Adiak User’s Guide

Matthew LeGendre <legendre1@llnl.gov>

June 3, 2019

# Introduction

Adiak is an API for recording meta-data about HPC simulations. An HPC application code may, for example, record what user invoked it, the version of the code being run, a computed time history showing density changes, or how long the application spent performing file IO. Adiak represents this metadata as Name/Value pairs. Names are arbitrary strings, with some standardization, and the values are represented by a flexible dynamic type system.

Tools can subscribe to Adaik’s name/value pairs. Example tools could be be performance analysis tools recording metadata, a workflow manager, or a serializer that writes metadata to a standard format.

Adiak has a C and C++ API. The implementation is C-only, and the C++ API is header-only wrapper around the C interface. When using the C++ API values can be passed as template parameters, allowing for terse and convenient flexibility in value types. When using the C API values types are described by printf-inspired strings.

## Examples

### Basic Example

To register a Name/Value pair with Adiak’s C++ API:

#include “adiak.hpp”

int main(int argc, char \*argv[]) {  
adiak::init(NULL);  
adiak::value(“dimensions” 3);  
adiak::value(“runid”, “exe34\_51”);  
adiak::value(“calced\_sum”, 5.15, adiak\_physics);  
adiak::fini();  
}

To register the same Name/Value with the C API:  
#include “adiak.h”

int main(int argc, char \*argv[]) {  
adiak\_init(NULL);  
adiak\_namevalue(“dimensions”, adiak\_general, “%d”, 3);  
adiak\_namevalue(“runid”, adiak\_general, “%s”, “exec34\_51”);  
adiak\_namevalue(“calced\_sum”, adiak\_physics, “%f”, 5.15);  
adiak\_fini();  
}

There are several points to note about the above examples:

* Adiak has different include files, adiak.h and adiak.hpp, for the C and C++ interfaces respectively. You only need to use one. It is safe to mix calls to the C and C++ APIs, though these examples do not illustrate that.
* Before making any other Adiak API calls you must call adiak’s init function. The NULL parameter in this example disables MPI support in Adiak. A pointer to an MPI communicator can be passed in this parameter to enable MPI support
* Both of the adiak::value and adiak\_namevalue are performing the same task: associating a group of names: dimensions, runid, and calced\_sum, with respective values of: int, string, and double. In the C++ example we can simply pass both the name and value to the adiak::value function. In the C example we have to describe the type of the value, using a printf-style string.
* The C example also sets an adiak category when it passes adiak\_general. This can be used to group name/value pairs into one of several categories, which some tools may find useful. The C++ interface can also set a category, but it is passed as a default parameter that’s set to adiak\_general already.
* We finish by calling the appropriate fini function to adiak.

## Implicit Routines

Adiak contains several convenience routines that register commonly used Name/Value pairs under standardized names. For example:

#include “adiak.hpp”

int main(int argc, char \*argv[]) {  
 adiak::init(NULL);  
 adiak::uid();  
 adiak::date()  
 adiak::fini();  
}

The adiak::uid() function creates a name of “uid” and associates it with a string of the user id that owns this process. For example, this call might create a Name/Value pair of “uid”/”jsmith1”. The adiak::date function associates a “date” name with the current date and time. There are a larger set of standardized names in Section 3 of this document. Equivalent functions are also available in the C interface.

## Tool Interface

Tools can plug into Adiak to receive name/value pairs provided by the application. The name/value pairs are provided by callbacks, which can be delivered in batches at certain points or as the application provides them. For example:

#include “adiak\_tool.h”

void cb(const char \*name, adiak\_category\_t category, const char \*subcategory, adiak\_value\_t \*value, adiak\_datatype\_t \*t, void \*opaque\_value)

{

printf(“Received name/value pair of %s/%s of type %s\n”, name, value\_to\_string(value), adiak\_type\_to\_string(t));  
}

void init\_tool() {  
 adiak\_register\_cb(1, adiak\_category\_all, cb, 1, NULL);  
}

Unlike the application interface, there is only a C interface for tools.

In this example, the tool receives a callback to cb every time the application uses adiak to register a name/value pair. The value\_to\_string function in this example processes the value, and can be found in Appendix A.

Multiple tools can register to receive Adiak callbacks. Tool can also iterate over and examine the already-set name/value pairs.

## Concepts

As described in the introduction, Adiak is an interface for providing name/values pairs to subscriber tools. This section describes several other important design decisions.

### Memory Management and Data Lifetime

Adiak makes a copy of every data value that is passed through the application interface. This means that:

* It is safe to free data values after passing them to Adiak.
* Adiak is not appropriate for large data structures that consume a significant fraction of memory.

Adiak’s data copies are deep, which means that containers and pointers are followed when doing the copy.

The adiak utility API has calls, such as adiak::clean(), which deallocate all data value copies made by Adiak.

The name C-style strings passed to Adiak are not copied—Adiak only retains a pointer to these strings. The application should ensure that such strings have a program lifetime or through the next adiak::clean call.

The tool interface can retain pointers to Adiak data values, through if it does so the tool should watch for adiak::clean operations to clean those pointers.

### Data Types

The details of Adiak’s type system are covered in the API section, this section covers the high-level concepts.

Adiak needs a type system to manage data values. The application language types are converted to Adiak types, which can then be copied and passed to Adiak tools. In most ways the Adiak type system is less flexible than the application language’s type system, and in other ways it is more specialized.

Adiak only supports a few base type: integrals, floating points, strings. On top of those base types, Adiak provides specialized types that provide more information about the data: filepaths, dates, version numbers, and other specializations in that are described in the API section. All of these types can be placed in containers: ranges, sets, lists, and tuples.

Containers can be nested and mixed. A two-dimensional array of integers could be, for example, a list of a list of integers. Adiak sets, lists, and tuples include the size of the container in the type system—when using the C-style application interface the size of the container will need to be passed along with the data values.

In the C-style interface types are described with a printf-style type string, which is passed along with the value when registering name/value pairs. For example, integers are represent as %d, sets as brackets [], and lists as braces {}.

# API Guide

As discussed above, Adiak has two interfaces: an application interface for registering name/value pairs and a tool interface for receiving name/value pairs. The application interface is divided into an explicit part, where the application registers application-level names and values, and a implicit part that registers system-level name/value pairs with standardized names. In addition, there are initialization and finalization routines for the application interface.

## Data Types

This section describes the data types used by Adiak. These types are used by both Adiak’s application and tool interfaces.

#### adiak\_type\_t

typedef enum {

adiak\_type\_unset = 0,

adiak\_long,

adiak\_ulong,

adiak\_int,

adiak\_uint,

adiak\_double,

adiak\_date,

adiak\_timeval,

adiak\_version,

adiak\_string,

adiak\_catstring,

adiak\_path,

adiak\_range,

adiak\_set,

adiak\_list,

adiak\_tuple

} adiak\_type\_t;

The adiak\_type\_t enum is part of the Adiak type system. It is typically used to describe a simple single type that is part of the adiak\_datatype\_t structure, which describes more complex compound types.

Adiak can encode types as a c-style printf string (see adiak\_namevalue). The following table describes each type and its printf-style encoding character. The C++ interface does not require the printf-style typestring, instead relying on templating for type introspection.

|  |  |  |
| --- | --- | --- |
| Type | Typestring | Description |
| adiak\_type\_unset |  | A place-holder value for uninitialized types. It should not be passed to Adiak. |
| adiak\_long | %ld | A signed long |
| adiak\_ulong | %lu | An unsigned long |
| adiak\_int | %d | An int |
| adiak\_uint | %u | An unsigned int |
| adiak\_double | %f | A double-precision floating point value |
| adiak\_date | %D | A signed long representing the seconds since epoch |
| adiak\_timeval | %t | A struct timeval |
| adiak\_version | %v | A NULL-terminated string representing a version number |
| adiak\_string | %s | An arbitrary NULL-terminated string |
| adiak\_catstring | %r | A NULL-terminated string that should not be alphanumerically sorted (e.g., a githash). |
| adiak\_path | %p | A NULL terminated string that is a file path |
| adiak\_range | <typestring> | A compound type representing a range of values. Another typestring should be passed between the “< >” arrow brackets. The values passed to the C-interface should be passed as two values, v1 and v2, that represent the range as [v1, v2).  To create a range in the C++ interface use the special overloaded version of adiak::value that takes to value parameters. |
| adiak\_set | [typestring] | A compound type representing a set of values. The set is unordered and should not contain duplicate values. Another typestring should be passed between the “[ ]” square brackets. Two values passed to the C interface should be a size integer and a c-style array of the subtype.  The values passed to the C++ interface should be a std::set with the subtype. |
| adiak\_list | {typestring} | A compound type representing a lsit of values. The list order is preserved. Another typestring should be passed between the “{ }” braces. Two values passed to the C interface should be a size integer and a c-style array of the subtype.  The values passed to the C++ interface should be a std::list or a std::vector with the subtype. |
| adiak\_tuple | (typestring1,  typestring2,  typestring3,  …) | A compound type representing a collection of types. N typestrings should be passed between the “( )” parenthesis. In the C-interface, the values should be passed as N values.  The values passed to the C++ interface should be as a std::tuple with N elements. |

#### adiak\_datatype\_t

typedef struct adiak\_datatype\_t {

adiak\_type\_t dtype;

adiak\_numerical\_t numerical;

int num\_elements;

int num\_subtypes;

struct adiak\_datatype\_t \*\*subtype;

} adiak\_datatype\_t;

The adiak\_datatype\_t struct describes a full Adiak type. The adiak\_type\_t above describes a type property, and a compound collection of type properties in an adiak\_datatype\_t describes a full type. If an adiak\_datatype\_t is a compound type, then it will have subtypes.

The dtype member describes the type at this level in the type tree.

The numerical field describes the numerical properties of this data type.

For basic types, the num\_elements field is 0. For compound types, num\_elements describes how many elements are in a list, set or tuple. A range always has two elements.

For compound types, the subtype array points this type’s subtypes. Lists, sets, and ranges have one subtype. Tuples can have any number of subtypes. Simple types have zero subtypes.

The num\_subtypes field is the size of the subtype array.

#### adiak\_value\_t

typedef union adiak\_value\_t {

signed long v\_long;

int v\_int;

short v\_short;

char v\_char;

double v\_double;

void \*v\_ptr;

union adiak\_value\_t \*v\_subval;

} adiak\_value\_t;

The adiak\_value\_t union contains the value part of a name/value pair. It is used with an adiak\_datatype\_t, which describes how interpret the value. The following table describes what union value should be read given an accompanying adiak\_type\_t:

|  |  |
| --- | --- |
| Type | Data Type |
| adiak\_type\_unset |  |
| adiak\_long | v\_long |
| adiak\_ulong | v\_long (cast to unsigned long) |
| adiak\_int | v\_int |
| adiak\_uint | v\_int (cast to unsigned int) |
| adiak\_double | v\_double |
| adiak\_date | v\_long (as seconds since epoch) |
| adiak\_timeval | v\_ptr (cast to struct timeval \*) |
| adiak\_version | v\_ptr (cast to char \*) |
| adiak\_string | v\_ptr (chast to char \*) |
| adiak\_catstring | v\_ptr (cast to char \*) |
| adiak\_path | v\_ptr (cast to char \*) |
| adiak\_range | v\_subval (as adiak\_value\_t[2]) |
| adiak\_set | v\_subval (as adiak\_value\_t array) |
| adiak\_list | v\_subval (as adiak\_value\_t array) |
| adiak\_tuple | v\_subval (as adiak\_value\_t array) |

#### adiak\_category\_t

typedef enum {

adiak\_category\_unset = 0,

adiak\_category\_all,

adiak\_general,

adiak\_performance,

adiak\_control,

} adiak\_category\_t;

The adiak\_category\_t enum is used to classify Adiak name/values into classes of data. Tools can choose to subscribe to only certain categories of data.

The adiak\_category\_unset value is used for uninitialized data and should not be explicitly specified.

The adiak\_category\_all value is only used for tools that are subscribing to data. By specifying this value, the tool receives all categories of data. Applications should not categorize their data as adiak\_category\_all.

The adiak\_control category is used to pass notice of events such as adiak initialization and flushes to tools. It should not be explicitly used by an application.

## Application Interface

This section describes Adiak’s Application interface. It is primarily routines for registering name/value pairs, and for initializing and flushing adiak data.

As previously discussed, Adiak has a C++ and a C interface. The C++ interface is wrapped in an adiak:: namespace. The C-interface functions are prefixed by the string adiak\_.

### Initialization and Finalization

These routines are used to initialize Adiak and flush its data.

#### adiak::init

void adiak\_init(void \*mpi\_communicator\_p);

void adiak::init(void \*mpi\_communicator\_p);

These routines initialize Adiak. One of them must be called by the application before registering name/value pairs.

If MPI is enabled in the application, then the mpi\_communicator\_p parameter should be a pointer to an MPI Communicator, cast into a void\*. MPI\_COMM\_WORLD would typically be the correct communicator to pass. The case occurs to avoid introducing an MPI dependency into Adiak’s header files. The init routine should still be called from each rank in the MPI job.

Adiak should be initialized after MPI\_Init has completed, in an MPI job.

This routine can be safely called multiple times. Subsequent calls have no effect.

#### adiak::fini

void adiak\_fini();

void adiak::fini();

This routine finalizes Adiak. It causes data to be flushed, and values that require end-of-porgram, such as adiak::walltime(), to be generated.

This routine should be called before MPI\_Finalize(), in an MPI job.

If this routine is called multiple times it will re-trigger any flush and recollect end-of-program values. It is safe to register more name/value pairs after and adiak::fini(), though they may require a subsequent flush.

#### adiak::flush

bool adiak::flush();

int adiak\_flush();

This routine flushes data from Adiak, but does not trigger collection of end-of-program metrics.

These functions respectively return true or 0 on success, and false or -1 on failure.

#### adiak::clean

int adiak\_clean()

bool adiak::clean()

This routine frees all heap memory used by adiak. This includes the cache of all name/value pairs passed to Adiak. After this call completes, adiak will report it has no name/value pairs. Adiak should not be used again after this call completes.

These functions respectively return true or 0 on success, and false or -1 on failure.

### Name/Value Interface

This routines in this interface are used to register name/value pairs with Adiak.

#### adiak::value

int adiak\_namevalue(const char \*name, adiak\_category\_t category, const char \*subcategory, const char \*typestr, ...);  
  
template <typename T>

bool value(std::string name, T valuea, T valueb,

adiak\_category\_t category = adiak\_general, std::string subcategory = "");

template <typename T>

bool value(std::string name, T value, adiak\_category\_t category = adiak\_general, std::string subcategory = "");

The adiak::value functions register a Name/Value pair with Adiak. Values are associated with the specified name, and are described by the specified type. In the c-interface version o this function, the type is passed in a printf-style format string, which is used to interpret the varargs parameter that contains the value. In the c++-interface, the value is passed as a template parameter and the type determined from template introspection.

The name parameter is a string that contains the name of the name/value pair. Adiak makes a copy of the string, and memory associated with it can be freed after this call completes.

The category parameter is metadata about this name/value pair describing how it should be categoried. Tools can subscribe to a set of categories.

The subcategory parameter is an optional string that provides an application-generated category string for this name/value pair. Adiak does not interpret this string, but passes it on to underlying tools.

The typestr parameter describes the type of the value passed to Adiak as a printf-style type string. See adiak\_datatype\_t documentation for more information about format characters when specifying this type string. As examples, the typestr could be “%d”, “%s”, or “[%p]” to respectively describe an integer, string, or set of paths value, which are passed in the varargs parameter.

The varargs parameter in the c-interface passes the value. It is interpreted based on the typestr.

In the C++ interface, the value template parameter passes the value that will be associated with this name/value. The type is implied by the template parameter. See the adiak\_datatype\_t documentation describing how C++ types are converted to Adiak types.

The C++ interface that takes two values, valuea and valueb, is used to make values with an adiak type of range.

On success the C++-interface functions return true, and the C-interface function returns 0. On a failure, these calls respectively return false and -1.

#### adiak::user

bool adiak::user();

int adiak\_user();

These routines create an Adiak name of “User” with a adiak\_string value of the name of the person running this process, e.g., “John Smith”.

These functions respectively return true or 0 on success, and false or -1 on failure.

#### adiak::uid

These routines create an Adiak name of “uid” and a adiak\_string value of the username of the person running this process, e.g., “jsmith3”.

These functions respectively return true or 0 on success, and false or -1 on failure.

#### adiak::launchdate

bool adiak::launchdate();

int adiak\_launchdate();

These routines create an Adiak name of “launchdate” and a adiak\_date value of the time when this process was started. It operates on a 1-second granularity.

These functions respectively return true or 0 on success, and false or -1 on failure.

#### adiak::launchday

bool adiak::launchday();

int adiak\_launchday();

These routines create an Adiak name of “launchday” and a adiak\_date value of the time when this process was started, but rounded down to the previous midnight at GMT+0. This can be used to group jobs that ran on the same day.

These functions respectively return true or 0 on success, and false or -1 on failure.

#### adiak::executable

bool adiak::executable();

int adiak\_executable();

These routines create an Adiak name of “executable” and a adiak\_string value of the executable name that launched this process, with the path information stripped.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::executablepath

bool adiak::executablepath();

int adiak\_executablepath();

These routines create an Adiak name of “executablepath” and a adiak\_path value of the executable running this process. A full absolute path is registered.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::libraries

bool adiak::libraries();

int adiak\_libraries();

These routines create an Adiak name of “libraries” and a adiak\_set of adiak\_path values of the dynamic libraries that are loaded into the current process. This does not collect static library information.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::cmdline

bool adiak::cmdline();

int adiak\_cmdline();

These routines create an Adiak name of “cmdline” and a adiak\_list of adiak\_string values of the argv parameters used to create the current process.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::hostname

bool adiak::hostname();

int adiak\_hostname();

These routines create an Adiak name of “hostname” and a adiak\_string value of the hostname of system the current process is running on.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::clustername

bool adiak::clustername();

int adiak\_clustername();

These routines create an Adiak name of “clustername” and a adiak\_string value of the name of the cluster running this job. This value is approximated by stripping numeric information out of the hostname.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::walltime

bool adiak::walltime();

int adiak\_walltime();

These routines create an Adiak name of “walltime” and a adiak\_time value of the amount of walltime this process ran for. This value is not calculated until adiak::fini is called.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::systime

bool adiak::systime();

int adiak\_systime();

These routines create an Adiak name of “systime” and a adiak\_time value of the amount of time the current process spent in system calls. This value is not calculated until adiak\_fini is called.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::cputime

bool adiak::();

int adiak\_();

These routines create an Adiak name of “cputime” and a adiak\_time value of the amount of time the current process spent on the CPU. This value is not calculated until adiak\_fini is called.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::jobsize

bool adiak::jobsize();

int adiak\_jobsize();

These routines create an Adiak name of “jobsize” and a adiak\_uint value of the number of MPI ranks in the current job. Adiak must be initialized with MPI support for this call to work.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::hostlist

bool adiak::hostlist();

int adiak\_hostlist();

These routines create an Adiak name of “hostlist” and a adiak\_set of adiak\_string values of the set of hostnames that are hosting an MPI job. Adiak must be initialized with MPI support for this call to work.

These functions respectively return true or 0 on success, and false or -1 on failure

#### adiak::numhosts

bool adiak::numhosts();

int adiak\_numhosts();

These routines create an Adiak name of “numhosts” and a adiak\_uint value of the number of unique hosts that are part of an MPI job. Adiak must be initialized with MPI support for this call to work.

These functions respectively return true or 0 on success, and false or -1 on failure

### Tool Interface

The tool interface consumes name/value pairs that come from the application interface. Adiak does not specify what a tool does with these inputs. They could output name values into a file, correlate them with other data, or save them in a data store.

This document describes the interface for collecting name/value pairs. Tools can register a callback to receive name/value pairs as they’re passed to Adiak, or they can iterate and examine the existing name/value pairs.

Adiak makes a copy of every name and value passed to it. It is safe to keep pointers to Adiak data, up until a call to adiak::clean (which triggers a adiak\_control callback).

#### adiak\_register\_cb

typedef void (\*adiak\_nameval\_cb\_t)(const char \*name,   
adiak\_category\_t category, const char \*subcategory,  
adiak\_value\_t \*value, adiak\_datatype\_t \*t, void \*opaque\_value);

void adiak\_register\_cb(int adiak\_version, adiak\_category\_t category, adiak\_nameval\_cb\_t nv, int report\_on\_all\_ranks, void \*opaque\_val);

The adiak\_register\_cb call registers a callback function to receive name value pairs. The callback should have a signature matching adiak\_nameval\_cb\_t. It receives name/value, category, value types, and an opaque value passed from adiak\_register\_cb.

The adiak\_version parameter specifies what version of Adiak this tool was written for. Adiak will not deliver callbacks using types (or enums) from future Adiak versions to earlier tool versions.

The category parameter filters callback to a particular Adiak category. If a tool wants to receive all name/value pairs it should pass adiak\_category\_all as this parameter.

The nv parameter is a pointer to the callback function.

If report\_on\_all\_ranks is zero, and Adiak was initialized with MPI support, then only rank 0 of a job will receive these callbacks. Otherwise, all ranks in an MPI job will receive callbacks.

The opaque\_val parameter is forwarded from the adiak\_register\_cb function to the callback. The tool is free to pass arbitrary values through this parameter, or NULL.

#### adiak\_list\_namevals

void adiak\_list\_namevals(int adiak\_version, adiak\_category\_t category, adiak\_nameval\_cb\_t nv, void \*opaque\_val);

This routine iterates over the set of name/values that are already registered with Adiak. Each name/value is passed to a adiak\_nameval\_cb\_t. Unlike adiak\_register\_cb, this routine does not continue to deliver callbacks for future name/value pairs.

deliver callbacks using types (or enums) from future Adiak versions to earlier tool versions.

The category parameter filters callback to a particular Adiak category. If a tool wants to receive all name/value pairs it should pass adiak\_category\_all as this parameter.

The nv parameter is a pointer to the callback function.

The opaque\_val parameter is forwarded from the adiak\_register\_cb function to the callback. The tool is free to pass arbitrary values through this parameter, or NULL.

#### adiak\_type\_to\_string

char \*adiak\_type\_to\_string(adiak\_datatype\_t \*t, int long\_form);

This routine converts an adiak\_datatype\_t to a string.

The t parameter is the datatype that should be converted.

If long\_form is non-zero, the string will be human readable, such as “set of int”. If zero, the string will be in the printf-style format described in adiak\_type\_t documentation, such as “[%d]”.

The returned string is malloc’d. After it is no longer needed, the tool should free the string.