MacOS Instrumentation Programmer’s Guide

Table of Contents

Introduction

The instrumentation system provides services that allow you to record information about running software. This can be used to expose system behavior, and is particularly useful for investigating runtime performance.

This guide describes the instrumentation services, how they are organized, and how they may be used to record instrumentation information from within your programs.

It is intended for anyone who wants to be able to produce runtime information from within their code, and later analyze that information in order to understand how the code is operating.

In order to use the instrumentation system, you should be generally familiar with building and running Macintosh programs. You should also be familiar with the `printf()` formatted output mechanism, described in Appendix B of *The C Programming Language, Second Edition,* by Kernighan & Ritchie.

This guide begins by describing instrumentation points in general and the various types in particular. It then discusses how to create these points and write instrumentation data to them. Finally, it documents the programming interface to the instrumentation system.

About the Instrumentation System

This section describes the types of instrumentation provided, instrumentation classes, the Instrumentation Tree, and the various forms that instrumentation data can take.

Generally speaking, there are two types of instrumentation data that you can record: trace events, and statistics. Trace events, also called traces, are used to indicate that a particular event occurred at a particular time. Every time your software logs a trace event, a trace event record is created and will be written out to permanent storage.

Traces are typically used to track the program flow of control – such as function entry and exit – and to determine the breakdown of execution time among various components.

Statistics represent the various values or totals that your software can measure over time.

Statistics require much less overhead than traces, because rather than having to record every call, the instrumentation system must only keep track of the current values for each statistic. These values are periodically sampled and written to permanent storage.

The trade-off is that there is no record of when a particular statistic value was set by your software; its timestamp refers to when it was sampled by the instrumentation system.

Your code may produce both types of instrumentation data – and call most instrumentation routines – at any interrupt level. Exceptions are noted in the “Instrumentation Reference” section.

Instrumentation Classes

In order to record traces for a particular event or to define a particular statistic, you must create an instrumentation class. An instrumentation class is a uniquely-named container for instrumentation data. When you create an instrumentation class, you define its name and its type. You may also provide type-specific information, such as the range and resolution of a histogram statistic.

You use the instrumentation class reference that is returned from the creation call to specify the destination of the instrumentation data that you wish to record. An instrumentation class is either enabled or disabled; when it is disabled, none of its information is collected or written to permanent storage.

The Instrumentation Tree

Instrumentation classes are organized in a tree structure. Every instrumentation class has a parent node in the Instrumentation Tree; the root node of the tree is specified by the constant `kInstRootClassRef`. Trace and statistics classes must be leaf nodes; only a special type of instrumentation node, the Path Instrumentation Class, can have children.

Because every instrumentation class is a member of the instrumentation tree, they are sometimes referred to as instrumentation nodes.

Organizing your instrumentation classes into a tree allows you to avoid name conflicts among different classes. By creating subtrees for each of your software components, any component may use generic class names without interfering with the others.

Trace Event Records

Every time you log a trace event, the instrumentation system records a trace event record. This record includes the trace instrumentation class reference and a timestamp indicating when the trace was logged.

A pair of trace events belonging to the same class may be marked as a “start event” and an “end event.” This allows analysis software to recognize that they represent a single “event range,” and to display them appropriately. These traces are normally logged at the beginning and end of a single routine or interesting operation.

A trace event between a start event and an end event may be marked as a “middle event.” Such an event record will be displayed by the analysis software as part of the event range; it is useful for segmenting a procedure into several identifiable parts.

An event that includes a start, an end, and zero or more middle event records is called a multi-part event.

Trace event records may also include user-defined data. This data is formatted into strings using `InstDataDescriptorRef`’s, which are similar to `printf()` format descriptor strings.

In the current implementation, trace event records are stored in a circular buffer in memory and periodically transferred from the buffer to an instrumentation data file. It is possible to fill the circular buffer by logging trace events faster than the system can transfer them out of the buffer. If this happens, older trace records are lost as they are overwritten by newer ones.

The current implementation includes a mechanism to increase the buffer size in order to avoid this problem. See the *Instrumentation System User’s Guide* for details.

It is also possible to create a special type of trace instrumentation class, called a trace summary class. Individual trace event records are not produced when a trace is logged to such a class; instead the class just keeps track of how many traces were logged and how much time was spent between its start and end traces.

Types of Statistics

A statistics instrumentation class is a container that holds a particular set of values; they are updated over time by your software. The instrumentation system defines a variety of different statistics; each one is best-suited to keeping track of a particular type of information.

A Magnitude is simply a signed 32-bit number. As a magnitude class is updated with new values, the instrumentation system keeps track of the maximum magnitude supplied, the minimum, the current value, the cumulative total, and the number of times it was updated.

Magnitudes are often used to record simple varying values, such as the amount of free memory at a given time.

A Growth value is an unsigned 64-bit counter that is updated with 32-bit incremental values. Like a magnitude class, growth classes keep track of the current update increment, the maximum increment encountered, the minimum increment, the current total, and the number of times the class was updated.

However, since growth values can only increase, they are generally used for counting events such as cache hits and misses. Analysis software may display the information inside growth statistics as a rate, such as the number of cache hits per second.

A Histogram is specified by a range and a bucket width. The Histogram class maintains a set of buckets of the specified width distributed evenly over the range. Each bucket holds the number of values supplied to the histogram that fell inside that bucket’s range. The class also maintains an overflow count, which holds the number of values that fell outside the entire range.

Histograms are good for profiling things like parameter values, which tend to vary unpredictably over a certain range. For instance, you could create a histogram of the block size parameter of a memory allocation routine to profile allocation sizes.

The Split Histogram is a variant of the histogram; in it, the range is divided into two parts with different bucket widths for each part. A Split Histogram can be used in place of a regular histogram to cover one part of a range in greater – or lesser – detail.

The Tally statistic is similar to a histogram, in that it keeps track of a set of counts. It is created with a fixed number of buckets, which are initially undefined. You update a tally with a bucket identifier and an increment. If there is a bucket associated with that identifier, its count is incremented accordingly. If not, and there are still buckets left undefined, one is defined for the identifier and its initial count is set to the increment. If all buckets have been defined, the overflow count is incremented instead.

Tallies are often used to create profiles of operations whose profiled parameter does not vary over a fixed range, or whose range is too large to make a histogram practical. For instance, you could use a tally to count how many files of various types your application was asked to open, using the file type as the bucket identifier.

Using the Instrumentation System

This section describes how to create, enable, and disable instrumentation classes, how to log trace information, and how to record statistics information.

Creating Instrumentation Classes

The instrumentation system provides routines that allow you to create each type of instrumentation class. For example, calling `InstCreateHistogramClass` will create a Histogram statistic class.

Each creation routine returns a reference to the new instrumentation class. This reference is used for subsequent operations involving that class.

The position of the class within the instrumentation tree is determined by the parent node reference and the class name string that is supplied to the creation routine. The parent node reference must be either `kInstRootClassRef`, or an instrumentation class created by a previous call to `InstCreatePathClass`. The class name string may be a simple class name, or a colon-delimited “partial pathname” relative to the parent node reference.

For example, the code in in figure 1-1 will create a growth instrumentation class called “Page Faults” as a child of a parent node called “Totals,” which itself is a child of the root of the tree.

Listing 1-1 Sample code: creating a class

```
OSStatus    CreatePageFaultsStat( InstGrowthClassRef *pNewClassRef)
{
      err = InstCreateGrowthClass( kInstRootClassRef,
                  "Totals:Page Faults", kInstEnableClassMask, pNewClassRef);
      return err;
}
```

The class is created enabled, so it will be eligible to accept and record data immediately.

Specifying a partial path in the class name parameter of a creation routine will create intervening path nodes if they do not already exist. For example, calling `InstCreateMagnitudeClass` with the class name "Graphics:Testing:LineBlits" when the instrumentation tree is empty will create three nodes: a magnitude node called "LineBlits," and two path nodes called "Graphics" and "Testing."

Instrumentation class references are global to the system. If you attempt to create the same instrumentation node twice, the original class reference is returned by the second creation call, with no error. If you attempt to create a node of a different type with the same name as an existing node, at the same position in the instrumentation tree, then an error will be returned.

Once `InstDisposeClass` has been called on an instrumentation class, it is unusable by any of its clients.

Logging Trace Information

In order to log trace information, you must create a trace instrumentation class to hold the event records. The `InstCreateTraceClass` function will return a trace instrumentation class reference.

Normally, traces are logged using the `InstLogTraceEvent` function.

```
      InstLogTraceEvent( traceClassRef, kInstNoEventTag, kNilOptions);
```

This creates a simple trace event record belonging to the trace class represented by `traceClassRef`, stamped with the current time.

You can create a multi-part event by calling `InstLogTraceEvent` more than once, specifying `kInstStartEvent`, `kInstMiddleEvent`, or `kInstEndEvent` in the `InstEventOptions` field. Each multi-part event must have one start event record and one end event record.

The start, middle, and end events of a multi-part event must have the same `InstEventTag`; this allows the instrumentation system to recognize them as being part of the same event. The event tag that you choose should be locally unique among other traces of that class; for example, a function which is called recursively that logs a start trace on entry and an end trace at exit should ensure that it does not create several start events with the same event tag.

Similar precautions should be taken when the creation of a start/end pair may be interrupted by the same trace being logged at a higher interrupt level.

You can use the `InstCreateEventTag` function to generate event tags that are guaranteed to be unique across the system.

If you are not creating multi-part events, you may use the `InstEventTag` parameter to record an arbitrary 32-bit value in the trace event record.

In order to record user-defined data along with the standard trace event information, you must create a DataDescriptor that defines how the data can be formatted into an ASCII string. You create a DataDescriptor by calling `InstCreateDataDescriptor`; you can create more than one by calling `InstCreateDataDescriptors`. The format strings you specify follow the rules for the standard `printf()` format descriptor strings.

The `InstLogTraceEventWithData` function takes a reference to a DataDescriptor and an arbitrary number of parameters containing the data. It creates a trace event record containing a string representation of the the data.

Listing 1-2 Sample code: logging an event with data

```
InstCreateDataDescriptor( "%d:%d:%d", &aDataDesc);
InstLogTraceEventWithData( traceClassRef, kInstNoEventTag, kNilOptions,
                              aDataDesc, hours, minutes, seconds);
```

The `InstLogTraceEventWithDataStructure` function does the same job as the `InstLogTraceEventWithData` function; it is provided for development environments that do not support functions with a variable number of arguments. Instead, it accepts a pointer to a structure that contains the argument list.

Listing 1-3 Sample code: calling InstLogTraceEventWithDataStructure

```
struct      MyTime
{
      UInt32            hours, minutes, seconds;
};
MyTime            t = { 12, 0, 0 };
InstLogTraceEventWithDataStructure( traceClassRef, kInstNoEventTag,
                              kNilOptions, aDataDesc, (Byte*) &t, sizeof t);
```

Members of the structure must be four-byte aligned.

When you no longer need the DataDescriptor, you may release its storage by calling `InstDisposeDataDescriptor`.

Logging to a Summary Trace Class

Under some circumstances, you may not wish to have an individual trace event record produced every time you log a trace event. Usually this is because the event occurs very frequently and generates a large amount of trace data.

As an alternative, you can set the `kInstSummaryTraceClassMask` option bit when you call the `InstCreateTraceClass` function to create the trace instrumentation class. This will produce a summary trace class. Trace events are logged to a summary trace class using the same routines as with regular trace classes, but the results appear as a growth statistic.

The statistics data has the following interpretation: the number of times the class was updated corresponds to the number of trace events that were logged. The total growth value corresponds to the number of microseconds spent between each start and end event logged. The the minimum and maximum increments correspond to the minimum and maximum start/end microsecond durations.

Recording Statistics Information

In order to record statistics information, you must create a statistics instrumentation class of the appropriate type to hold the data. The `InstCreateGrowthClass`, `InstCreateMagnitudeClass`, `InstCreateHistogramClass`, `InstCreateSplitHistogramClass`, and `InstCreateTallyClass` functions all return an instrumentation class reference.

When a class is created its values are zero; tally classes are empty. You put data into a class by calling the appropriate update function: `InstUpdateGrowth` for growth statistics, `InstUpdateMagnitudeAbsolute` or `InstUpdateMagnitudeDelta` for magnitudes, `InstUpdateHistogram` for histograms and split histograms, and `InstUpdateTally` for tallies.

When a class is updated, it records the fact in its update count and modifies its data based on the update.

Instrumentation classes are global to the system, so all updates to the same statistics class reference will end up in the same container.

In the current implementation, individual statistic updates are not recorded; instead, “snapshots” are taken of the current values of each enabled statistic at various times. See the *Instrumentation System User’s Guide* for more details.

Enabling and Disabling Instrumentation Classes

An instrumentation class is either enabled or disabled. When a trace class is enabled, logging a trace event to it will place a trace event record into the circular buffer, which will get copied to permanent storage. When it is disabled, logging a trace has no effect.

When a statistics class is enabled, calling its update routine will increment the update count and modify the current value based on the update. When the instrumentation system samples statistics, the current value will be written to permanent storage. When a statistics class is disabled, calling its update routine has no effect; disabled statistics are not sampled by the instrumentation system.

You enable and disable instrumentation classes by calling the `InstEnableClass` and `InstDisableClass` functions. Disabling a path instrumentation class – including kInstRootClassRef – will disable the entire subtree; enabling it will re-enable those members of the subtree that were previously enabled.

Instrumentation Reference

Constants

This section describes the constants defined by the instrumentation interface. You use these constants to specify the root of the instrumentation tree, instrumentation class options, and trace event options.

When you are creating an instrumentation class, you must specify its parent node. Use the constant `kInstRootClassRef` to specify the root of the instrumentation tree.

```
#define kInstRootClassRef       ( (InstClassRef) -1)
```

You may specify that an instrumentation classes is initially enabled or disabled. Pass `kInstEnableClassMask` in the InstClassOptions parameter to enable it, or `kInstDisableClassMask` to disable it.

When you create a trace instrumentation class, you may add `kInstSummaryTraceClassMask` to the `InstEventOptions` parameter to create a trace summary class.

```
enum {      kInstDisableClassMask             = 0x00,
            kInstEnableClassMask              = 0x01
            kInstSummaryTraceClassMask        = 0x20
};
```

When you log a trace event, you may specify that the event marks the beginning of a multi-part event, a point in the middle, or the end. Pass `kInstStartEvent`, `kInstMiddleEvent`, or `kInstEndEvent` in the `InstEventOptions` parameter of the routine you use to log trace events .

```
enum {      kInstStartEvent                   = 1,
            kInstEndEvent                     = 2,
            kInstMiddleEvent                  = 3
};
```

If you log a multi-part event, you must specify the same `InstEventTag` in the start, middle and end events. This allows the analysis software to recognize them as a set.

If you are not logging a multi-part event, you may specify `kInstNoEventTag` to indicate that you are not using the `InstEventTag`.

```
enum {      kInstNoEventTag = 0     };
```

Instrumentation Routines

This section describes the routines provided by the instrumentation system. You can use these routines to initialize and terminate the 68K instrumentation library, create and destroy instrumentation classes and data descriptors, provide data to instrumentation classes, and enable or disable instrumentation classes.

Except where noted, each of these routines may be called from any interrupt level.

Initialization and Termination

You can use these routines to initialize and terminate the 68K instrumentation library. PowerPC clients do not have to call initialization or termination routines; the operating system initializes and terminates shared libraries when necessary.

InstInitialize68K

You call `InstInitialize68K` once from 68K code to initialize the statically-linked 68K instrumentation library.

```
pascal OSStatus         InstInitialize68K( void);
```

DESCRIPTION

The InstInitialize68K function opens a connection to the PowerPC instrumentation implementation and sets up the Mixed Mode tables necessary to call it from a 68K environment. It does not require external support from an A5- or A4-world.

SPECIAL CONSIDERATIONS

The `InstInitialize68K` function should not be called from interrupt time.

InstTerminate68K

You can call `InstTerminate68K` to release storage allocated by calling `InstInitialize68K`.

```
pascal OSStatus         InstTerminate68K( void);
```

DESCRIPTION

The `InstTerminate68K` function closes the connection to the PowerPC instrumentation implementation and deallocates any global storage.

SPECIAL CONSIDERATIONS

The `InstTerminate68K` function should not be called from interrupt time. You do not have to call it at all if your process calls ExitToShell.

Creating and Destroying Instrumentation Classes

You can use these routines to create and destroy instrumentation classes.

InstCreatePathClass

You can call `InstCreatePathClass` to place a new path instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreatePathClass( InstPathClassRef parentClass,
                  const char *className, InstClassOptions options,
                  InstPathClassRef *returnPathClass);
```

`parentClass`	The parent class node of the class to be created
`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`returnPathClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The InstCreatePathClass function creates a new path instrumentation class node in the instrumentation tree. Path classes are analogous to folders in a file system; you can use the class reference that is returned as the `parentClass` in subsequent creation calls.

InstCreateTraceClass

You can call `InstCreateTraceClass` to place a new trace instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreateTraceClass( InstPathClassRef parentClass,
      const char *className, OSType component, InstClassOptions options,
      InstTraceClassRef *returnTraceClass);
```

`parentClass`	The parent class node of the class to be created
`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`component`	A four-character code used to identify the software component that produces the trace events.
`options`	Either `kInstEnableClassMask` or `kInstDisableClassMask`.
`returnTraceClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The `InstCreateTraceClass` function creates a container for the trace event records that are produced by calling `InstLogTraceEvent`, `InstLogTraceEventWithData`, or `InstLogTraceEventWithDataStructure`. A trace event class is usually associated with a particular client routine or operation.

Pass `kInstSummaryTraceClassMask` in the `options` parameter to create a summary trace class.

The component code is put into all trace event records, but it is not currently used by the instrumentation system.

InstCreateHistogramClass

You can call `InstCreateHistogramClass` to place a new histogram instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreateHistogramClass( InstPathClassRef parentClass,
      const char *className, SInt32 lowerBounds, SInt32 upperBounds,
      UInt32 bucketWidth, InstClassOptions options,
      InstHistogramClassRef *returnHistogramClass);
```

`parentClass`	The parent class node of the class to be created
`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`lowerBounds`	The lower limit of the histogram range.
`upperBounds`	The upper limit of the histogram range.
`bucketWidth`	The portion of the range that each bucket will cover.
`options`	Either `kInstEnableClassMask` or `kInstDisableClassMask`.
`returnHistogramClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The `InstCreateHistogramClass` function creates a container for the histogram data that is produced by calling `InstUpdateHistogram`. The number of buckets the histogram will maintain is determined implicitly by dividing the histogram range by the bucket width.

SPECIAL CONSIDERATIONS

In the current implementation, each histogram data point consists of one 32-bit value for each bucket in the histogram; a data point is recorded every sample period. Thus, creating histograms with many buckets will result in large instrumentation data files.

InstCreateSplitHistogramClass

You can call `InstCreateSplitHistogramClass` to place a new split histogram instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreateSplitHistogramClass(
      InstPathClassRef parentClass, const char *className,
      SInt32 histogram1LowerBounds, UInt32 histogram1BucketWidth,
      SInt32 knee, SInt32 histogram2UpperBounds,
      UInt32 histogram2BucketWidth, InstClassOptions options,
      InstSplitHistogramClassRef *returnSplitHistogramClass);
```

`parentClass`	The parent class node of the class to be created

`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`histogram1LowerBounds`	The lower limit of the histogram range.
`histogram1BucketWidth`	The portion of the first part of the histogram range that each bucket of the first set will cover.
`knee`	Where the first part of the histogram range ends and the second begins.
`histogram2BucketWidth`	The portion of the second part of the histogram range that each bucket of the second set will cover.
`histogram2UpperBounds`	The upper limit of the histogram range.
`options`	Either `kInstEnableClassMask` or `kInstDisableClassMask`.
`returnSplitHistogramClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The `InstCreateSplitHistogramClass` function creates a container for the histogram data that is produced by calling `InstUpdateHistogram`. The number of buckets the split histogram will maintain is determined implicitly by dividing the first and second parts of the histogram range by the first and second bucket widths, respectively.

SPECIAL CONSIDERATIONS

In the current implementation, each histogram data point consists of one 32-bit value for each bucket in the histogram; a data point is recorded every sample period. Thus, creating histograms with many buckets will result in large instrumentation data files.

InstCreateMagnitudeClass

You can call `InstCreateMagnitudeClass` to place a new magnitude instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreateMagnitudeClass( InstPathClassRef parentClass,
                    const char *className, InstClassOptions options,
                    InstMagnitudeClassRef *returnMagnitudeClass);
```

`parentClass`	The parent class node of the class to be created

`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`options`	Either `kInstEnableClassMask` or `kInstDisableClassMask`.
`returnMagnitudeClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The `InstCreateMagnitudeClass` function creates a container for the magnitude data that is produced by calling `InstUpdateMagnitudeAbsolute` or `InstUpdateMagnitudeDelta`.

InstCreateGrowthClass

You can call `InstCreateGrowthClass` to place a new growth instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreateGrowthClass( InstPathClassRef parentClass,
                    const char *className, InstClassOptions options,
                    InstGrowthClassRef *returnGrowthClass);
```

`parentClass`	The parent class node of the class to be created
`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`options`	Either `kInstEnableClassMask` or `kInstDisableClassMask`.
`returnGrowthClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The `InstCreateGrowthClass` function creates a container for the growth data that is produced by calling `InstUpdateGrowth`.

InstCreateTallyClass

You can call `InstCreateTallyClass` to place a new tally instrumentation class node into the instrumentation tree.

```
pascal OSStatus InstCreateTallyClass( InstPathClassRef parentClass,
      const char *className, UInt16 maxNumberOfValues,
      InstClassOptions options, InstTallyClassRef *returnTallyClass);
```

`parentClass`	The parent class node of the class to be created
`className`	The zero-terminated name of the class to be created. It may be a colon-delimited partial path, relative to `parentClass`.
`maxNumberOfValues`	The maximum number of buckets to create.
`options`	Either `kInstEnableClassMask` or `kInstDisableClassMask`.
`returnTallyClass`	On exit, the instrumentation class reference of the new class.

DESCRIPTION

The `InstCreateTallyClass` function creates a container for the tally data that is produced by calling `InstUpdateTally`. Initially, no buckets are created; buckets are allocated as `InstUpdateTally` is called with new bucket identifiers.

SPECIAL CONSIDERATIONS

In the current implementation, each tally data point consists of two 32-bit values for each bucket allocated by the tally; a data point is recorded every sample period. Thus, creating tallies with many buckets will result in large instrumentation data files.

InstDisposeClass

You can call `InstDisposeClass` to prevent any more data from being added to an instrumentation class.

```
pascal void InstDisposeClass( InstClassRef theClass);
```

`theClass`	The instrumentation class you wish to dispose of.

DESCRIPTION

The `InstDisposeClass` function marks an instrumentation class as unusable; it can no longer be enabled or disabled, and further operations on it will have no effect.

Calling `InstDisposeClass` on a path instrumentation class implicitly disposes of its children.

SPECIAL CONSIDERATIONS

Calling this function does not free any significant system resources. It is normally not necessary to dispose of an instrumentation class once it is created.

Creating and Destroying Data Descriptors

This section describes how to create and destroy data descriptors, which are used to specify how user-defined data is to be formatted in trace event records.

InstCreateDataDescriptor

You can call `InstCreateDataDescriptor` to create a DataDescriptor from a format string.

```
pascal OSStatus InstCreateDataDescriptor( const char *formatString,
                                  InstDataDescriptorRef *returnDescriptor);
```

`formatString`
A zero-terminated printf() format string.

`returnDescriptor`
On exit, a reference to the new DataDescriptor.

DESCRIPTION

The `InstCreateDataDescriptor` function creates a DataDescriptor that you can use to specify how a list of parameters is to be formatted by the `InstLogTraceEventWithData` or `InstLogTraceEventWithDataStructure` functions. The format string should follow the same rules as a `printf()` format string, with one descriptor for every parameter that you wish to record.

SPECIAL CONSIDERATIONS

This function should not be called at interrupt time.

InstCreateDataDescriptors

You can call `InstCreateDataDescriptors` to create a set of DataDescriptor’s with a single call.

```
pascal OSStatus InstCreateDataDescriptors( const char **formatStrings,
                        UInt32 numberOfDescriptors,
                        InstDataDescriptorRef *returnDescriptorList);
```

`formatStrings`
: An array of zero-terminated printf() format strings.

`numberOfDescriptors`
: The number of strings in the formatStrings array.

`returnDescriptorList`
: On exit, this points to an array of `InstDataDescriptorRef`’s.

DESCRIPTION

The `InstCreateDataDescriptors` function is a shortcut for creating a set of `InstDataDescriptorRef`’s from an array of format strings. Each element of the `returnDescriptorList` array corresponds to the same element of the `formatStrings` array.

See the description of the `InstCreateDataDescriptor` function, above, for more details.

SPECIAL CONSIDERATIONS

This function should not be called at interrupt time.

InstDisposeDataDescriptor

You can call `InstDisposeDataDescriptor` to release the storage used by a DataDescriptor.

```
pascal void InstDisposeDataDescriptor(
                              InstDataDescriptorRef theDescriptor);
```

`theDescriptor`
: The reference to a DataDescriptor that you no longer need.

DESCRIPTION

The `InstDisposeDataDescriptor` function releases the storage used by a DataDescriptor. You should not use the `InstDataDescriptorRef` after calling this function.

SPECIAL CONSIDERATIONS

This function should not be called at interrupt time.

InstDisposeDataDescriptors

You can call `InstDisposeDataDescriptors` to release the storage used by an array of DataDescriptor's.

```
pascal void InstDisposeDataDescriptors(
                              InstDataDescriptorRef *theDescriptorList,
                              UInt32 numberOfDescriptors);
```

`theDescriptorList`
A pointer to an array of `InstDataDescriptorRef`'s.

`numberOfDescriptors`
The number of DataDescriptor's in the array.

DESCRIPTION

The `InstDisposeDataDescriptors` function releases the storage used by the DataDescriptor's in `theDescriptorList`. You should not use any of these `InstDataDescriptorRef`'s after calling this function.

SPECIAL CONSIDERATIONS

This function should not be called at interrupt time.

Logging Trace Events

This section describes the routines you use to log trace events.

InstLogTraceEvent

You can call `InstLogTraceEvent` to add a trace event record to a trace instrumentation class.

```
pascal void InstLogTraceEvent( InstTraceClassRef theTraceClass,
                  InstEventTag eventTag, InstEventOptions options);
```

`theTraceClass`	The trace instrumentation class that you wish to log an event against.
`eventTag`	A tag that is used to link multi-part trace events.
`options`	A value used to specify that that this is a multi-part event.

DESCRIPTION

The `InstLogTraceEvent` function will time-stamp an event record for the specified trace class, which will be recorded by the instrumentation system. The options parameter can be used to indicate that the trace record is part of a multi-part event (`kInstStartEvent`, `kInstMiddleEvent`, or `kInstEndEvent`). The eventTag parameter is used to link the members of a multi-part event.

If you are not creating multi-part events, you can use the `eventTag` parameter to store an arbitrary 32-bit value in the event record.

InstLogTraceEventWithData

You can call `InstLogTraceEventWithData` to add a trace event record that includes user-defined data to a trace instrumentation class.

```
pascal void InstLogTraceEventWithData( InstTraceClassRef theTraceClass,
                        InstEventTag eventTag, InstEventOptions options,
                        InstDataDescriptorRef theDescriptor, ...);
```

`theTraceClass`	The trace instrumentation class that you wish to log an event against.
`eventTag`	A tag that is used to link multi-part trace events.
`options`	A value used to specify that that this is a multi-part event.
`theDescriptor`	A data descriptor that will be used to format the data into a string.
`(...)`	An arbitrary number of parameters describing the data.

DESCRIPTION

The `InstLogTraceEventWithData` function works similarly to the `InstLogTraceEvent` function, described above, but includes user-defined data in the trace event record.

SPECIAL CONSIDERATIONS

In the current implementation, `InstLogTraceEventWithData` is significantly more expensive to call than `InstLogTraceEvent`.

This function cannot be called from 68K code.

This function should not be called at interrupt time.

InstLogTraceEventWithDataStructure

You can call `InstLogTraceEventWithData` to add a trace event record that includes user-defined data to a trace instrumentation class.

```
pascal void InstLogTraceEventWithDataStructure(
      InstTraceClassRef theTraceClass, InstEventTag eventTag,
      InstEventOptions options, InstDataDescriptorRef descriptorRef,
      const UInt8 *dataStructure, ByteCount structureSize);
```

`theTraceClass`	The trace instrumentation class that you wish to log an event against.
`eventTag`	A tag that is used to link multi-part trace events.
`options`	A value used to specify that that this is a multi-part event.
`theDescriptor`	A data descriptor that will be used to format the data into a string.
`dataStructure`	A pointer to a structure containing the argument list.
`structureSize`	The size of the structure pointed to by `dataStructure`.

DESCRIPTION

The `InstLogTraceEventWithDataStructure` function works similarly to the `InstLogTraceEvent` function, described above, but includes user-defined data in the trace event record.

Each member of the argument list structure should be 32-bit aligned.

SPECIAL CONSIDERATIONS

In the current implementation, `InstLogTraceEventWithDataStructure` is significantly more expensive to call than `InstLogTraceEvent`.

This function should not be called at interrupt time.

InstCreateEventTag

You can call `InstCreateEventTag` to generate a unique `InstEventTag`.

```
pascal InstEventTag InstCreateEventTag( void);
```

DESCRIPTION

The `InstCreateEventTag` function will return a different `InstEventTag` every time you call it. Clients can call it from different processes or execution levels without getting duplicate values.

Updating Statistics Classes

This section describes the routines that you use to update statistics classes.

InstUpdateGrowth

You can call `InstUpdateGrowth` to update the value of a growth statistics class.

```
pascal void InstUpdateGrowth( InstGrowthClassRef theGrowthClass,
                              UInt32 increment);
```

`theGrowthClass`	The growth statistics class that you wish to update.
`increment`	The value that you wish to add to the current total.

DESCRIPTION

The `InstUpdateGrowth` function adds the `increment` to the current total maintained by the growth class. It also sets the minimum increment and / or the maximum increment values as necessary, and adds one to the update count.

InstUpdateMagnitudeAbsolute

You can call `InstUpdateMagnitudeAbsolute` to set the current value of a magnitude statistics class.

```
pascal void InstUpdateMagnitudeAbsolute(
            InstMagnitudeClassRef theMagnitudeClass, SInt32 newValue);
```

`theMagnitudeClass`
The magnitude statistics class that you wish to update.

`newValue` The new magnitude value for the class.

DESCRIPTION

The `InstUpdateMagnitudeAbsolute` function sets the current value of the magnitude class to the value supplied, and adds this value to the running total of all values set so far. It also sets the minimum and / or the maximum values as necessary, and adds one to the update count.

InstUpdateMagnitudeDelta

You can call `InstUpdateMagnitudeDelta` to set the current value of a magnitude statistics class relative to the previous value.

```
pascal void InstUpdateMagnitudeDelta(
      InstMagnitudeClassRef theMagnitudeClass, SInt32 delta);
```

`theMagnitudeClass`
The magnitude statistics class that you wish to update.

`delta` The amount to add to the current value to get the new value.

DESCRIPTION

The `InstUpdateMagnitudeDelta` function is equivalent to reading the current value of `theMagnitudeClass`, adding `delta` to it, and calling the `InstUpdateMagnitudeAbsolute` function with the result.

InstUpdateHistogram

You can call `InstUpdateHistogram` to add a new value to a histogram or split histogram class.

```
pascal void InstUpdateHistogram(
                InstHistogramClassRef theHistogramClass,
                SInt32 value, UInt32 count);
```

`theHistogramClass`	The histogram class you wish to update.
`value`	A value that falls within the range of the histogram.
`count`	The weight of the value in the histogram's bucket.

DESCRIPTION

The `InstUpdateHistogram` function identifies which bucket in the histogram the value falls into and adds `count` to the bucket's "hit count." If the value falls outside the range specified in the histogram, `count` is added to the overflow count instead. The histogram's update count is incremented.

The `InstUpdateHistogram` function is used to update both histograms and split histograms.

Normally, the value of `count` is 1.

InstUpdateTally

You can call `InstUpdateTally` to add a new value to a tally class.

```
pascal void InstUpdateTally( InstTallyClassRef theTallyClass,
                             void *bucketID, UInt32 count);
```

`theTallyClass`	The tally class you wish to update.
`bucketID`	The bucket identifier you wish to add to.
`count`	The value you wish to add to the bucket count.

DESCRIPTION

The `InstUpdateTally` function first tries to find a bucket within its list that has been registered to `bucketID`. If it finds one, it adds `count` to the bucket's value. If not, and the number of buckets is below its maximum, it creates a new one one for `bucketID` and sets its value to `count`. If all the buckets have been created, it adds `count` to its overflow count instead. The tally's update count is incremented.

Normally, the value of `count` is 1.

Enabling and Disabling Instrumentation Classes

This section describes the routines that allow you to enable and disable instrumentation classes.

InstEnableClass

You can call `InstEnableClass` to enable an instrumentation class.

```
pascal OSStatus InstEnableClass( InstClassRef classRef);
```

`classRef`	The class you wish to enable.

DESCRIPTION

The `InstEnableClass` will enable the specified class. Enabling an enabled class has no effect.

InstDisableClass

You can call `InstDisableClass` to disable an instrumentation class.

```
pascal OSStatus InstDisableClass( InstClassRef classRef);
```

`classRef`	The class you wish to disable.

DESCRIPTION

The `InstDisableClass` will disable the specified class. Disabling a disabled class has no effect.

Summary of the Instrumentation System

Constants

```
#define kInstRootClassRef      ( (InstClassRef) -1)

enum {      kInstDisableClassMask              = 0x00,
            kInstEnableClassMask               = 0x01,
            kInstSummaryTraceClassMask         = 0x20
};

enum {      kInstStartEvent                    = 1,
            kInstEndEvent                      = 2,
            kInstMiddleEvent                   = 3
};

enum {      kInstNoEventTag                    = 0   };
```

Instrumentation Routines

Initialization and Termination

```
pascal OSStatus         InstInitialize68K( void);
pascal OSStatus         InstTerminate68K( void);
```

Creating and Destroying Instrumentation Classes

```
pascal OSStatus InstCreatePathClass( InstPathClassRef parentClass,
                  const char *className, InstClassOptions options,
                  InstPathClassRef *returnPathClass);

pascal OSStatus InstCreateTraceClass( InstPathClassRef parentClass,
      const char *className, OSType component, InstClassOptions options,
      InstTraceClassRef *returnTraceClass);

pascal OSStatus InstCreateHistogramClass( InstPathClassRef parentClass,
      const char *className, SInt32 lowerBounds, SInt32 upperBounds,
      UInt32 bucketWidth, InstClassOptions options,
      InstHistogramClassRef *returnHistogramClass);

pascal OSStatus InstCreateSplitHistogramClass(
      InstPathClassRef parentClass, const char *className,
      SInt32 histogram1LowerBounds, UInt32 histogram1BucketWidth,
      SInt32 knee, SInt32 histogram2UpperBounds,
      UInt32 histogram2BucketWidth, InstClassOptions options,
      InstSplitHistogramClassRef *returnSplitHistogramClass);

pascal OSStatus InstCreateMagnitudeClass( InstPathClassRef parentClass,
                        const char *className, InstClassOptions options,
                        InstMagnitudeClassRef *returnMagnitudeClass);

pascal OSStatus InstCreateGrowthClass( InstPathClassRef parentClass,
                        const char *className, InstClassOptions options,
                        InstGrowthClassRef *returnGrowthClass);

pascal OSStatus InstCreateTallyClass( InstPathClassRef parentClass,
      const char *className, UInt16 maxNumberOfValues,
      InstClassOptions options, InstTallyClassRef *returnTallyClass);

pascal void InstDisposeClass( InstClassRef theClass);
```

Creating and Destroying Data Descriptors

```
pascal OSStatus InstCreateDataDescriptor( const char *formatString,
                              InstDataDescriptorRef *returnDescriptor);

pascal OSStatus InstCreateDataDescriptors( const char **formatStrings,
                        UInt32 numberOfDescriptors,
                        InstDataDescriptorRef *returnDescriptorList);

pascal void InstDisposeDataDescriptor(
                              InstDataDescriptorRef theDescriptor);

pascal void InstDisposeDataDescriptors(
                              InstDataDescriptorRef *theDescriptorList,
                              UInt32 numberOfDescriptors);
```

Logging Trace Events

```
pascal void InstLogTraceEvent( InstTraceClassRef theTraceClass,
                  InstEventTag eventTag, InstEventOptions options);

pascal void InstLogTraceEventWithData( InstTraceClassRef theTraceClass,
                        InstEventTag eventTag, InstEventOptions options,
                        InstDataDescriptorRef theDescriptor, ...);


pascal void InstLogTraceEventWithDataStructure(
      InstTraceClassRef theTraceClass, InstEventTag eventTag,
      InstEventOptions options, InstDataDescriptorRef descriptorRef,
      const UInt8 *dataStructure, ByteCount structureSize);

pascal InstEventTag InstCreateEventTag( void);
```

Updating Statistics Classes

```
pascal void InstUpdateGrowth( InstGrowthClassRef theGrowthClass,
                              UInt32 increment);


pascal void InstUpdateMagnitudeAbsolute(
            InstMagnitudeClassRef theMagnitudeClass, SInt32 newValue);


pascal void InstUpdateMagnitudeDelta(
      InstMagnitudeClassRef theMagnitudeClass, SInt32 delta);

pascal void InstUpdateHistogram(
                  InstHistogramClassRef theHistogramClass,
                  SInt32 value, UInt32 count);

pascal void InstUpdateTally( InstTallyClassRef theTallyClass,
                              void *bucketID, UInt32 count);
```

Enabling and Disabling Instrumentation Classes

```
pascal OSStatus InstEnableClass( InstClassRef classRef);
pascal OSStatus InstDisableClass( InstClassRef classRef);
```