching elements in your hardware handler. If you use caching, remember to change

this cached information in the code that implements the `SetPosition()` and `Reset()` functions.

```
IDUTASWPOS UTADLL GetPosition(
    HUTAHWMOD hModule,          // handle to hardware module
    HUTAPB hParameterBlock,     // handle to parameter block
    LPVOID pUserInitData,       // optional for enhanced functionality
    IDUTASWELM idElement        // identifier of switching element
    );
```

Parameters

hModule
: Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
: Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData
: Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

idElement
: Unsigned integer that identifies a switching element in the switching module.

Return Value

Returns the position of the specified switching element. 0 is the default (reset) position.

Remarks

This function must appear in all hardware handlers that control switching hardware.

Switching elements, *idElement*, start at 1 and need not be contiguous. The numbers are declared by the module developer and map to the various relays, rotary switches, muxes, etc. in the module as described in the `DeclareNodes()` function.

Each switching element can be set to a position, *idPosition*. Positions start at 0, continue upward and must be contiguous. 0 is the default position. For a simple relay, 0 is open and 1 is closed.

When running a testplan, HP TestExec SL calls this function in response to path closures requested by switching actions in tests. Depending upon the options specified in the switching actions, HP TestExec SL may "remember" the current positions of switching elements so it can return them to their original states at the end of a test.

Example

```
IDUTASWPOS UTADLL GetPosition(
   HUTAHWMOD hModule,
   HUTAPB hParameterBlock,
   LPVOID pUserInitData,
   IDUTASWELM idElement)
   {
   // Handler code. Code might examine bit in VXI register
   // or format correct string and send to switching
   // module. It might also keep a cache of the current
   // state based on calls to SetPosition function.
   return thePosition; // return position of element
   }
```

See also

SetPosition()

SetPosition()

This function is used in a switching handler to set the position of switching elements, such as opening and closing relays, in a switching module. Code

that you write to implement this function should do whatever is needed to set the position of a switching element in the switching module.

Parameters

```
UTAUSECS UTADLL SetPosition(
    HUTAHWMOD hModule,          // handle to hardware module
    HUTAPB hParameterBlock,     // handle to parameter block
    LPVOID pUserInitData,       // optional for enhanced functionality
    IDUTASWELM idElement,       // identifier for switching element
    IDUTASWPOS idPosition       // identifier for position of switching
                                // element
    );
```

hModule
: Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
: Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData
: Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

idElement
: Unsigned integer that identifies the switching element whose position is being set.

idPosition
: Unsigned integer that identifies the position of a switching element in the switching module.

Return Value

Returns the time, in microseconds, that HP TestExec SL should wait for resetting to complete, if any. This will be however long it takes switching elements to change to their new positions.

Remarks

This function must appear in all hardware handlers that control switching hardware.

Switching elements, *idElement*, start at 1 and need not be contiguous. The numbers are declared by the module developer and map to the various relays, rotary switches, multiplexers, etc. in the module as described in the `DeclareNodes()` function.

Each switching element can be set to a position, *idPosition*. Positions start at 0, continue upward and must be contiguous. 0 is the default position. For a simple relay, 0 is open and 1 is closed.

When running a testplan, HP TestExec SL calls this function in response to path closures requested by switching actions in tests in the testplan. HP TestExec SL decodes the switching path (node name to node name) and determines which switching module, switching element in that module, and position of the switching element to set.

Example

```
UTAUSECS UTADLL SetPosition(
   HUTAHWMOD hModule,
   HUTAPB hParameterBlock,
   LPVOID pUserInitData,
   IDUTASWELM idElement,
   IDUTASWPOS idPosition)
   {
   // Code might set bit in VXI register or format correct
   // string and send to switching module.
   // Talk to hardware with the proper instrument drivers to set the
   // switching element to proper position. For example, a rotary
   // switch position. Simpler example is a relay. It has two
   // positions, opened and closed.
```

```
// Return number of microseconds before the relay will close or
// zero if providing an "IsPositionSet" routine.
return TIME_TO_CHANGE;
}
```

See also

IsPositionSet()
GetPosition()

Optional General-Purpose Functions

The functions listed in this section are optional. They are useful in various kinds of hardware handlers. If the additional functionality they provide is useful to you, you can implement them in your hardware handler.

DeclareStatus()

This function is useful if you want your hardware handler to send status information to HP TestExec SL's Watch window for debugging. Code that you implement in this function defines fields that appear in the Watch window when watching is enabled for a hardware module that uses this hardware handler.

```
void UTADLL DeclareStatus(
    HUTAHWMOD hModule,      // handle to hardware module
    HUTAPB hParameterBlock  // handle to parameter block
    );
```

Parameters

hModule

Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information. Also, it is a required parameter for the `UtaHwModDeclareStatus()` API.

hParameterBlock

Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

Return Value

(none)

Remarks

This function is optional in a hardware handler.

This function uses a call to the `UtaHwModDeclareStatus()` API.

Note that to provide information for the Watch window, you must write code that implements this function and the `GetStatus()` function.

Example

```
/* Declare all the watchable fields for this switching module. */
void UTADLL DeclareStatus (HUTAHWMOD hModule, HUTAPB hParameterBlock)
{
IDUTAHWSTAT idStatus;
idStatus = UtaHwModDeclareStatus (hModule, 1, "Row 0");    // returns 1
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 0"); // returns 2
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 1"); // returns 3
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 2"); // returns 4
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 3"); // returns 5
idStatus = UtaHwModDeclareStatus (hModule, 1, "Row 1");    // returns 6
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 0"); // returns 7
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 1"); // returns 8
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 2"); // returns 9
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 3"); // return 10
}
```

See also

GetStatus()
UtaHwModDeclareStatus()

GetStatus()

This function is useful if you want your hardware handler to send status information to HP TestExec SL's Watch window for debugging. HP TestExec SL calls this function to update data fields that appear in the Watch window when watching is enabled for a hardware module that uses this hardware handler.

The names and ids of the fields are defined in code you write to implement the `DeclareStatus()` routine, which uses the `UtaHwModDeclareStatus()` API.

```
void UTADLL GetStatus(
    HUTAHWMOD hModule,          // handle to hardware module
    HUTAPB hParameterBlock,     // handle to parameter block
    LPVOID pUserInitData,       // optional for enhanced functionality
    IDUTAHWSTAT idStatus,       // identifier of a status field
    LPSTR lpszBuffer,           // buffer that contains status information
    int nBufferSize             // size of buffer that contains status
                                // information
    );
```

Parameters

hModule

Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock

Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData
: Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

idStatus
: The identifier of the status field you define in code that implements the `DeclareStatus()` function. It identifies the field whose current status you are to provide.

lpszBuffer
: The `lpszBuffer` passed into this function lets you return status information for the `idStatus` field. It is a textual representation of the status information.

nBufferSize
: Specifies the length of the buffer provided. Do not index `lpszBuffer` longer than `nBufferSize` because information longer than the size of `lpszBuffer` cannot be displayed in the Watch window.

Return Value

(none)

Remarks

This function is optional in a hardware handler.

Example

```
*/ Provide the text status for each of the watchable fields. */
#define INSTRUMENT 0
#define ROW_0      1
#define ROW_1      6
#define GET_STATUS_CHAR(n) (((E2X4MUXParmStruct*) \
                           pUserInitData)->nElementState[n] ? 'X':'O')
#define GET_STATUS(n) (((E2X4MUXParmStruct*) \
                  pUserInitData)->nElementState[n] ? "CLOSED" : "OPEN")
```

```
void UTAAPI GetStatus (HUTAHWMOD hModule,
                       HUTAPB hParameterBlock,
                       LPVOID pUserInitData,
                       IDUTAHWSTAT idStatus,
                       LPSTR lpszBuffer,
                       int nBufferSize)
{
  lpszBuffer = "";
  if (idStatus == INSTRUMENT) /* Summary status of all of the relays */
    {
      if (nBufferSize > 8)
        sprintf (lpszBuffer, "%c%c%c%c%c%c%c%c",
                 GET_STATUS_CHAR(0),
                 GET_STATUS_CHAR(1),
                 GET_STATUS_CHAR(2),
                 GET_STATUS_CHAR(3),
                 GET_STATUS_CHAR(4),
                 GET_STATUS_CHAR(5),
                 GET_STATUS_CHAR(6),
                 GET_STATUS_CHAR(7));
      return;
    }
  if (idStatus == ROW_0) /* Summary status of row 0 relays */
    {
      if (nBufferSize > 4)
        sprintf (lpszBuffer, "%c%c%c%c",
                 GET_STATUS_CHAR(0),
                 GET_STATUS_CHAR(1),
                 GET_STATUS_CHAR(2),
                 GET_STATUS_CHAR(3));
      return;
    }
```

```
  if (idStatus == ROW_1) /* Summary status of row 1 relays */
    {
      if (nBufferSize > 10) return;
      sprintf (lpszBuffer, "%c%c%c%c",
               GET_STATUS_CHAR(4),
               GET_STATUS_CHAR(5),
               GET_STATUS_CHAR(6),
               GET_STATUS_CHAR(7));
      return;
    }

  if (idStatus < ROW_1) /* relays in row 0 */
    {
      if (nBufferSize > 10)
        sprintf (lpszBuffer, "%s", GET_STATUS(idStatus - 2));
      return;
    }

  if (idStatus > ROW_1) /* relays in row 1 */
    {
      if (nBufferSize > 10)
        sprintf (lpszBuffer, "%s", GET_STATUS(idStatus - 3));
      return;
    }
}
```

See also

DeclareStatus()

AdviseTrace()

This function is called “on the fly” whenever the status of tracing changes in HP TestExec SL. It is useful if you wish to vary the behavior of your hardware handler according to whether tracing is enabled or disabled; for

example, you may be able to speed up the operation of your hardware handler by formatting trace information only when tracing is enabled.

```
void UTADLL AdviseTrace(
    HUTAHWMOD hModule,          // handle to hardware module
    HUTAPB hParameterBlock,     // handle to parameter block
    LPVOID pUserInitData,       // optional for enhanced functionality
    BOOL bTracingEnabled,       // current status of tracing
    );
```

Parameters

hModule
Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData
Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

bTracingEnabled
TRUE if tracing is enabled, FALSE if it is disabled.

Return Value

(none)

Remarks

This function is optional in a hardware handler.

The status of tracing changes in HP TestExec SL whenever:

- A testplan that is running moves from a test for which tracing is enabled to a test for which tracing is not enabled, or vice-versa.
- You use Debug | Trace Settings in HP TestExec SL's menu bar to change an option for tracing.

Example

```
void UTADLL AdviseTrace(HUTAHWMOD hModule,
                        HUTAPB hParameterBlock,
                        LPVOID pUserInitData,
                        BOOL bTracingEnabled)
  {
  if (bTracingEnabled) // If tracing was turned on
     ...(set appropriate behavior for tracing enabled)
  else // If tracing was turned off
     ...(set appropriate behavior for tracing disabled)
  }
```

AdviseMonitor()

This function is useful when you want HP TestExec SL to monitor the status of hardware and do something when that status changes. This can be especially useful in automation applications. For more information, see "Monitoring the Status of Hardware" in Chapter 2 of the *Customizing HP TestExec SL* book.

void UTADLL AdviseMonitor(

HUTAHWMOD *hModule*, // handle to hardware module

HUTAPB *hParameterBlock*, // handle to parameter block

LPVOID *pUserInitData*, // optional for enhanced functionality

);

Parameters

hModule

Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock

Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData

Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

Return Value

(none)

Remarks

This function is optional in a hardware handler.

By default, this function is called every 100 milliseconds by HP TestExec SL. You can change the interval for polling by adding an entry named `Monitor Time Slice` to file "<*HP TestExec SL home*>\tstexcsl.ini" and specifying a different value in *micro*seconds, as shown below:

```
[Process]
Monitor Time Slice=500000
```

Note

The value of `Monitor Time Slice` affects the performance of your test system. The lower the value—i.e., the more frequently HP TestExec SL calls the `AdviseMonitor()` function in hardware handlers—the more time your system spends polling instead of testing.

Example

```
void UTADLL AdviseMonitor (HUTAHWMOD hModule,
                           HUTAPB hParameterBlock,
                           LPVOID pUserInitData,
                           BOOL bTracingEnabled)
  {
  ...(code that interrogates hardware)
  ...
  ...(code that does a task based on results of interrogating hardware)
  }
```

AdviseUserDefinedMessage()

This function is useful when you want HP TestExec SL to control hardware in response to a user-defined message, such as a message sent by an operator interface.

void UTADLL AdviseUserDefinedMessage(

HUTAHWMOD *hModule*,	// handle to hardware module
HUTAPB *hParameterBlock*,	// handle to parameter block
LPVOID *pUserInitData*,	// optional for enhanced functionality
long *lID*,	// identifier of the type of user-defined // message
LPCSTR *lpszMessage*	// pointer to a user-defined message

);

Parameters

hModule
: Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
: Handle to a parameter block for a structure.

pUserInitData
: Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the Init() function.

lID
: A long whose value identifies which kind of user-defined message was received.

lpszMessage
: A pointer to a string that contains the user-defined message that was received.

Return Value

(none)

Remarks

This function is optional in a hardware handler.

Example

```
void UTADLL AdviseUserDefinedMessage(HUTAHWMOD hModule,
                                     HUTAPB hParameterBlock,
                                     LPVOID pUserInitData,
                                     long lID)
                                     LPCSTR lpszMessage)
  {
  if (lID = 10020) // Assumes ID 10020 means to turn on warning light
    UtaHwModTraceEx (hModule, "Operator interface wants to turn on a
      warning light %d, %s", lID, lpszMessage);
  }
```

Optional Switching-Specific Functions

The functions described in this section are specific to hardware handlers used to control switching hardware; i.e., "switching handlers." If the additional functionality they provide is useful to you, you can implement them in your hardware handler.

IsPositionSet()

This function lets you determine if a specified switching element has finished closing or opening. It is useful if you need to poll a switching module to verify that one of its switching elements has changed position.

BOOL UTADLL IsPositionSet(

HUTAHWMOD *hModule*, // handle to hardware module

HUTAPB *hParameterBlock*, // handle to parameter block

LPVOID *pUserInitData*, // optional for enhanced functionality

IDUTASWELM *idElement* // identifier of switching element

);

Parameters

hModule
: Handle to an instance of a hardware module. The handle to the module is useful should you need to raise an exception or wish to provide trace information.

hParameterBlock
: Handle to a parameter block for a structure. The parameter block passed into this function (which is declared with the `DeclareParms()` function) contains data unique to this instance of the hardware module, such as its location and configuration.

pUserInitData
: Pointer to user-defined initialization data optionally used to enhance the functionality of the hardware handler. This is the value returned by the `Init()` function.

idElement
: Unsigned integer that identifies a switching element in the hardware module. This is the switching element whose change of position is being verified.

Return Value

Returns TRUE when the specified switching element has finished changing positions. Return FALSE if it has not completed changing positions.

Remarks

This function is optional in a hardware handler.

This function seldom is used because a timer function is usually provided for this purpose, and the time needed to wait for the switching element to change positions (such as opening/closing a relay) is returned by the `SetPosition()` function. Sometimes, though, such as when using rotary switches or cascaded multiplexers, you do not know how much time is needed. In that case, you can return 0 from `SetPosition()`, which causes HP TestExec SL to call this function to explicitly interrogate the switching module. Once called, this function will continue to be called until it returns TRUE.

Example

```
BOOL UTADLL IsPositionSet(
   HUTAHWMOD hModule,
   HUTAPB hParameterBlock,
   LPVOID pUserInitData,
   IDUTASWELM idElement)
   {
   // Code might check a bit in a VXI register or format
   // correct string and send to switching module.
   // Determine if the relay is closed yet and return TRUE
   // or FALSE;
   // If this routine is not provided, the default is
   // equivalent to always returning TRUE.
   return TRUE;
   }
```

See also

SetPosition()

The Hardware Handler API

You call functions in the Hardware Handler API from functions you write when creating a hardware handler.

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaHwModDeclareAdjacent()

This API function is used to declare adjacencies—i.e., adjacent nodes that can be connected via a switching element—in a hardware module.

```
void UtaHwModDeclareAdjacent(
    HUTAHWMOD hModule,          // handle to hardware module
    LPCSTR lpszFromNode,        // name of node
    LPCSTR lpszToNode,          // name of node
    IDUTASWELM idElement,       // identifier of switching element
    IDUTASWPOS idPosition       // identifier of position of switching
                                // element
);
```

Parameters

hModule
Handle to an instance of a hardware module.

lpszFromNode
Name of the first node in a pair of adjacent nodes.

lpszToNode
Name of the second node in a pair of adjacent nodes.

idElement
Identifier of the switching element that connects the pair of adjacent nodes.

idPosition
Identifier of the position the switching element must be in to connect the adjacent nodes.

Return Value

(none)

Remarks

This API is used in the `DeclareNodes()` function.

See also

DeclareNodes()

UtaHwModDeclareNode()

This API function is used to declare a node in a hardware module.

```
HUTANODE UtaHwModDeclareNode(
    HUTAHWMOD hModule,              // handle to hardware module
    LPCSTR  lpszNodeName,           // name of node
    LPCSTR  lpszDescription=NULL    // optional description of node
    LPCSTR  lpszKeywordList=NULL    // optional keyword(s) for node
    );
```

Parameters

hModule
Handle to an instance of a hardware module.

lpszNodeName
Name of the node.

lpszDescription
An optional description of the node.

lpszKeywordList
An optional list of keywords associated with the node. Keywords declared here make it easier to find specific nodes when using the Switching Topology Editor to define switching topology. Keywords in the list must be enclosed in quotation marks and separated by commas—e.g., “keyword” or “keyword1, keyword2”—and must not contain spaces.

Note

If your hardware handler is written in C—i.e., your source files have a .C extension—you must use NULL or a null string (““) as a placeholder for the *lpszKeywordList* parameter even if you do not wish to specify a value for it. If your hardware handler is written in C++—i.e., your source files have a .CPP extension—you can omit this parameter entirely.

Return Value

Returns the handle to the node being declared.

Remarks

This API is used in the `DeclareNodes()` function.

See also

DeclareNodes()

UtaHwModDeclareParm()

This API function is used to declare parameters in a parameter block for a hardware module.

```
HUTADATA UtaHwModDeclareParm(
    HUTAHWMOD hModule,              // handle to hardware module
    HUTAPBDEF hPbDef,               // handle to parameter block
                                    // definition
    LPCSTR lpszParmName,            // name of parameter
    LPCSTR lpszType,                // data type of parameter
    LPCSTR lpszDescription = NULL   // optional description of parameter
    );
```

Parameters

hModule
Handle to the module passed to the `DeclareParms()` function.

hPbDef
Handle to the parameter block definition used by the Switching Topology Editor.

lpszParmName
Name you wish to call the parameter being declared.

lpszType
Data type of the parameter being declared; i.e., int, real, string, etc.

lpszDescription
An optional description of the parameter that you want the Switching Topology Editor to present to its user. A handle to the created parameter is passed back. You can optionally use this parameter to set a "default" value to be presented to users of the Switching Topology Editor.

Return Value

Returns the handle to the parameter being declared or NULL if the parameter could not be declared

This handle is useful if you wish to set the parameter to a default value that appears when using the Switching Topology Editor to define switching topology. An example might look like this:

```
HUTADATA hData = UtaHwModDeclareParm(
                 hModule,
                 hParameterBlockDefinition,
                 BOARDID_PARMNAME,
                 "CUtaInt32",
                 "GPIB board ID");
UTAINT32 UtaInt32SetValue((HUTAINT32) hData, 24); // Set default value
```

Remarks

This API is used in the `DeclareParms()` function.

See also

DeclareParms()

UtaHwModDeclareRev()

This API function is used to declare the current revision of the hardware handler software. Implementing this function provides the Switching Topology Editor with revision information for hardware handlers when creating topology layers.

void UtaHwModDeclareRev(

HUTAHWMOD *hModule*,	// handle to hardware module
LONG *wMajorNumber,*	// major number of revision
LONG *wMinorNumber,*	// minor number of revision
LPCSTR *lpszAuditText*	// descriptive text

);

Parameters

hModule
Handle to the hardware module whose hardware handler revision is being declared.

wMajorNumber
Major number that denotes a significant revision.

wMinorNumber
Minor number that denotes a lesser revision.

lpszAuditText
Text that describes the revision.

Return Value

(none)

Remarks

This function is used when writing the `DeclareParms()` function in a hardware handler.

The switching topology data associated with each testplan must accurately match hardware resources described in the hardware handler(s) used with the testplan. When you use this function to declare a hardware handler's revision, you can use `UtaHwModGetRev()` to return the revision in use when the topology data was created, and mimic backward-compatible functionality in the hardware handler as needed.

See also

DeclareParms()
UtaHwModGetRev()

UtaHwModGetRev()

This API function queries the switching topology data for the current testplan and returns the version of hardware handler in use when the testplan's switching topology layers were created.

```
HUTAREV UtaHwModGetRev(
    HUTAHWMOD hModule          // handle to hardware module
);
```

Parameters

hModule
: Handle to the hardware module whose revision when creating layers in the switching topology needs to be known.

Return Value

Returns the handle to revision data that contains the version of the hardware handler used when the switching topology layers for the testplan were created.

Remarks

This function is optional and can be used to maintain backward compatibility as hardware handlers are modified over time. For example, if the version returned by this function is older than the current version, you can have your hardware handler implement the old functionality.

See also

UtaHwModDeclareRev()

UtaHwModTrace()

This API function is used to send a user-defined message to HP TestExec SL's Trace window in the default stream of trace information from a hardware handler.

```
void UtaHwModTrace(
    HUTAHWMOD hModule,     // handle to hardware module
    LPCSTR lpszMessage     // pointer to a string sent to Trace window
    );
```

Parameters

hModule
: Handle to a hardware module associated with a hardware handler in which this function is implemented.

lpszMessage
: Pointer to a string that contains a message to be sent to the Trace window when this function is called.

Return Value

(none)

Remarks

This function is optional.

This function sends a message to the Trace window only when tracing is enabled for a test that contains an action that uses a hardware handler (such as a switching action) in which this function appears.

Any formatting of your message string must be handled outside this function. Also, this function does not let you control which stream of trace information your message appears in. If desired, you can use `UtaHwModTraceEx()` to format a message string, specify the name of the trace stream in which it appears, and send the message string in a single function.

Example

```
UTAUSECS UTADLL Reset (HUTAHWMOD hModule,
                       HUTAPB hParameterBlock,
                       LPVOID pInitData)
  {
  ...(code that implements the Reset function)
  // Send a message to the Trace window if tracing is enabled for a test
  // that contains an action that uses this hardware handler
  UtaHwModTrace (hModule, "Opened all switching elements\n");
  ...(more code in Reset function)
  }
```

See also

UtaHwModTraceEx()
UtaHwModIsTracing()

UtaHwModTraceEx()

This API function is used to format, specify the trace stream for, and send a user-defined message to HP TestExec SL's Trace window from a hardware handler.

void UtaHwModTraceEx(

HUTAHWMOD *hModule*,	// handle to hardware module
LPCSTR *lpszStreamName*,	// pointer to name of stream in // which message appears
LPCSTR *pFormat*	// pointer to a formatted message // string sent to the Trace window

);

Parameters

hModule

Handle to a hardware module associated with a hardware handler in which this function is implemented.

lpszStreamName
Pointer to a string that contains the name of the stream of trace information to which the message should be sent when this function is called. Specify "" (null string) to have the message sent to the default trace stream.

pFormat
Pointer to a string that contains a formatted message to be sent to the Trace window when this function is called.

Return Value

(none)

Remarks

This function is optional.

This function sends a message to the Trace window only when tracing is enabled for a test that contains an action that uses a hardware handler (such as a switching action) in which this function appears.

Example

```
// Send a message to the Trace window if tracing is enabled for a test
// that contains an action that uses this hardware handler. Message is
// sent to the default trace stream.
UtaHwModTraceEx ("", "Current value is %d\n", nValue);

// To illustrate how UtaHwModTraceEx() simplifies formatting, the
// previous example might look like this if it used UtaHwModTrace():
char szMessageString[20];
sprintf (szMessageString, "Current value is %d\n", nValue);
UtaHwModTrace(hModule, szMessageString);

// Send a message to the Trace window if tracing is enabled for a test
// that contains an action that uses this hardware handler. Message is
// sent to user-specified trace stream.
UtaHwModTraceEx ("MyTraceStream", "Current value is %d\n", nValue);
```

See also

UtaHwModTrace()
UtaHwModIsTracing()

UtaHwModIsTracing()

This API function is used to determine if tracing is enabled for one or more tests that contain actions that use a hardware handler in which this function is implemented.

```
BOOL UtaHwModIsTracing(
    HUTAHWMOD hModule          // handle to hardware module
);
```

Parameters

hModule
Handle to an instance of a hardware module.

Return Value

Returns TRUE if tracing is enabled, and FALSE if it is not.

Remarks

This function is optional.

Collecting and formatting data to be sent to the Trace window takes time, which means that a hardware handler that implements tracing may run slower than one that does not. Because tracing is useful when you are debugging tests, but usually is not needed during production testing, you may want to use this function to turn tracing features on when tests are "listening" for Trace information from your hardware handler, and off when they are not. This lets you speed up your hardware handler by collecting and formatting trace information only when tracing is being used.

Example

```
// The following example is simplistic because very little trace
// information is being processed, but speed gains could be significant
// if more data were being processed for tracing; e.g., if you
// interrogated a switching module for the status of all of its
// switching elements and formatted that information for presentation
// in the Trace window.
UTAUSECS UTADLL Reset (HUTAHWMOD hModule,
                       HUTAPB hParameterBlock,
                       LPVOID pInitData)
  {
  ...(code that implements the Reset function)
  // Send trace message only if this module is being traced
  If UtaHWModIsTracing(hModule)
    UtaHwModTrace (hModule, "Opened all switching elements\n");
  ...(more code in Reset function)
  }
```

See also

UtaHwModTrace()
UtaHwModTraceEx()

UtaHwModDeclareStatus()

This API function is used to specify the indentation level of status labels sent to the Watch window, which lets you nest status information so it is presented in an organized manner.

IDUTAHWSTAT UtaHwModDeclareStatus(

HUTAHWMOD *hModule*,	// handle to hardware module
UTASTATLEVEL *Level*,	// indentation level of status label
LPCSTR *lpszLabel*	// pointer to string that contains a status label

);

Parameters

hModule
: Handle to a hardware module associated with a hardware handler in which this function is implemented.

Level
: The desired indentation level for the status label. *Level* starts at 1 and can be incremented to indent this status label under the previous label.

lpszLabel
: Pointer to a string that contains the status label whose indentation level is being set.

Return Value

A status identifier that indicates the indentation level of a status label sent to the Watch window

Remarks

This function is optional.

This function is used when implementing the `DeclareStatus()` function in a hardware handler.

Calling this API defines each of the status labels. The level defines the indentations (heirarchical groupings) of the status.

Example

```
// Declare all the watchable fields for this switching module
void UTADLL DeclareStatus (HUTAHWMOD hModule, HUTAPB hParameterBlock)
{
IDUTAHWSTAT idStatus;
idStatus = UtaHwModDeclareStatus (hModule, 1, "Row 0");    // returns 1
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 0"); // returns 2
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 1"); // returns 3
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 2"); // returns 4
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 3"); // returns 5
idStatus = UtaHwModDeclareStatus (hModule, 1, "Row 1");    // returns 6
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 0"); // returns 7
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 1"); // returns 8
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 2"); // returns 9
idStatus = UtaHwModDeclareStatus (hModule, 2, "Column 3"); // return 10
}
```

See also

DeclareStatus()

4

The Exception Handling API Reference

This chapter describes the Exception Handling API, whose functions let you define what conditions to consider as exceptions during testing, handle user-defined exceptions and exceptions from external sources (such as the operating system), and programmatically abort testing.

For more information about handling exceptions in specific programming languages, see language-specific topics in the *Using HP TestExec SL* book.

Functions Used to Raise & Examine Exceptions

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaExcRaiseUserError()

This function is used to raise a user-defined exception from an action routine or switching handler. It also lets you specify an error message and severity indicator to be associated with the exception.

```
void UtaExcRaiseUserError(
    LPCSTR lpszMessage,     // error message associated with the
                            // exception
    int iSeverity           // severity indication for the exception
);
```

Parameters

lpszMessage
A message raised when an exception occurs.

iSeverity
An integer used to denote the severity of the exception.

Return Value

(none)

Remarks

You can use *iSeverity* to augment the error message with an integer that indicates the severity of the exception. Severity values are not defined by Hewlett-Packard and can be used in any fashion you consider appropriate.

When used by itself, `UtaExcRaiseUserError()` raises an exception that aborts testing and whose error message appears in the Report window. But if you follow `UtaExcRaiseUserError()` with a function that manipulates exceptions, such as `UtaExcRegReceiveError()`, testing continues and exceptions do not appear in the Report window.

Example

```
// Example causes the following to display in Report window when
// encountered while testplan is executing:
//   Condition raised a user-defined exception! (Severity: 5)
char chMessage [60];
long lSeverity;
...(do something)
if (some condition == some value) // raise an exception?
  {
  strcpy (chMessage, "Condition raised a user-defined exception!");
  lSeverity = 5;
  UtaExcRaiseUserError(chMessage, lSeverity);
  }
...(testing is aborted because exception occurred)
```

See also

UtaExcGetSeverity()

UtaExcRegIsError()

This function tests for the presence of one or more exceptions that have been raised but not yet received.

BOOL UtaExcRegIsError(

void **reserved* **UTACPP_DEFAULT(NULL)** // (not used)

);

Parameters

**reserved*
(not used)

Return Value

TRUE if exceptions have been raised and not yet received.

Example

```
// Check for exception, take corrective action, & clear
if (UtaExcRegIsError()) {
   ...(code that takes corrective action)
   UtaExcRegClearError();
   };
```

See also

UtaExcRegClearError()

UtaExcRegGetErrorCount()

This function returns the number of exceptions that have been raised and not yet received.

int UtaExcRegGetErrorCount(

void **reserved* **UTACPP_DEFAULT(NULL)** // (not used)

);

Parameters

**reserved*
(not used)

Return Value

An integer that reports the number of exceptions that have been raised but not received.

Example

```
// Example can raise user-defined exceptions while action is doing
// tasks. Each exception has a severity level associated with it. Near
// the end of the action, a routine checks to see if exceptions
// occurred and receives them if they did. If the severity of an
// exception exceeds a threshold, a value of -1 is written to a
// parameter named "parm1" in the action's parameter block. If "parm1"
// is a reference to a symbol in a symbol table, actions in other tests
// can access the symbol table to see if this action raised one or more
// "serious" exceptions.
HUTAEXC hUtaException;
long lSeverity, lNumExceptions, lCounter;
char chMessage[40];
...(do something)
// action routine contains one or more routines to see if an
// exception condition exists
if (some condition == some value) // raise an exception?
  {
  strcpy (chMessage, "Condition raised an exception!");
  lSeverity = 10; // assign severity level to this exception
  UtaExcRaiseUserError(chMessage, lSeverity); // raise exception
  }
...(testing continues)
...
...(near end of testing)
if (UtaExcRegIsError()) // if exception(s) exist
  {
  lNumExceptions = UtaExcRegGetErrorCount(); // get # of exceptions
  // receive all exceptions & get handle to first in list
  hUtaException = UtaExcRegReceiveError();
  for (lCounter = 1; lCounter <= lNumExceptions; lCounter++)
    {
    if (UtaExcGetSeverity(hUtaException) > 5) // test severity
      UtaPbSetInt32(hParmBlock, "parm1", -1); // write to parm.
    if (lCounter < lNumExceptions)
      // get handle to next exception
      hUtaException = UtaExcGetNextError(hUtaException);
    };
  };
```

UtaExcRegClearError()

This function clears any existing exceptions that have been raised but does not examine or receive them.

```
void UtaExcRegClearError(
    void *reserved UTACPP_DEFAULT(NULL)          // (not used)
    );
```

Parameters

**reserved*
(not used)

Return Value

(none)

Example

```
// Check for exception, take corrective action, & clear
if (UtaExcRegIsError()) {
   ...(code that takes corrective action)
   UtaExcRegClearError();
   };
```

See also

UtaExcRegIsError()

UtaExcRegReceiveError()

This function returns a handle to the first in a list of exceptions.

```
HUTAEXC UtaExcRegReceiveError(
    void *reserved UTACPP_DEFAULT(NULL)          // (not used)
    );
```

Parameters

**reserved*
(not used)

Return Value

A handle to the first exception in a list of exceptions.

Remarks

Use `UtaExcGetExceptionType()` and other `UtaExcGet...()` routines to get information about the exception from the handle.

When this function receives a list of exceptions, it zeroes the value reported by the `UtaExcGetErrorCount()` function.

Example

```
// Example can raise user-defined exceptions while action is doing
// tasks. Each exception has a severity level associated with it. Near
// the end of the action, a routine checks to see if exceptions
// occurred and receives them if they did. If the severity of an
// exception exceeds a threshold, a value of -1 is written to a
// parameter named "parm1" in the action's parameter block. If "parm1"
// is a reference to a symbol in a symbol table, actions in other tests
// can access the symbol table to see if this action raised one or more
// "serious" exceptions.
HUTAEXC hUtaException;
long lSeverity, lNumExceptions, lCounter;
char chMessage[40];
...(do something)
// action routine contains one or more routines to see if an
// exception condition exists
if (some condition == some value) // raise an exception?
   {
   strcpy (chMessage, "Condition raised an exception!");
   lSeverity = 10; // assign severity level to this exception
   UtaExcRaiseUserError(chMessage, lSeverity); // raise exception
   }
...(testing continues)
...
...(near end of testing)
if (UtaExcRegIsError()) // if exception(s) exist
   {
   lNumExceptions = UtaExcRegGetErrorCount(); // get # of exceptions
   // receive all exceptions & get handle to first in list
   hUtaException = UtaExcRegReceiveError();
   for (lCounter = 1; lCounter <= lNumExceptions; lCounter++)
      {
      if (UtaExcGetSeverity(hUtaException) > 5) // test severity
        UtaPbSetInt32(hParmBlock, "parm1", -1); // write to parm.
      if (lCounter < lNumExceptions)
        // get handle to next exception
        hUtaException = UtaExcGetNextError(hUtaException);
      };
   };
```

See also

UtaExcGetExceptionType()
various UtaExcGet...() functions

UtaExcGetNextError()

Given the handle to an exception, this function returns the handle to the next exception if more than one exception has been raised.

```
HUTAEXC UtaExcGetNextError(
    HUTAEXC hUtaException,                  // handle to an exception
    void *reserved UTACPP_DEFAULT(NULL)     // (not used)
    );
```

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

The handle to the next exception.

Remarks

Use function `UtaExcRegReceiveError()` to get a handle to the list of exceptions.

Example

```
// Example can raise user-defined exceptions while action is doing
// tasks. Each exception has a severity level associated with it. Near
// the end of the action, a routine checks to see if exceptions
// occurred and receives them if they did. If the severity of an
// exception exceeds a threshold, a value of -1 is written to a
// parameter named "parm1" in the action's parameter block. If "parm1"
// is a reference to a symbol in a symbol table, actions in other tests
// can access the symbol table to see if this action raised one or more
// "serious" exceptions.
HUTAEXC hUtaException;
long lSeverity, lNumExceptions, lCounter;
char chMessage[40];
...(do something)
// action routine contains one or more routines to see if an
// exception condition exists
if (some condition == some value) // raise an exception?
   {
   strcpy (chMessage, "Condition raised an exception!");
   lSeverity = 10; // assign severity level to this exception
   UtaExcRaiseUserError(chMessage, lSeverity); // raise exception
   }
...(testing continues)
...
...(near end of testing)
if (UtaExcRegIsError()) // if exception(s) exist
   {
   lNumExceptions = UtaExcRegGetErrorCount(); // get # of exceptions
   // receive all exceptions & get handle to first in list
   hUtaException = UtaExcRegReceiveError();
   for (lCounter = 1; lCounter <= lNumExceptions; lCounter++)
      {
      if (UtaExcGetSeverity(hUtaException) > 5) // test severity
        UtaPbSetInt32(hParmBlock, "parm1", -1); // write to parm.
      if (lCounter < lNumExceptions)
        // get handle to next exception
        hUtaException = UtaExcGetNextError(hUtaException);
      };
   };
```

See also

UtaExcRegReceiveError()

UtaExcGetErrorMessage()

This function returns a string containing the error message associated with an exception.

```
LPCSTR UtaExcGetErrorMessage(
    HUTAEXC hUtaException,                  // handle to an exception
    void *reserved UTACPP_DEFAULT(NULL)     // (not used)
    );
```

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

A pointer to a string containing an error message associated with an exception.

Example

```
// Example raises an exception & displays its associated error message
HUTAEXC hUtaException;
char chMessage [60];
long lSeverity;
...(do something)
// Raise an exception
if (some condition == some value) // test for some condition
   {
   strcpy (chMessage, "Test condition 1 raised an exception!");
   lSeverity = 1;
   UtaExcRaiseUserError(chMessage, lSeverity); // raise an exception
   };
...(do something)
hUtaException = UtaExcRegReceiveError(); // receive the exception
ErrorString = UtaExcGetErrorMessage(hUtaException); // get message
AfxMessageBox(ErrorString, MB_OK); // display the error message
```

UtaExcGetExceptionType()

This function returns the type (ID) of exception that occurred.

IDUTAEXC UtaExcGetExceptionType(

HUTAEXC *hUtaException*, // handle to an exception

void **reserved* **UTACPP_DEFAULT(NULL)** // (not used)

);

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

An ID that identifies the type of exception.

Remarks

You can use `UtaExcGetExceptionType()` for all exception types. Once you know which type of exception you have, use a `UtaExcGet...()` routine to get more information about the specific type of exception.

Example

```
// Although the example displays the exception's ID & cause in a
// message box, you probably would use the ID & cause to decide which
// kind of action to take to handle the exception
HUTAEXC hUtaException;
char chExceptionMessage[20];
long lID, lCause;
// raise a user-defined exception
UtaExcRaiseUserError("Exception was raised", 3);
// receive the exception & return the handle to it
hUtaException = UtaExcRegReceiveError();
// get the exception's ID & cause
lID = UtaExcGetExceptionType(hUtaException);
lCause = UtaExcGetExceptionCause(hUtaException);
// display the exception's ID & cause
sprintf(chExceptionMessage, "ID = %d & Cause = %d", lID, lCause);
AfxMessageBox(chExceptionMessage, MB_OK);
```

See also

various UtaExcGet...() functions

UtaExcGetCause()

This function returns the cause of an exception.

```
int UtaExcGetCause(
    HUTAEXC hUtaException,                  // handle to an exception
    void *reserved UTACPP_DEFAULT(NULL)     // (not used)
    );
```

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

An integer that returns the cause of the exception.

Remarks

Use `UtaExcGetExceptionType()` to determine the type of exception before using `UtaExcGetCause()`.

`UtaExcGetCause()` is valid only for UTA_CORE_EXCEPTION_TYPE, UTA_ARCHIVE_EXCEPTION_TYPE, UTA_FILE_EXCEPTION_TYPE, and UTA_INST_EXCEPTION_TYPE. It cannot be used for user-defined exceptions raised by `UtaExcRaiseUserError()`.

Example

```
// Although the example displays the exception’s ID & cause in a
// message box, you probably would use the ID & cause to decide which
// kind of action to take to handle the exception
HUTAEXC hUtaException;
char chExceptionMessage[20];
long lID, lCause;
// raise a user-defined exception
UtaExcRaiseUserError("Exception was raised", 3);
// receive the exception & return the handle to it
hUtaException = UtaExcRegReceiveError();
// get the exception's ID & cause
lID = UtaExcGetExceptionType(hUtaException);
lCause = UtaExcGetExceptionCause(hUtaException);
// display the exception's ID & cause
sprintf(chExceptionMessage, "ID = %d & Cause = %d", lID, lCause);
AfxMessageBox(chExceptionMessage, MB_OK);
```

See also

UtaExcGetExceptionType()

UtaExcGetSeverity()

This function returns the severity level that was set when the exception occurred.

```
int UtaExcGetSeverity(
    HUTAEXC hUtaException,                  // handle to an exception
    void *reserved UTACPP_DEFAULT(NULL)     // (not used)
    );
```

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

An integer that identifies the severity of an exception.

Remarks

This function is valid only for UTA_USER_EXCEPTION_TYPE.

Example

```
// Example can raise user-defined exceptions while action is doing
// tasks. Each exception has a severity level associated with it. Near
// the end of the action, a routine checks to see if exceptions
// occurred and receives them if they did. If the severity of an
// exception exceeds a threshold, a value of -1 is written to a
// parameter named "parm1" in the action's parameter block. If "parm1"
// is a reference to a symbol in a symbol table, actions in other tests
// can access the symbol table to see if this action raised one or more
// "serious" exceptions.
HUTAEXC hUtaException;
long lSeverity, lNumExceptions, lCounter;
char chMessage[40];
...(do something)
// action routine contains one or more routines to see if an
// exception condition exists
if (some condition == some value) // raise an exception?
   {
   strcpy (chMessage, "Condition raised an exception!");
   lSeverity = 10; // assign severity level to this exception
   UtaExcRaiseUserError(chMessage, lSeverity); // raise exception
   }
...(testing continues)
...
...(near end of testing)
if (UtaExcRegIsError()) // if exception(s) exist
   {
   lNumExceptions = UtaExcRegGetErrorCount(); // get # of exceptions
   // receive all exceptions & get handle to first in list
   hUtaException = UtaExcRegReceiveError();
   for (lCounter = 1; lCounter <= lNumExceptions; lCounter++)
      {
      if (UtaExcGetSeverity(hUtaException) > 5) // test severity
        UtaPbSetInt32(hParmBlock, "parm1", -1); // write to parm.
      if (lCounter < lNumExceptions)
        // get handle to next exception
        hUtaException = UtaExcGetNextError(hUtaException);
      };
   };
```

UtaExcGetOsError()

This function returns the value of an exception as defined in the operating system where the exception occurred.

```
LONG UtaExcGetOsError(
    HUTAEXC hUtaException,                  // handle to an exception
    void *reserved UTACPP_DEFAULT(NULL)     // (not used)
    );
```

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

A value associated with an exception in the operating system.

Remarks

This function is valid only for UTA_FILE_EXCEPTION_TYPE.

Example

```
// Although the example displays the exception’s value in a message
// box, you probably would use the value to decide which kind of
// action to take to handle the exception
HUTAEXC hUtaException;
char chExceptionMessage[20];
long lValue;
...(an exception occurs in the operating system)
// receive the exception & return the handle to it
hUtaException = UtaExcRegReceiveError();
// get the exception's value
lValue = UtaExcGetOSError(hUtaException);
// display the exception's value
sprintf(chExceptionMessage, "Exception value = %d, lValue);
AfxMessageBox(chExceptionMessage, MB_OK);
```

See also

UtaExcGetStatus()

UtaExcGetStatus()

This function returns a value corresponding to the status assocated with an exception.

int UtaExcGetStatus(

HUTAEXC *hUtaException*, // handle to an exception

void **reserved* **UTACPP_DEFAULT(NULL)** // (not used)

);

Parameters

hUtaException
The handle to an exception.

**reserved*
(not used)

Return Value

An integer that returns the status of an exception.

Remarks

`UtaExcGetStatus()` is valid only for UTA_OLE_EXCEPTION_TYPE.

Example

```
// Although the example displays the exception's status in a message
// box, you probably would use the value to decide which kind of
// action to take to handle the exception
HUTAEXC hUtaException;
char chExceptionMessage[20];
long lStatus;
...(an exception occurs)
// receive the exception & return the handle to it
hUtaException = UtaExcRegReceiveError();
// get the exception's value
lStatus = UtaExcGetStatus(hUtaException);
// display the exception's value
sprintf(chExceptionMessage, "Exception Status = %d, lStatus);
AfxMessageBox(chExceptionMessage, MB_OK);
```

See also

UtaExcGetOSError()

UtaExcRegDisplayErrors()

This routine will display all of the exceptions (errors) in a dialog box and clear all of the pending exceptions when the user presses the OK button in the dialog box.

void UtaExcRegDisplayErrors();

Parameters

(none)

Return Value

(none)

Remarks

Following `UtaExcRaiseUserError()` with `UtaExcRegDisplayError()` notifies the user that an exception was raised and then continues testing.

Example

```
// Example causes the following to display in a dialog box when
// encountered while testplan is executing:
//   Condition raised a user-defined exception! (Severity: 5)
// Testing continues when dialog box's OK button is pushed.
char chMessage [60];
long lSeverity;
...(do something)
if (some condition == some value) // raise an exception?
   {
   strcpy (chMessage, "Condition raised a user-defined exception!");
   lSeverity = 5;
   UtaExcRaiseUserError(chMessage, lSeverity);
   UtaExcRegDisplayErrors(); // show exception in a dialog box
   }
...(testing continues)
```

See also

UtaExcRaiseUserError()

Functions Used to Abort Testing

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

UtaKeepAlive()

This function is used to keep the windowing system "alive" when you write actions that take a long time to complete. It also allows an abort condition and other such windowing actions to be responded to while you wait.

```
BOOL UtaKeepAlive ();
```

Parameters

(none)

Return Value

TRUE if an operator abort condition exists.

Remarks

This function can be useful during extended periods when HP TestExec SL's user interface is unresponsive, such as when making lengthy measurements. Under such conditions, it may not be obvious whether the action is simply taking a long time to complete or if it is "hung." Invoking this function processes events in the event queue immediately instead of waiting for the normal interval to elapse before they are processed.

Be aware that this function can affect the timing of action code. When timing is critical, we recommend that you do not use this function but instead settle for less responsiveness from HP TestExec SL's user interface while your action code executes.

Example

```
// Example uses UtaKeepAlive() to "pump" the event queue during a
// lengthy measurement operation. When it is invoked, UtaKeepAlive()
// also tests for the presence of an operator abort condition. If such
// a condition exists, an exception is raised.
// Following function called by a button in the operator interface
UtaSetOperatorAbort(); // operator pressed the abort key
// Following lines are in your action routine
int nIndex;
// some routine that takes a long time to finish
for (nIndex = 1; nIndex <= 100000; nIndex++)
   {
   ...(code that takes a reading & stores it in array element whose
   ...index is nCounter)
   // process message queue & see if operator abort key was pressed
   if (UtaKeepAlive())
      {
      ...(code to raise an exception)
      };
   };
```

See also

UtaSetOperatorAbort()

UtaIsOperatorAbort()

This functions tests to see if an operator has pressed the "Abort" key in the operator interface; i.e., it tests for the presence of an abort condition generated by the `UtaSetOperatorAbort()` function.

BOOL UtaIsOperatorAbort();

Parameters

(none)

Return Value

TRUE if an operator abort condition exists.

Example

```
// Simple example creates an abort, acknowledges it & clears condition
// Following function called by a button in the operator interface
UtaSetOperatorAbort(); // operator pressed the abort key
// Following lines are in your action routine
if (UtaIsOperatorAbort()) // test for "abort" condition
  {
  // Next line invokes a dialog box but in reality you probably
  // would take some other action, such as shut down power supplies
  AfxMessageBox ("Abort key was pressed!", MB_OK);
  UtaClearOperatorAbort(); // clear the "abort" condition
  };
```

See also

UtaSetOperatorAbort()

UtaSetOperatorAbort()

This function can be called from an operator interface, such as by pushing an "Abort" button, to indicate the operator wishes to abort testing.

void UtaSetOperatorAbort(

BOOL *bAbort* **UTACPP_DEFAULT(TRUE)** // abort/no abort

);

Parameters

bAbort
TRUE indicates an abort condition is desired.

Return Value

(none)

Example

```
// Simple example creates an abort, acknowledges it & clears condition
// Following function called by a button in the operator interface
UtaSetOperatorAbort(); // operator pressed the abort key
// Following lines are in your action routine
if (UtaIsOperatorAbort()) // test for "abort" condition
   {
   // Next line invokes a dialog box but in reality you probably
   // would take some other action, such as shut down power supplies
   AfxMessageBox ("Abort key was pressed!", MB_OK);
   UtaClearOperatorAbort(); // clear the "abort" condition
   };
```

See also

UtaIsOperatorAbort()

UtaClearOperatorAbort()

This function clears an operator abort condition generated by the `UtaSetOperatorAbort()` function.

void UtaClearOperatorAbort();

Parameters

(none)

Return Value

(none)

Example

```
// Simple example creates an abort, acknowledges it & clears condition
// Following function called by a button in the operator interface
UtaSetOperatorAbort(); // operator pressed the abort key
// Following lines are in your action routine
if (UtaIsOperatorAbort()) // test for "abort" condition
  {
  // Next line invokes a dialog box but in reality you probably
  // would take some other action, such as shut down power supplies
  AfxMessageBox ("Abort key was pressed!", MB_OK);
  UtaClearOperatorAbort(); // clear the "abort" condition
  };
```

See also

UtaSetOperatorAbort()

5

The Runtime API Reference

This chapter describes the Runtime API, whose functions let you replace the default user interface for operators with a custom interface.

For more information, see Chapter 1 in the *Customizing HP TestExec SL* book.

Functions for Registering a Personality

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

InitializeUserModule()

WORD InitializeUserModule(HWND *ParentWnd***)**

This function will be called by the Test Executive after the user-defined module code is loaded. The user code now has an opportunity to perform any initialization it requires, such as opening a serial port to an automation handler and registering for any event callbacks of interest to it.

The ParentWnd parameter is the handle of the application's main window. The Test Executive main application is implemented in Visual C++. If you are implementing your user interface DLL in Visual C++ (the recommended method) you will probably want to call CWnd::FromHandle(ParentWnd) to get a reference to the CWnd of the main app.

The user will normally be implementing a user interface personality in this extension module. The user interface will typically be implemented as a dialog or form window. In this init call, the user must take responsibility for correctly initializing his application environment and bringing up the user interface form. The mechanics of doing are potentially unique to each implementation

The user has the option of implementing the loadable personality as a normal (modeless) form or dialog or as a modal dialog. The return conditions are slightly different in each case. A form or non-modal dialog will create its window and return VOK. This signifies that the personality is posted and active and the Test Executive can return to the idle loop.

A modal dialog will take over control of the Windows event loop and not return until the user exits the dialog. When it exits, it must return VUIComplete or VExitRequest (assuming no errors). VUIComplete signifies the personality has exited normally. The Test Executive will

immediately begin a new login sequence. A return of VExitRequest indicated the user has requested the termination of the program. The Test Executive will unload and exit.

Any return code other than VOK, VUIComplete, or VExitRequest will be considered an error condition. these return constants are defined in the header file “pubapi.h”. This is the standard include file for the C or C++ language bindings of the runtime API.

ShutdownUserModule()

void ShutdownUserModule(*void*)

Either the application is exiting or this user module is being unloaded manually by the Test Executive. The module should clean up all resources it has allocated. In particular, it must unregister all callbacks it set at its registration time.

Any user interface windows created by this module should be destroyed here.

Functions for Controlling the State of the Test Executive

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VContinueSequence()

WORD VContinueSequence(*void*)

This will resume the current testplan session if the sequencer is stopped in a continuable state (system state is *Sequencing*). If not running, this will do a run rather than a continue. This would be used by a user interface that provides a Pause button. The VContinueSequence() function would be the proper way to resume execution following the pause.

The return code is a status state for the sequencer. The status codes (see file “pubapi.h”) are:

SEQ_HALT_NO_ERROR	Normal halt condition.
SEQ_BREAKPOINT_HALT	Sequencer has paused at a breakpoint.
SEQ_STEP_HALT	Sequencer has paused after executing one statement in response to a VStepSequence() call.
SEQ_USER_STOP	The sequencer halted in response to a VStopSequence() request.
SEQ_USER_PAUSE	The sequencer halted in response to a VPauseSequence() request.
SEQ_LIMIT_ERROR	The sequencer has halted because of a test limit error. This halt will not happen until the failure limit count is reached.
SEQ_EXCEPTION_HALT	An uncaught exception was registered and there was no abort sequence to shutdown the system. If an abort sequence exists, execution will trap to the first statement in it and the sequencer will continue.
SEQ_PRERUN_ERROR	The testplan could not be prerun. If this code is returned the sequencer never started; therefore, it never entered the *Sequencing* state.

Control does not return until the sequencer halts again.

VLoadTestplan()

WORD VLoadTestplan(LPCSTR *path*)

Request loading the testplan indicated in the path string. The path must be fully specified because no search strategy is used. If the requested path is an empty string (""), an empty testplan will be created. User personalities should never create a new, empty testplan. There is currently no documented way to edit the empty testplan from a user personality.

If the system is in the *TestplanLoaded* state and this function call is made requesting a different testplan (that is, the paths do not match), the current testplan is unloaded (as in VUnloadTestplan) and the new one is loaded.

Successful completion of this operation brings the system to the *TestplanLoaded* state, the testplan name placed in the TestplanName slot in the SystemTable, and the system event AdviseTestplanLoaded sent. This function returns TRUE on success.

VPauseSequence()

WORD VPauseSequence(*void***)**

This is a request to bring the sequencer to a paused state. The paused state is a suspension of execution in a restartable state. The system was **not** brought to a safe—i.e., cleaned up and unpowered—state.

If the sequencer is currently running (system state is the *Running* state or higher), a pause request will be noted and the call will return. The sequencer is still running during the execution of this function. The pause will be honored after the sequencer completes the current test statement.

This call allows the user interface to implement a Pause button.

VRunSequence()

WORD VRunSequence(HVSEQ *seq***)**

This is the main execution control API. A personality calls this function to request the sequencer to begin a testplan session. The sequencer will be active until it has reached a halt condition. Note that the sequencer could be in a “loop forever” mode.

The parameter *seq* is currently not used. It will allow the personality to run any test sequence in the testplan. The sequence named “Testmain” will always be run.

If the system is in the *TestplanLoaded* state, the system state will go to *Sequencing* and then automatically on to *Running*. After that, the state will fluctuate among *Running, TestExecuting,* and *Reporting* until it finally exits via *Sequencing* to TestplanLoaded. All event transition notifications will be sent as states change.

This function does not return until the sequencer halts. Even though the execution thread is stuck in the sequencer for the duration of the testplan session, the Windows message queue will be polled at various points. This lets the user interface remain (mostly) alive; for example, the user interface could still respond to a press of a Stop button. The stop will be handled as a Windows event. The response of the personality to the Stop button event would include making a call to VStopSequence() that will condition the sequencer to shut down at the end of the current test statement. Control will then return to the caller with a code indicating the sequencer stopped in response to a stop request.

The sequencer can stop in either a halted state or a continuable state. Halted implies that all required cleanups have been completed and the UUT is powered down to a safe state. Continuable implies that sequencing is temporarily suspended. In this case, there is the potential of dangerous voltage or current conditions on the UUT.

The return code is an exit state for the sequencer. The exit codes (see file "pubapi.h") are:

SEQ_HALT_NO_ERROR	Normal halt condition.
SEQ_BREAKPOINT_HALT	Sequencer has paused at a breakpoint.
SEQ_STEP_HALT	Sequencer has paused after executing one statement in response to a VStepSequence() call.
SEQ_USER_STOP	The sequencer halted in response to a VStopSequence() request.
SEQ_USER_PAUSE	The sequencer halted in response to a VPauseSequence() request.
SEQ_LIMIT_ERROR	The sequencer has halted because of a test limits error. This halt will not happen until the failure limit count is reached.
SEQ_EXCEPTION_HALT	An uncaught exception was registered and there was no abort sequence to shut down the system. If an abort sequence exists, execution will trap to the first statement in it and the sequencer will continue.
SEQ_PRERUN_ERROR	The testplan could not be prerun. If this code is returned the sequencer never started; therefore, it never entered the *Sequencing* state.

VStepSequence()

WORD VStepSequence(*void***)**

If the sequencer is in a continuable state (*Sequencing* state), this will request the sequencer to run through the next test and pause again. Normal sequencer behavior is followed; i.e., looping and halt counts will be honored.

A system integrator could use this to build a troubleshooter interface that allowed the technician to step through the testplan.

VStopSequence()

WORD VStopSequence(*void*)

This is a request to bring the sequencer to a halted state.

If the sequencer is currently running (system state is the *Running* state or higher), a stop request will be noted and the call will return. The halt will be honored after the sequencer completes the current test statement. The sequencer is still running during the execution of this function. If the sequencer is paused in a continuable state (system state is *Sequencing*), any required cleanups will be run immediately and the system returned to the *TestplanLoaded* state.

This call would be used by a user interface to implement a Stop button.

VUnloadTestplan()

WORD VUnloadTestplan(*void*)

Attempt to unload the current testplan.

Successful completion of this request will bring the system to the *Empty* state and the TestplanName slot in the SystemTable set to "". AdviseTestplanUnloaded() will be sent. This function returns TRUE on success.

Functions for Miscellaneous Server Requests

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VAppExit()

WORD VRequestLogin(*void*)

Close the current testplan and exit HP TestExec SL. Returns FALSE if unsuccessful and TRUE (which never is seen because the application is exited) if successful.

VClearReport()

void VClearReport(*void*)

Inform all registered report windows that they should erase their contents. This will usually be done at the beginning of a test run (*Prerun* to *Sequencing* state transition).

VClearTrace()

void VClearTrace(*void*)

Inform all registered trace windows that they should erase their contents. This will usually be done at the beginning of a test run (*Prerun* to *Sequencing* state transition).

VGetCountedLoops()

LONG VPUBAPI VGetCountedLoops(*void*)

Returns the upper boundary for how many times the testplan should repeat to complete one "run," such as a run requested by `VRunSequence()`.

VGetLoopMode()

WORD VPUBAPI VGetLoopMode(*void***)**

Returns a value that indicates whether the current looping mode is by count (1) or by time (2). If this API function cannot return a valid value—for example, if no testplan is loaded—it returns 0.

VGetTestExecutable()

BOOL VPUBAPI VGetTestExecutable(HVTEST *aTest***)**

Returns TRUE if the test is enabled for execution, or FALSE if it is not.

VGetTestSkip()

BOOL VPUBAPI VGetTestSkip(HVTEST *aTest***)**

Returns TRUE if the test is marked to be skipped, or FALSE it is not.

VGetTimedLoops()

BOOL VPUBAPI VGetTimedLoops(LONG **lDays***, WORD** **nHours***, WORD** **nMinutes***, WORD** **nSeconds***)**

Returns the minimum time specificed for how long the test should run.

VRequestLogin()

void VRequestLogin(*void***)**

Post a login request to be processed after the application returns to the idle state. This is used by a non-modal user interface when it wants to close itself and begin a new user login sequence. The user interface code should do the VRequestLogin() call in its close code, usually after it has destroyed its window and just before it takes the return that will leave the user interface for good. Some system integrators may choose to have their user interfaces exit the entire application instead of returning to the login. In this case, call VAppExit() instead.

VSendReportMsg()

void VSendReportMsg(LPCSTR *msg***)**

Copy the preformatted message string to each registered report window. The message formatting should be handled by a VRegisterTestReport callback. This function will normally be called by the VRegisterTestReport callback.

VSendTraceMsg()

void VSendTraceMsg(LPCSTR *msg***)**

Copy the preformatted message string to each registered trace window. The message formatting should be done by whoever is initiating the trace call. This may be user code or low level I/O handlers.

VSendUserDefinedMsg()

void VPUBAPI VSendUserDefinedMsg(LONG *ID***, LPCSTR** *msg***)**

Broadcasts a message, *msg*, to all potential listeners, and an integer, *ID*, that identifies the type of message being sent. Listeners must parse messages according to their *ID*s to decide which messages are meant for them. Does not wait for a response. The memory used by *msg* may be reclaimed after this call returns.

VSendUserDefinedQuery()

BOOL VPUBAPI VSendUserDefinedQuery(LONG *IDSent***, LPCSTR** *msgSent***, LONG** **IDRet***,* **LPCSTR** **msgRet***,* **double** *secsTimeout***)**

Broadcasts a message, *msgSent*, to all potential listeners, and an integer, *ID*, that identifies the type of message being sent. Waits a specified number of seconds, *secsTimeout*, for a response via **IDRet* and **msgRet*.

VSendUserDefinedResponse()

void VPUBAPI VSendUserDefinedResponse(LONG *ID***, LPCSTR** *msg***)**

Broadcasts a message, *msg*, to a specific listener who is awaiting a response, and an integer, *ID*, that identifies the type of message being sent.

VSetVariant()

WORD VSetVariant(LPCSTR *name*)

Set the named variant as the current system variant. Return 0 if the named variant is not defined in the current testplan; otherwise return non-zero. See VCreateVariantNameList() to enumerate the available variants.

Functions for Callback Registration

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VRegisterTestplanLoaded()

WORD VRegisterTestplanLoaded(*void* **(*callback)(LPCSTR))**
void VUnregisterTestplanLoaded(*void* **(*callback)(LPCSTR))**

This callback will be made when a new testplan has been successfully loaded into memory. Name contains the name of the testplan. Each listener registered on the VRegisterTestplanLoaded() callback will be notified. This event is a side effect of a call to VLoadTestplan().

Use VUnregisterTestplanLoaded() to remove yourself from this callback chain.

VRegisterTestplanUnloaded()

WORD VRegisterTestplanUnloaded(*void* **(*callback)(***void***))**
void VUnregisterTestplanUnloaded(*void* **(*callback)(***void***))**

The current testplan has just been removed. Name contains the name of the testplan. This event is a side effect of a call to VUnloadTestplan(). It may also be generated by a VLoadTestplan() call if another testplan is currently loaded.

Use VUnregisterTestplanUnloaded() to remove yourself from this callback chain.

VRegisterIdlePoll()

WORD VRegisterIdlePoll(*void* (*callback)(WORD))
void VUnregisterIdlePoll(*void* (*callback)(WORD))

This callback will be entered on an on-going basis as long as the system is in an idle state. Its purpose is to "pump" the automation personality—that is to support its polling of activity from the automation handler equipment.

The automation personality must be able to respond to events initiated by handler systems external to the Test Executive or the sequencer. These handlers will often use the serial interface or switch closures to control the tester. The automation handler must have the same ability to run a testplan as a user would from the operator interface. The problem, though is that since Windows is not multi-tasking, no built in mechanism exists to support these externally initiated events. Windows events support a pseudo-tasking approach, but they work because the keyboard and mouse are deeply tied into the interrupt system of the PC hardware.

The best strategy is to use the existing paradigm of Windows idle loop processing. This gives an application the ability to be continually activated to monitor external events. The Test Executive enters the idle loop whenever it is not otherwise busy processing test data. In the idle loop, we will repeatedly call all callback functions registered as a VRegisterIdlePoll() listener. This callback does not need to return a value. Its responsibility is to detect the external state transitions required and call the state transition requests to act on them.

The Automation and Operator interfaces have complete freedom to do anything required to wait for, poll for, or be told about the module placement. Any required hardware or interface may be used to handshake with the handlers or user.

A typical scenario is for the Operator interface to poll for a barcode when the system is in an idle state. It would wait for a string coming in a serial port. On receipt of the barcode it will, among other things, call the VRunSequence() API to begin the testplan run.

The parameter supplied to the callback function is the current test executive state (Vstate). Cast it back to a VState enum value. (This is legal but "undefined". It should work fine since there is a unique mapping between the enum and an int.) If you are using this callback to do automation polling,

you usually want to skip the polling if the state is >= VSequencingState. In other words, if the sequencer is running, don't look for a new start event.

The normal use of this callback is just to poll for work to do. The AdviseIdlePoll callback should ordinarily do a quick check of the handler state and return. The system integrator should never put a while loop in the IdlePoll callback to wait for an event to happen because that will defeat the Windows event loop mechanisms. The user poll callback must quickly exit back to the idle loop to allow normal system activity to continue. The callback will be repeatedly called as long as the system is idle.

If the user interface is implemented as a modal dialog, it will retain control of the idle loop when it is active and the idle polling will not happen. The system integrator must then take responsibility for performing any polling required for their application.

Use VUnregisterIdlePoll() to remove yourself from this callback chain.

VRegisterSequenceBegin()

WORD VRegisterSequenceBegin(*void* **(*callback)(***void***))**
void VUnregisterSequenceBegin(*void* **(*callback)(***void***))**

Notification that the sequencer has entered its active state. The sequencer is considered active as long as it is pursuing a testplan run goal. No tests are executing at this point If the user pauses the testplan (as opposed to halting) it is still considered active.

This strange wording is to emphasize that, at a conceptual level, we separate the process of running through a testplan once from the process of deciding how many times to rerun the testplan. The user may describe to the sequencer both the condition that will cause the test process to halt and the desired length or number of times to run.

Use VUnregisterSequenceBegin() to remove yourself from this callback chain.

VRegisterSequenceEnd()

WORD VRegisterSequenceEnd(*void* **(*callback)(LONG))**
void VUnregisterSequenceEnd(*void* **(*callback)(LONG))**

This is the halt condition. The sequencer has left the active state. It is no longer pursuing a goal of running tests, but the testplan is still runnable. The status code is one of a predefined list of codes describing the reason for halting the sequence. The codes are defined in pubapi.h.

A paused sequencer does not transition down through this state. If the sequencer pauses, it is non Running but it is still Sequencing. A VStopSequence() call to a paused sequence will terminate the sequencing state and notify this transition.

Use VUnregisterSequenceEnd() to remove yourself from this callback chain.

VRegisterRunningBegin()

WORD VRegisterRunningBegin(*void* **(*callback)(LONG))**
void VUnregisterRunningBegin(*void* **(*callback)(LONG))**

The sequencer has entered the state of actively executing a testplan. It can be assumed that tests are being executed. If the interface developer wishes to color a status indicator when the tester is running or measure testplan run time, this is the event that should initiate it.

The count is a running count of the number of times the testplan has been reinvoked in this sequencing session. If the sequencer is running in a looping mode, the testplan may be reexecuted many times. The count is reset (a session ends) whenever the system makes the transition from the Sequencing state to the TestplanLoaded state. This is basically whenever the sequencer halts.

Use VUnregisterRunningBegin() to remove yourself from this callback chain.

VRegisterRunningEnd()

WORD VRegisterRunningEnd(*void* **(*callback)(LONG))**
void VUnregisterRunningEnd(*void* **(*callback)(LONG))**

The sequencer is leaving the state of actively executing a testplan. It can be assumed that no tests are currently being executed. The sequencer is still in an active state.

Use VUnregisterRunningEnd() to remove yourself from this callback chain.

VRegisterTestBegin()

WORD VRegisterTestBegin(*void* **(*callback)(const** *HVTEST***))**
void VUnregisterTestBegin(*void* **(*callback)(const** *HVTEST***))**

A test is about to be executed. The handle passed in identifies the test and can be used to inquire about its state information.

This might be used by a personality to post an activity indicator as test procedures are executed. It might also be used in conjunction with VRegisterTestEnd() to time test execution times.

Use VUnregisterTestBegin() to remove yourself from this callback chain.

VRegisterTestEnd()

WORD VRegisterTestEnd(*void* **(*callback)(const** *HVTEST***))**
void VUnregisterTestEnd(*void* **(*callback)(const** *HVTEST***))**

The end of a test execution. This would be used to bracket the VRegisterTestBegin() activity.

Use VUnregisterTestEnd() to remove yourself from this callback chain.

VRegisterTestReport()

WORD VRegisterTestReport(*void* **(*callback)(const** *HVTEST***))**
void VUnregisterTestReport(*void* **(*callback)(const** *HVTEST***))**

Each personality has the ability to control the format, destination, and even the existence of failure reporting. Reporting is independent of data logging.

This callback will come to a personality following each test stmt, whether it passes or fails. The HVTEST is a handle to a test statement. It contains most of the information required for reporting. Through the handle and the API interfaces, the user code can determine the test name, the pass/fail status, the measured result, the limits, even the number of times the test has been executed since this session began. The user code may use any of this data, format it as desired, and trace it or print it according to their local policy.

Use VUnregisterTestReport() to remove yourself from this callback chain.

VRegisterVariantChange()

WORD VRegisterVariantChange(*void* **(*callback)(LPCSTR))**
void VUnregisterVariantChange(*void* **(*callback)(LPCSTR))**

Notification to the personality that a new variant value has been set. The personality might want to display the current variant value. It might even want to take some special action depending on the particular variant value.

Use VUnregisterVariantChange() to remove yourself from this callback chain.

VRegisterUserDefinedMsg()

WORD VRegisterUserDefinedMsg(*void* **(*callback)(LONG, LPCSTR))**
void VUnregisterUserDefinedMsg(*void* **(*callback)(LONG, LPCSTR))**

This supplies a message to the user interface. The message comes from a user action routine. If no one receives it, it is discarded. The placement and form of the message text is determined by the user interface/system integrator.

Do not assume msg is persistent! Copy it if you need to save it.

The tag parameter is available to give more flexibility. It is just a parameter that is passed along. A customer may define it as they wish. One site may wish to use it as a line number to allow multiple lines of message on the screen at one time. Another user may define it as a severity tag to guide the interface in how to render and display the message.

This message is supplementary to the normal test reporting initiated by VRegisterTestReport(). Only user interfaces who register to receive VRegisterUserDefinedMsg() callbacks will get the message.

This event is broadcast to all registered listeners in response to the VSendUserDefinedMsg() API call.

Use VUnregisterUserDefinedMsg() to remove yourself from this callback chain.

VRegisterReportClear()

WORD VRegisterReportClear(*void* **(*callback)(***void***))**
void VUnregisterReportClear(*void* **(*callback)(***void***))**

This event results from a call to the API function VClearReport().

Advise personalities that old contents of the report window (if it exists) are not longer needed. This would typically be done at the start of a test sequence (VRunSequence() call).

The report output may not be displayed in a user window. It may be directed to a file or printer. In that case this message would be a good trigger to flush the file or page eject the printer.

Use VUnregisterReportClear () to remove yourself from this callback chain.

VRegisterSendReportMsg()

WORD VRegisterSendReportMsg(*void* **(*callback)(LPCSTR))**
void VUnregisterSendReportMsg(*void* **(*callback)(LPCSTR))**

This event is initiated by a call to VSendReportMsg().

This supplies a message to the user interface with a request to place it on the report output stream. If no one receives it, it is discarded. The placement and form of the message text is determined by the user interface/system integrator.

Do not assume msg is persistent! Copy it if you need to save it.

Use VUnregisterSendReportMsg() to remove yourself from this callback chain.

VRegisterTraceClear()

WORD VRegisterTraceClear(*void* **(*callback)(***void***))**
void VUnregisterTraceClear(*void* **(*callback)(***void***))**

Advise personalities that old contents of the trace window (if it exists) are not longer needed. This would typically be done at the start of a test sequence (VRunSequence() call). The event is initiated by a call to VClearTrace().

Use VUnregisterTraceClear () to remove yourself from this callback chain.

VRegisterSendTraceMsg()

WORD VRegisterSendTraceMsg(*void* **(*callback)(LPCSTR))**
void VUnregisterSendTraceMsg(*void* **(*callback)(LPCSTR)**

This event is initiated when the VSendTraceMsg() call is made.

This supplies a message to the user interface with a request to place it on the trace output stream. The trace output should be considered an optional component. If no one receives it, it is discarded. The placement and form of the message text is determined by the user interface/system integrator.

Do not assume msg is persistent! Copy it if you need to save it.

Use VUnregisterSendTraceMsg() to remove yourself from this callback chain.

Functions for Halting the Test Sequencer

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VConfigureHaltOnFailure()

WORD VConfigureHaltOnFailure(LONG *failCount*)

Select the HaltOnFailure mode. This is the default mode and the mode that will typically be used for production testing. The parameter failCount is a number specifying the exact number of failures to allow before halting.

The definition of this mode is that the sequencer will attempt to execute the testplan in its entirety. The first failCount-1 failures will be ignored. On the failCountth failure the sequencer will execute any pending testgroup cleanups and halt. By honoring the testgroup cleanups, the system will attempt to being the system into a safe state and the UUT in an unpowered safe state.

Returns TRUE if failCount < 0.

VConfigureNoHalt()

WORD VConfigureNoHalt()

Override all halting controls. Run the testplan to completion regardless of errors.

Returns TRUE.

VConfigurePauseOnFailure()

WORD VConfigurePauseOnFailure(LONG *failCount*)

Select the PauseOnFailure mode. The parameter failCount is a number specifying the exact number of failures to allow before pausing.

The definition of this mode is that the sequencer will attempt to execute the testplan in its entirety. The first failCount-1 failures will be ignored. On the failCountth failure the sequencer will pause; that is, cease executing further tests. The failing test will have completed, that is it will have done its reporting, datalogging, and local cleanup. The system will likely still be in a powered state. This mode is used for module troubleshooting and testplan debugging. Since the testplan is paused in the vicinity of the failure and the overall system state is still intact, the system state can be examined (externally) to help determine the failure cause.

Note

This capability may be removed because of UUT and/or system damage or operator safety considerations.

Returns TRUE if failCount < 0.

VGetFailCountLimit()

LONG VGetFailCountLimit(*void*)

Return the current value of the sequencer terminal error count. This it the total count of failing tests that is specified by VConfigureHaltOnFailure() or VConfigurePauseOnFailure().

VGetHaltMode()

WORD VGetHaltMode(*void*)

This function returns the current sequencer halt mode. The return codes are defined in the header file in the section commented by “Sequencer return codes.” Current values returned are SEQ_HALT, SEQ_PAUSE, and SEQ_IGNORE.

Functions for Causing the Test Sequencer to Repeat

Note If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VConfigureCountedLoops()

WORD VConfigureCountedLoops(LONG *maxLoops***)**

Set the sequencer to repeat the testplan up to maxLoops times. This is an upper limit. The major halt mode setting will be honored, so the only way to guarantee that the testplan will loop this many times is to set the major mode to VConfigureNoHalt.

A typical use would be to set the halt mode to halt on some max number of failures and also set this control to loop the testplan a lot of times. The test will then halt on the failCount limit being reached or the maxLoops being reached, whichever comes first.

Returns TRUE if maxLoops is > 0.

During normal production testing, the setting would be VConfigureCountedLoops(1).

Note This function and VConfigureTimedLoops() are mutually exclusive; i.e., you cannot simultaneously use counted loops and timed loops.

VConfigureTimedLoops()

WORD VConfigureTimedLoops(LONG *days***, WORD** *hours***, WORD** *minutes***, WORD** *seconds***)**

Set the sequencer to repeat the testplan for at least this long. This is a lower limit. The major halt mode setting will be honored, so the only way to guarantee that the testplan will loop this long is to set the major mode to

VConfigureNoHalt. After each pass of the testplan, if the failCount limit has not been reached, the time will be checked. If the elapsed time is less than the required minTime, the testplan will be executed once more. The effect will be to run the testplan for at least this long, not up to this long or exactly this long.

Returns TRUE if the setting was accepted.

Note

This function and VConfigureCountedLoops() are mutually exclusive; i.e., you cannot simultaneously use counted loops and timed loops.

Functions for Interacting with System Data

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VGetFixtureID()

WORD VGetFixtureID(*void*)

Return the fixture code of the mass interconnect currently connected to the system. Return -1 if there is not one connected.

VGetTestplanName()

LPCSTR VGetTestplanName(*void*)

Return a string containing the name of the currently loaded testplan, or NULL if no testplan is loaded.

VGetTestName()

LPCSTR VGetTestName(*HVTEST*)

Return a string containing the name of the currently loaded test, or NULL if no test is loaded.

VGetTestText()

LPCSTR VGetTestText(*HVTEST*)

Returns a copy of the text presented for this statement by the testplan editor.

VGetTestRunCount()

LONG VGetTestRunCount(*HVTEST***)**

Return a count of the number of times this test has been executed since the last count reset.

VGetTestPassCount()

LONG VGetTestPassCount(*HVTEST***)**

Return a count of the number of times this test has passed since the last count reset.

VGetTestFailCount()

LONG VGetTestFailCount(*HVTEST***)**

Return a count of the number of times this test has failed since the last count reset.

VResetRunFlags()

void VResetRunFlags(*void***)**

Reset the run, pass, and fail counts to 0. This is automatically done each time a new testplan is loaded.

VTestJudgment()

WORD VTestJudgment(*HVTEST***)**

Returns the judgment code currently stored in the test statement identified by the handle. The judgment codes are:

0 = pass

>0 = fail

-1 = not run

A test with no limit check returns 0 for the judgment. Judgments are reset to -1 each time the user calls VRunSequence().

VGetResult()

BOOL VGetResult(*HVTEST***,** *HUTADATA**)

When passed the handle to a test, returns via a pointer the last measured result stored in the test identified by the handle. A test without limits checking returns NULL. The Boolean return variable's value is True if the data returned via the pointer is valid, and False if it is not.

VFindTest()

HVTEST VFindTest(*LPCSTR***)**

Look up the test by name. Returns a handle to the test statement, or NULL if not found.

VGetTestNameArraySize()

WORD VGetTestNameArraySize(HVSEQ *hSeq=NULL***)**

Return a count of the total number of test names in the current testplan. The hSeq parameter is not currently used and is ignored. This function returns 0 if no testplan is loaded.

VGetTestNameAt()

LPCSTR VGetTestNameAt(WORD index, *HVSEQ hSeq=NULL***)**

The test names are treated as a zero-based array of LPCSTR parameters. The number of elements in the array is defined by VGetTestNameArraySize(). This function is an array accessor that returns the name of the the test at array position "index" (0 <= index < VGetTestNameArraySize()). The hSeq parameter is not currently used and is ignored.

This is provided for the convenience of system integrators who might wish to post a list of available tests as part of a troubleshooter personality.

VGetVariantNameArraySize()

WORD VGetVariantNameArraySize(*void***)**

Return a count of the total number of test names in the current testplan. This function returns 0 if no testplan is loaded.

VGetVariantNameAt()

LPCSTR VGetVariantNameAt(WORD *index***)**

The test names are treated as a zero-based array of LPCSTR parameters. The number of elements in the array is defined by VGetVariantNameArraySize(). This function is an array accessor that returns the name of the the variant at array position “index” (0 <= index < VGetVariantNameArraySize()).

This is provided for the convenience of system integrators who might wish to post a list of available variants to allow an operator to configure a testplan for a particular functional test mode, such as Hot or QA.

VRunTest()

WORD VRunTest(*HVTEST***)**

Provides a means for an ambitious personality to directly control the selection and execution of tests. The return code is the test judgment, the same value returned by VTestJudgment(). Yes, you could write your own sequencer, but don’t without a real good reason. You lose a lot of the state tracking and other services the sequencer provides.

VIsPermitted()

WORD VIsPermitted(LPCSTR *resourceName***, LPCSTR** *operation***)**

This is the publicly supported security filter for user-defined personalities. ResourceName and operation parameters identify a resource and access type defined in the security file. A certain set of these security combinations will be predefined. The user may create new entries. The personality should check for permission before doing anything that alters the system state. Nothing enforces the security checks. At a later time the security checks may

migrate into the state transitions. We do not do that now because there is no strong precedent established for the number, names, and semantics of the levels.

Functions for Controlling Datalogging

Note

If you prefer a list of functions with page numbers, see the table of contents. If you already know which function you need, you can look in the index to find its page number.

VConfigureLogDirectory()

void VConfigureLogDirectory(LPCSTR *newDir***)**

Set a new datalogging directory path. The path must be valid. That is, all directories in the path must exist. The leaf directory must be writable. This setting will apply to the NEXT run of the testplan.

VGetLogDirectory()

LPCSTR VGetLogDirectory(*void***)**

Return the path to the directory to be used for datalogging output.

Index

G

H

I

M

P

R

S

T

U

V

W