IPS System Programmers Reference Guide

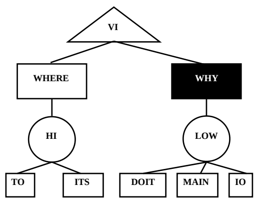
Version 5.0 (October 7, 1992)

Jeff Hollingsworth

Bruce Irvin

Barton P. Miller

University of Wisconsin-Madison



1. Introduction

In addition to its many built-in features, the IPS-2 system includes several interfaces that make it possible for users to extend the tool without having to modify the system. This guide is designed for people who wish to use these extensible features to add additional sources of data or visualizations. This guide is divided into sections that describe each of the external interfaces to IPS-2: External Data Collectors, External Visualization Modules, and thread libraries. In addition, UNIX style man page are provided for each of the interfaces.

For information on getting started with IPS-2 see the IPS-2 User's guide. In addition, an Installation and a Porting Guide are also included with the distribution. We would appreciate hearing about any External Data Collectors or External Visualization you create. We would also appreciate any comment about this document. Please send comments and bug reports to: ips@cs.wisc.edu.

2. Collecting and Viewing External Data

The IPS trace logs document events that occurred within a given program. However, external events can also affect a program's performance. IPS allows the user to view external events in the same histogram and table displays that are used for built-in events. External events are collected with External Data Collection Unit (EDCU) programs, passed to IPS through an external histogram interface, and displayed in the user interface. This section describes EDCUs, how to create them, and how to use IPS to view and analyze external events. A case study on how to improve parallel programs using IPS and external data appears in [. EDCU .].

2.1. What is an EDCU?

An EDCU is simply a program that collects performance data. If an IPS user specifies that an EDCU should be run on a particular machine (Section 4.3), the IPS slave on that machine starts the EDCU before starting any application processes. The EDCU may collect any sort of external data including operating system statistics, information about processes, network statistics, or hardware data. The data is passed back to the master after the application program has completed and may be displayed in the same tables and graphs as built-in or derived metrics. If you are only interested in using existing EDCUs then you may skip to the section on using EDCUs (Section 4.3), but if you wish to create your own EDCUs you should also read section 4.2.

2.2. Creating your own EDCUs

Creating an EDCU is a two part process. First, you must write an EDCU program that collects metrics and stores them in histograms. The EDCU program should use the EDCU library to insure that it correctly obeys the external histogram interface. Second, you must create a metric description file that describes each metric collected by the EDCU. Metric descriptions contain the information that enables IPS to correctly display external data in the user interface.

2.2.1. The EDCU program

Before writing an EDCU program, you should familiarize yourself with the data structures and routines of the IPS EDCU library. The EDCU manual pages are attached as an appendix of this user manual. The *edcu(3)* manual page describes the details of the public data structures and routines of the EDCU library. It includes two examples that show how to use the library routines. The *histogram(5)* manual page describes the file format for external histograms. Normally, an EDCU programmer does not need to understand the file format because the EDCU library routines take care of reading and writing histogram files. The *vmcollect(1)* manual page describes an example EDCU that filters statistics from the output of the system utility **vmstat**. The sources for the vmcollect EDCU are included in the `misc/edcus/vmcollect` subdirectory of the IPS source tree and may be helpful in understanding the issues involved in creating an EDCU.

During initialization, an EDCU must call the routine **EDCinit()** to initialize the EDCU library. After the application program has completed, the slave sends a termination signal to the EDCU. The EDCU library catches the termination signal, writes all of the open histograms to file, and terminates the EDCU.

Data samples are collected in structures known as **histograms**. A histogram data structure is composed of an array of buckets (single precision floating point numbers) and some attributes that are used internally by the EDCU library. The *edcu(3)* manual page gives a full explanation of the histogram data structure.

The manner in which an EDCU collects a metric determines how the metric data should be stored in a histogram. For example, data can be collected as a count of events. This means that each histogram bucket represents the total number of events that occurred during the time interval covered by the bucket. Data that is collected by sampling the value of a resource or a rate should be handled differently. For sampled data, a bucket should represent the value of the "function" being sampled for the time interval covered by the bucket. To accommodate various collection methods, the EDCU library allows you to specify how data is to be deposited in histograms and how histograms should be compacted. Furthermore, the metric description language allows you to specify scaling of histograms so that data stored as event counts may be compared across multiple programs or executions.

Histograms are created with the **HistoCreate()** routine. The *edcu(3)* manual page describes this routine in detail and provides examples that show how to use it in an EDCU. Here we only explain the most critical parameters.

The *node* parameter of **HistoCreate()** points to a **HistNode** structure that describes the node in the IPS program tree to which the histogram should be attached. The four fields of a **HistNode** structure may be filled with the name of a node at each of the corresponding four levels of the IPS program tree. For example, to attach a metric to the machine node *topaz* for the application *myprogram*, you would fill the **program** field of the **HistNode** structure with the (null-terminated) string "`myprogram`", and the **machine** field with the string "`topaz`". You would simply set the **process** and **procedure** fields to NULL. Similarly, you may set any field of a **HistNode** structure to NULL if you do not wish to specify the node at that level. For example, if you want a metric to be attached to the machine node corresponding to the current machine of the current application, then set the **program** and **machine** fields (as well as the **process** and **procedure** fields) of the **HistNode** structure to NULL. The default level of a metric is determined by the metric's definition. For more instructions on specifying the default *level* of a metric as opposed to the specific tree node, see the section 4.2.2.

The number of buckets in a histogram, specified by the *numbuckets* parameter of **HistoCreate**, never changes once a histogram is created. When data samples run off of the end of a histogram, the EDCU library automatically compacts the histogram data into the first *numbuckets/2* buckets by combining pairs of adjacent buckets. The *compact* parameter of **HistoCreate()** controls how adjacent buckets are combined during compaction. Setting the *compact* parameter to *H_SUM* causes adjacent buckets to be summed, and *H_AVG* causes adjacent buckets to be averaged. For metrics that are stored as event counts, use *H_SUM* for *compact*, and for metrics that are stored as functions over time, use *H_AVG*.

Use the **HistoSample()** routine to deposit data samples into histograms. When depositing a sample, you may provide one or two timestamps. If you provide one timestamp, then the data is deposited as an instantaneous sample that will be deposited in exactly one bucket. If you provide two timestamps, then the data is considered to be an interval sample and may be deposited in several contiguous histogram buckets. The *edcu(3)* manual page includes programming examples that show how to use both sampling methods.

The *addmode* parameter of **HistoSample()** specifies exactly how the data is to be deposited into the buckets. For instantaneous samples, an *addmode* of *H_ADD* specifies that the data is to be added to the histogram bucket, and an *addmode* of *H_SET* specifies that the bucket should be set equal to the data. For metrics stored as event counts, use *H_ADD* when depositing instantaneous samples, and use *H_SET* for metrics that are stored as functions over time. For interval samples, an *addmode* of *H_SPLIT* specifies that the data is to be split evenly over the buckets in the interval, and an *addmode* of *H_SET* specifies that all the buckets in the interval should be set equal to the data. For metrics stored as event counts, use *H_SPLIT* when depositing interval samples, otherwise use *H_SET*.

When the application running under IPS has finished, the slave signals each EDCU, and the EDCUS write all open histograms to file. You may provide a *wrapup routine* to handle any shutdown procedures for your EDCU. For example, an EDCU may need to close files and connections, release semaphores, or reset devices to gracefully terminate. If your EDCU has a wrapup routine, pass a pointer to the routine as the parameter to the **EdcInit()** routine (otherwise be sure to pass a NULL to **EdcInit()**). The EDCU library will call the wrapup routine before it writes the histograms to file.

2.2.2. Creating a Metric Description File

The second major step in creating an external data collector is to create a metric description file. A metric description file includes a description of each metric collected by an EDCU and a description of each y-axis unit used by the metrics. Each metric description is written in a block structured form and contains information that describes the metric to IPS. Here is an example:

```
metric syscalls {
    class external;
    level machine;
    title "System Calls";
    xlabel "(Syscalls/Sec)";
    scalefactor 1/bucketwidth;
    aggregation sum;
    units ips_ops_sec;
    tablecalc sum;
    tabletitle "System Calls (sum)";
}
```

The **class**, and **level** attributes must be set in every metric description. The **class** attribute defines the class of the metric (base, external, or derived). For EDCU metrics, the **class** attribute is `external`. The **level** attribute indicates the default level of the metric. Valid values for **level** are `program`, `machine`, `process`, and `procedure`, corresponding to the levels of the program tree.

The rest of the metric attributes may be set in a metric description or may be omitted. If any of the attributes are omitted, the master will attempt to assign reasonable defaults. The **title** attribute is the string used in histogram menus and displays. The **xlabel** attribute specifies the metric's label on the x-axis of the histogram displays.

The **scalefactor** attribute applies to metrics that are stored as event counts and specifies how those metrics should be scaled in histogram displays. The master multiplies each bucket by the metric's scale factor just prior to displaying a histogram. The scale factor is needed when comparing between different runs or programs because there is no guarantee that the histogram bucket resolution will be equal between different runs or different programs. The constant `bucketwidth` equals the width (in seconds) of a single histogram bucket. By scaling each bucket by `1/bucketwidth` we can plot the metric as an event rate. For metrics that store time based data (e.g. CPU time, System time) as event counts, the convention is to use `100/bucketwidth` as the scale factor. For a metric stored as a function over time, no scale factor is generally needed and the **scalefactor** attribute may be omitted. The **scalefactor** attribute does not affect the computation of a metric's table value.

The **aggregation** attribute specifies how a metric at a given level of the IPS program tree should be aggregated at higher levels. Setting **aggregation** to `none` turns off aggregation for the metric, while `sum` specifies that the metric should be aggregated at higher levels by summing the values at lower levels, and `average` specifies that the metric should be averaged from lower levels. The **units** attribute specifies the units of the metric. A **units** type is defined separately, and is used primarily to label the y-axis of the histogram displays. Any number of metrics may use the same units and thus share a common y-axis. Here is an example of a **units** definition:

```
units ips_ops_sec {
    ylabel "Ops/sec";
}
```

The **tablecalc** attribute controls the calculation of table values from histograms (either `sum` or `average`). Finally, the **tabletitle** attribute controls the metric's title for table menus and displays.

2.2.3. Quick guide to key parameters

The following table is useful for determining which parameters to use for creating, sampling, and describing metrics that are stored as event counts vs. metrics that are stored as sampled functions.

Metric Type	HistoCreate(compact)	Sample(addmode)	Description(scalefactor)
Event Count	H_SUM	H_ADD or H_SPLIT	`1/bucketwidth`
Sampled Function	H_AVG	H_SET	

2.3. Using an EDCU

Running an EDCU from IPS is as simple as running a process. Just follow the steps for creating a process as in section 2.2.1, with the exception that you must set the "**process or edcu**" button to **edcu**. The EDCUs will appear in the program tree as octagons. Also, you must include the EDCU's metric description file at the top of your IPS script file. To do this, save your script file and use your text editor to add a line to your script file that looks like the following:

```
#include "metric_description_file"
```

Here, `metric_description_file` is the name of the metric description file for the EDCU. Finally restart IPS with your script file.

When you choose "**Run Program**" from the edit tree banner menu, IPS will run the program and the EDCUs, collect the trace data as well as the external metrics, and attach the external metrics to the proper nodes of the tree. You will find the external metrics in the same menus as the built-in metrics.

3. User Defined Displays

Currently, IPS can display data in many ways. However there is a variety of different kinds of performance data, and there are many different ways to display these data, so IPS-2 provides programmers the ability to obtain data from IPS-2 and display however they wish. Such programs are called an External Visualization Units (EVUs). From the IPS-2 user's standpoint, an EVU looks just like a built-in display. This section discusses how to write an EVU and how to describe your EVU to IPS-2.

3.1. Writing an EVU

Writing an EVU requires two steps: writing the EVU program, and describing the EVU to IPS-2. To assist the programmer in writing EVU programs, the IPS-2 EVU library is provided. The EVU library contains a set of routines that can be used to obtain data from the IPS-2 master. The library is described in detail in the *evu(3)* manual page, which is provided as an appendix to this document. The manual page also includes some simple example programs which demonstrate how the routines might be used in an EVU. You should familiarize yourself with the routines and types described in that document before continuing.

3.1.1. Error Handling

For most of the routines in the EVU library, it is possible that an error of some kind may be encountered. In these cases, the routine will return NULL, and the global error indicator **EVU_errno** will be set to indicate the kind of error that was encountered. A complete list of possible errors and their causes is included in the *evu(3)* man page.

3.1.2. EVU Library Routines

The types used by the EVU library are defined in the header file *evu_types.h*, and the interface routines are prototyped in *evu_lib.h*. The EVU need only include *evu_lib.h*, as this file includes evu_types.h.

A *metric descriptor* is a data structure which contains information about a metric, such as its name, labels used to display the metric, the units in which the metric is measured, and other information. An EVU can request a metric descriptor with **EVU_GetDesc()**. Any metric name found in the system-wide IPS-2 configuration file can be passed to EVU_GetDesc. The name of a derived or external metric which is known to IPS-2 can also be used. In addition to being useful when displaying a metric, a metric descriptor can be used to request metrics, as mentioned below in the description of EVU_GetThisMetric.

There are three interface routines which request metric information from the IPS-2 master: **EVU_GetOneMetric()**, **EVU_GetMetrics()**, and **EVU_GetThisMetric()**. EVU_GetOneMetric and EVU_GetMetrics request the master to bring up a property sheet listing all metrics defined for the currently selected node. For EVU_GetOneMetric, the user is allowed to select a single metric from the property sheet. For EVU_GetMetrics, the user is allowed to select any number of metrics. The selected node and metric(s) are then returned to the EVU. If no node is selected when this call is made, then these routines return NULL, and EVU_errno is set to reflect the error.

EVU_GetThisMetric allows the EVU to select a specific metric, and it also allows the EVU to request metric data for a specific node or for more than one node. To select a node, the EVU passes a *NodeName* structure as the *name* parameter. A node name consists of a level, which corresponds to the level of the node in the IPS-2 tree, and a list of components, which are the labels on the nodes from the root of the tree to the node. The *level* can be assigned any of the constants: *IPS_PROGRAM*, *IPS_MACHINE*, *IPS_PROCESS*, and *IPS_PROCEDURE*, defined in the header file *evu_types.h*, to indicate the appropriate level. Only the first *level* components of a node name are significant; the others are ignored.

To select a metric for more than one node, the EVU can use the wildcard name component *ANY* for one or more of the node name components. Thus, the *name* parameter should be thought of as a template. The IPS master expands such names into a list of names which match the template. If no name is given (the *name* parameter is NULL), then the master will select metrics for the currently selected node.

To request a specific metric, the EVU passes a metric descriptor as the *desc* parameter. If no descriptor is given (the *desc* parameter is NULL), then the master will bring up a property sheet from which the user can select one metric. If the given node name expands to more than one node (i.e., if it contains a wildcard component), then the property sheet will list all metrics which are defined for all of the nodes implied by the node name. If neither a node name nor a metric descriptor are given, then the call is equivalent to a call to EVU_GetOneMetric.

Another routine which can be used to request information from the master is **EVU_GetSelectedNode()**, which returns a node name structure corresponding to the currently selected node. If there is no node selected, then NULL is returned, and EVU_errno is set to reflect the error.

To get additional information about a certain node, the **GetNodeInfo()** routine is provided. The node name argument may contain wildcards, as long as it uniquely specifies a node. Otherwise, NULL is returned, and EVU_errno is set to EVU_NOTUNIQUE. This routine returns miscellaneous information about the given node (and possibly its parents), such as the pid for a process, or the directory in which a program was executed.

Once the EVU has finished making requests of IPS, it can call **EVU_Close()** to terminate the connection to the IPS master. It is not necessary to make this call, however, as the connection will be terminated when the EVU exits. An EVU which (for some reason) is short of file descriptors may choose to call this routine once it has received all necessary metrics, as it will free up the descriptors used for the communication with the master.

The EVU library contains a number of help functions which may be of use in labeling displays and generating helpful error messages. The first of these functions is **EVU_perror()**, which is similar to the UNIX *perror(3)* routine. EVU_perror generates an error message corresponding to the last error encountered by the EVU library on the standard output.

Likewise, **EVU_strerror()** is similar to the *strerror(3)* routine. EVU_strerror returns a pointer to a string corresponding to the last error encountered by the EVU library. The returned pointer points into the static data area of the EVU library, so it is important that the EVU not attempt to modify the contents of this string.

The **init_nodename()** routine is a convenience function which simplifies initializing a node name structure by assigning the *level*, *program*, *machine*, *process*, and *procedure* parameters to their appropriate locations in the given node name structure.

The **fprint_nodename()** and **sprint_nodename()** routines are useful for printing node names in a standard format. Fprint_nodename prints the given node name to the given output stream, and sprint_nodename formats the nodename the same way as fprint_nodename, but copies the formatted string into the passed-in buffer. If a NULL pointer is passed in as the *buffer* parameter, then a buffer of the appropriate size is allocated and used.

Finally, there are two global arrays which may also be of use in labeling displays: **ips_levels**, and **Ips_levels**. These arrays contain the standard labels used for the levels of the IPS tree. The labels in the former are lowercase, where they are capitalized in the latter. For example, `Ips_levels[IPS_PROCESS]` is the string "Process".

3.2. Describing an EVU to IPS-2

To describe an EVU to IPS-2, you need to add a entry to the IPS script file. Here is an example of an EVU definition:

```
evu "EVU name" {
    command "EVU startup command";
}
```

In this example, `EVU name` is the name of the EVU as you would like it to appear in the EVU menu, and `EVU startup command` is the shell command that IPS-2 should issue to start the EVU. This command will be passed to *sh(1)*, so any legal sh command can be used here. For example, an EVU which draws kiviat graphs might have an entry like the following:

```
evu "Kiviat" {
    command "../evus/kiviat";
}
```

Any EVUs defined in this way will show up in the EVU menu the next time IPS is started. For a description of how an EVU can be started up from the IPS master, see sections 2.3 and 2.3.6.

4. Using IPS-2 with Thread Packages

User level thread packages are an important part of many parallel programming environments. However, since these threads are not visible to the kernel, IPS-2 can not keep tack of them. However, IPS-2 provides a set of library calls that can be inserted into a thread package to make the tool work in this environment. These procedures tell IPS-2 about the creation, scheduling, and destruction of threads. When these calls have been inserted, each thread looks like a UNIX process to IPS-2. A thread will contain a process level node in the program tree, and all of the normal IPS-2 metrics can be computed on a thread by thread basis. Most IPS-2 user's need not worry about thread instrumentation. All that is required to run a thread based program under IPS-2, is to link the application with the instrumented version of thread library being used (The exact way to do this varies with thread packages, and you should consult the person who maintains your thread package for the correct commands).

Each underlying UNIX process is treated as a thread. The process thread is defined to be running when no application thread is scheduled on the physical process. This includes execution time in a process before a thread is first assigned to it.

`IPSinitThread()`

Initialize the thread support library. This must be called exactly once per **application** (e.g. not once per thread or process).

`int IPSnewThread(name)`

`name` is a name for the thread (or `NULL` if no name is desired). `IPSnewThread` returns a unique id for the thread. Creating a thread does **not** imply that it becomes the currently active thread. To activate a thread, call `IPSdispatchThread` with the desired thread id.

`IPSdefineProcessName(name)`

Define the name of a process to be `name`. This is not required by most thread libraries because the `name` argument to `IPSnewThread` provides the same functionality.

`IPSdispatchThread(id)`

`id` is the `threadId` for the thread that is starting to run.

`IPSterminateThread(id)`

`id` is the `threadId` for the thread that is being terminated.

`IPSterminateAllThreads()`

Terminate all existing threads. This is useful at the end of a program's execution.

`IPSdefineAllocations(smallAlloc, smallFree, pageAlloc, pageFree, sharing)`

`IPSdefineAllocations` defines the heap allocation routines used by the IPS-2 thread library. `smallAlloc` is a pointer to a function that allocates a small amount of memory (less than one page). `smallFree` is a pointer to a function that free memory allocated by the the `smallAlloc` function. `pageAlloc` is a pointer to a function that allocates page aligned data. The desired size is expressed in bytes (not pages). `pageFree` is a pointer to a function to free memory allocated by `pageAlloc`. `sharing` is either `TRUE` or `FALSE` and indicates if the memory returned by the allocators will be in shared memory after a process (not a thread) forks. By default the library uses malloc and free (or shared mallocs on multi-processors). However, some thread libraries provide their own allocation routines that must be used.

Notes

Threads will be assigned "process ids" outside the range of valid process id's for normal Unix processes. These are needed to maintain the naming convention for trace logs.

The implementation assumes that a hardware synchronized clock exists on each processor being used (e.g. the memory mapped wallclock on a Sequent Symmetry).

Appendix 1: Manual Pages

The following manual pages have been included: *imondump(1)*, *edcu(3)*, *evu(3)*, and *histogram(5)*.

NAME

imondump – a utility to print the contents of IPS-2 trace logs

SYNOPSIS

imondump

DESCRIPTION

is a utility to print out the contents of trace logs produced by the IPS-2 parallel performance tool. There are several command line options to

–a tells imondump that the trace files are auxiliary trace logs (imonaux????).

–s prints only a summary of the trace log. This option is useful to find out how many of each type of trace record is in a given trace log.

–S **indicates that all trace records before** *time* **should not be printed, or counted (-s option).**

–**E** **indicates that all trace records after** *time* **should not be printed or counted (-s option).**

SEE ALSO

IPS-2 user's guide, IPS-2 installation guide.

BUGS

An accurate stack is not kept for **-S** limited printing. This can cause some return records to not list the correct function.

AUTHOR

Jeff Hollingsworth - UW-Madison, IPS-2 Project

NAME

edcu – routines for creating an IPS External Data Collection Unit.

SYNTAX

#include "edcu.h"
#include <time.h>

EdcInit(finishfunc)
void (*finishfunc)();

EDCtime EdcNow()

EDCtime TvToEDCtime(tv)
struct timeval tv;

struct timeval EDCtimeToTv(tim)
EDCtime tim;

EDCtime UsecToEDCtime(UsecStamp)
unsigned long UsecStamp;

Histogram *HistoCreate(filename,numbuckets,width,metricname,node,compact)
char *filename;
int numbuckets;
EDCtime width;
char *metricname;
HistNode *node;
int compact;

HistoDestroy(H)
Histogram *H;

HistoSample(H,Stamp1,Stamp2,data,addmode)
Histogram *H;
EDCtime *Stamp1,*Stamp2;
float data;
int addmode;

HistoWrite(H)
Histogram *H;

HistoLastStamp(H)
Histogram *H;

DESCRIPTION

These routines will help you create an External Data Collection Unit (EDCU) for IPS. An EDCU collects system information in the form of time based histograms. An EDCU is usually designed to loop indefinitely, collecting statistics until it is halted asynchronously by an IPS slave analyst.

The header file *edcu.h* defines three data types for this package:

```
typedef double EDCtime;            /* unit of time = one second */

typedef struct {
  char *program;
  char *machine;
  char *process;
  char *procedure;
} HistNode;

typedef struct {
  char *filename;
  char *metricname;
```

```
  EDCtime starttime;
  EDCtime stoptime;
  EDCtime lasttime;
  EDCtime bucketwidth;
  int numbuckets;
  int maxbucketused;
  int firstrecvd;
  int compaction;           /* H_AVG, ... */
  HistNode node;
  float *buckets;
} Histogram;
```

The members of the Histogram structure are:

filename	File to which the histogram will be written.
metricname	Name of the metric being collected.
starttime	Start of the histogram.
stoptime	End of the histogram.
lasttime	Time of most recent data sample.
bucketwidth	Width of each histogram bucket.
numbuckets	Number of buckets in histogram.
maxbucketused	Bucket index of latest sample.
firstrecvd	Has the first data sample for the histogram been received?
compaction	Method to be used to compact the histogram.
node	IPS tree node to which the histogram will be attached.
buckets	Array of histogram buckets.

The *EdcInit* routine initializes the EDCU. Your EDCU should call this routine at the beginning of execution. When the EDCU is halted, the function *finishfunc* will be called. If *finishfunc* is NULL, no function will be called when the EDCU is halted. The last thing that the EDCU library will do when the EDCU is halted is write all histograms and exit.

The *EdcNow* routine returns a current timestamp. It obtains timestamps in the fastest manner possible, whether through a hardware timer (usclk), a memory mapped clock, or a system call (gettimeofday).

The *TvToEDCtime* and *EDCtimeToTv* routines convert timestamps between **EDCtime** and **struct timeval time formats.**

The *UsecToEDCtime* routine returns the **EDCtime** equivalent of a microsecond timestamp. This routine detects and accounts for periodic wraps in the hardware microsecond counter.

HistoCreate creates a histogram for the metric *metricname* with *numbuckets* buckets, each of width *width.* When the histogram is written with the **HistoWrite** routine, it will be written to the file **IMONDIR** */filename.* **IMONDIR** is an environment variable that should be defined by the user. If it is not defined, then it will be set to the environment variable **HOME.** If **HOME** is not defined then **IMONDIR** will be set to the current directory. The *width* argument should approximate the sampling period of the EDCU for this particular histogram. However, the *width* argument is only a suggestion, and the actual bucket widths may be increased if necessary. The *numbuckets* value will never be changed by the **edcu** library. If you furnish nonpositive values for either of these two arguments, **HistoCreate** will choose reasonable defaults for you.

The *node* argument will be blindly copied to the output histogram file. It is to be used to instruct IPS where to insert the histogram in the user's program tree. Any field of the node structure that is unknown or not applicable should be set to NULL.

The eventual interpretation of the *node* argument depends on the level defined for the histogram's metric. (The metrics themselves are defined in metric description files -- user guide) If the metric is a program level metric, then IPS will only look at the *program* member of the *node* argument. If the *program* member is NULL, then IPS will assume that the histogram refers to the current program. Similarly, the *machine* member should only be filled if the metric is a machine (or lower) level metric that refers to a machine other than the current machine. The *process* member of the *node* argument should be filled for process level and procedure level histograms. It indicates which process is associated with the histogram. For procedure level histograms, both the *procedure* and the *process* members of the *node* argument should be filled. In this case the *procedure* member indicates the procedure node to which the histogram will be attached.

The *compact* argument controls how adjacent buckets will be combined when the histogram is compacted. Whenever a sample is taken beyond the end of the histogram. The *numbuckets* buckets of the histogram will be compacted into *numbuckets/2* by combining adjacent pairs of buckets into one bucket. If *compact* is **H_SUM** then adjacent buckets will be summed to create a new bucket, and if *compact* is **H_AVG** then adjacent buckets will be averaged.

HistoDestroy frees all space allocated for a histogram.

Use the *HistoSample* routine to enter data into a histogram. The timestamps *Stamp1* and *Stamp2* indicate an interval over which the histogram's metric value was *data.* If *Stamp2* is NULL, then the sample is assumed to be instantaneous at time *Stamp1. HistoSample* detects and repairs timestamp glitches that are sometimes introduced when the system clock is read during an update.

The parameter *addmode* controls how the data is added to the histogram when the sample is taken over an interval (ie. *Stamp2* is not NULL). If *addmode* is H_SET then each bucket in the interval is set to *data.* If *addmode* is H_ADD then each bucket in the interval is incremented by *data.* If *addmode* is H_SPLIT then *data* is split and added evenly to all of the buckets in the interval. Buckets that only partially overlap the interval receive partial amounts of *data* according to the interval's overlap in the bucket. If *addmode* is H_SPLIT_AND_SET then *data* is split and used to set all of the buckets in the interval. Partial overlaps are handled as with H_SPLIT.

The *HistoWrite* routine writes the histogram to its file. The routine only writes out the buckets up to the latest sample. This routine also updates the external histogram directory for the EDCU. The external histogram directory will be written to the file **IMONDIR** */EDCdir.<pid>*. The external histogram directory will be used by the IPS slave analyst to discover all of the histograms for this EDCU.

The routine *HistoLastStamp* returns the most recent time stamp for a histogram. This is useful for EDCUs that collect data over consecutive intervals (see second example). If there have not been any prior timestamps, then **HistoLastStamp** returns **EDCtime_ERROR.**

EXAMPLES

The following code outlines an example EDCU that records instantaneous samples at regular intervals. The code assumes the availability of a function (HowManyFreePages()) that returns the number of free pages in the system.

```
#include <stdio.h>
#include "edcu.h"

extern float HowManyFreePages();

#define INTERVAL 1

main()
{
Histogram *Histo;
float FreePages;
EDCtime stamp;
HistNode node;

  node.program = node.machine = node.process = node.procedure = NULL;

  EdcInit(NULL);

 /* Create a histo with 1000 buckets, each 1 second wide */
  Histo = HistoCreate("FP.histo",1000,1.0,"freepages",&node,H_AVG);

 /* Loop indefinitely */
  while(1) {
   /* Get the data */
    FreePages = HowManyFreePages();

   /* Get the timestamp */
    stamp = EdcNow();

   /* Add the data to the histogram */
    HistoSample(Histo,&stamp,NULL,FreePages,H_SET);

    sleep(INTERVAL);
  }

}
```

The next example outlines an EDCU that collects samples over consecutive intervals. Note that this EDCU defines its own finish function, called *MyFinish()*. This example assumes the availability of two functions, *InitSwitchesProbe()* and *SwitchesSinceLasttime()*. InitSwitchesProbe() is assumed to initialize a system probe that counts context switches, and SwitchesSinceLasttime() is assumed to return the number of context switches since the last time either of the two functions was called.

```
#include <stdio.h>
#include "edcu.h"

extern InitSwitchesProbe();
extern float SwitchesSinceLasttime();

Histogram *histo;

void MyFinish()
{
  HistoWrite(histo);
}

main()
{
float Switches;
EDCtime stamp,lastStamp;
HistNode node;

  node.program = node.machine = node.process = node.procedure = NULL;

  EdcInit(MyFinish);

 /* Create a histo using defaults */
  histo = HistoCreate("CSwitch.histo",-1,-1.0,"cswitches",&node,H_SUM);

 /* Start the probe */
  InitSwitchesProbe();

 /* Do first sample */
  stamp = EdcNow();
  HistoSample(histo,&stamp,NULL,0.0,H_SET);

 /* Loop indefinitely */
  while(1) {
   /* Get the data */
    Switches = SwitchesSinceLasttime();

   /* Get the timestamp */
    stamp = EdcNow();
    lastStamp = HistoLastStamp(histo);

   /* Add the data to the histogram over the interval */
    HistoSample(histo,&lastStamp,&stamp,Switches,H_SPLIT);
  }

}
```

NOTES

Link load your EDCU with libedcu.a

Do not compile your EDCU with the ips instrumentation (compiler switch -pi).

If an EDCU exits before being halted by a slave, then the EDCU must exit with a status value of 0.

SEE ALSO

gettimeofday(2), usclk(4), getusclk(3), vmcollect(1), histogram(5)

NAME

evu – routines for creating an IPS External Visualization Unit.

SYNTAX

#include "evu_lib.h"

void EVU_Init()

MetricDesc *EVU_GetDesc(name)
char *name;

Metric *EVU_GetOneMetric(type)
int type;

Metric *EVU_GetMetrics(type, nmetrics)
int type;
int *nmetrics;

Metric *EVU_GetThisMetric(desc, name, type, nmetrics)
MetricDesc *desc;
NodeName *name;
int type;
int *nmetrics;

NodeName *EVU_GetSelectedNode()

NodeInfo *EVU_GetNodeInfo(name)
NodeName *name;

void EVU_Close()

void EVU_perror(str)
char *str;

char *EVU_strerror(errnum)
int errnum;

void init_nodename(name, level, program, machine, process, procedure)
NodeName *name;
int level;
char *program, *machine, *process, *procedure;

void fprint_nodename(stream, name, level)
FILE *stream;
NodeName *name;
int level;

char *sprint_nodename(buf, name, level)
NodeName *name;
char *buf;
int level;

DESCRIPTION

These routines will help you create an External Visualization Unit (EVU) for IPS. An EVU can display IPS data in formats not otherwise provided by IPS. The EVU library provides an interface to IPS, which allows an EVU to request and retrieve data from IPS.

The header file *evu_types.h* (which is included in defines six data types for this package, as well as some useful constants and external variables:

```
typedef float Bin;   /* Type of histogram bins */

#define IPS_NLEVELS  4   /* Number of levels in the IPS tree */

/* Types of metrics */
#define HISTO   0    /* Only histograms for now */
```

```
typedef struct{
    int level;
    char *components[IPS_NLEVELS];
} NodeName;

typedef struct {
    unsigned flags;
    int passive;
    char *login;
    char *command;
    char *execdir;
    char *tagsdir;
    int pid;
} NodeInfo;

typedef struct{
    char *name;
    char *title;
    int mid;
    char *unit_label;
    int unit_id;
} MetricDesc;

typedef struct{
    float time_per_bin;
    int numbins;
    Bin *bins;
} HistoData;

typedef union{
    HistoData hdata;
} MetricData;

typedef struct{
    MetricDesc *desc;
    NodeName *name;
    int type;
    MetricData *data;
} Metric;

extern char *ips_levels[];

extern char *Ips_levels[];

extern int EVU_errno;
```

Interface Types

A **NodeName** structure contains the name of a node in the IPS tree. Its fields are:

level — Indicates the level of the tree at which the node is found.

components — An array consisting of the labels of the nodes found on a path from the root of the tree to the node. The constants *IPS_PROGRAM*, *IPS_MACHINE*, *IPS_PROCESS*, and *IPS_PROCEDURE* can be used as indices into this array to access the various name components.

A **NodeInfo** structure contains additional information about a node.. Its fields are:

flags — Status flags pertaining to the node. Currently, only IPS_VISIBLE is supported, and indicates whether the node is currently hidden in the IPS user interface. example, ‘‘ips_cpu_time’’).

passive — A boolean flag, indicating whether the program was run in passive mode.

login — The login name used for the machine on which the selected node was run. NULL for nodes at the *IPS_PROGRAM* level.

command — The command line used to execute the process. NULL for nodes at the *IPS_PROGRAM* or *IPS_MACHINE* levels.

execdir — The execution directory for the process. NULL for nodes at the *IPS_PROGRAM* or *IPS_MACHINE* levels.

tagsdir — The directory in which a tags file can be found corresponding to the source code for the requested node. NULL for nodes at the *IPS_PROGRAM* or *IPS_MACHINE* levels.

pid — The pid associated with the requested node. NULL for nodes at the *IPS_PROGRAM* or *IPS_MACHINE* levels.

The **MetricDesc** structure contains general information about a metric. Its fields are:

name — The name under which the metric is defined in the IPS configuration file (for example, ‘‘ips_cpu_time’’).

title — A string used to identify the metric when it is displayed (for example ‘‘CPU Time’’).

mid — A unique, integer which is used to identify the metric. An EVU should never need to use this value, except perhaps to determine whether two metrics are the same.

unit_id — A unique, integer which identifies the units in which the metric is defined. The unit_id can be used to determine if two metrics are measured in the same units (and should be plotted on the same scale).

unit_label — The string used to label the units (ex. ‘‘%CPU’’).

The EVU library retrieves histogram data in the form of a **HistoData** structure. Its fields are:

time_per_bin — The amount of time (in seconds) represented by each histogram bin.

numbins — The number of bins in the histogram.

bins — The array of histogram bins.

A **MetricData** structure can hold metric data of any type. Currently, the EVU library only supports histogram data, so MetricData is a union with only one field.

The **Metric** structure describes an instance of a metric, i.e., the pairing of a metric and a node of the IPS tree. Its fields are:

desc — The descriptor of the metric.

name — The node of the IPS tree for which the data was retrieved.

type — The type of the metric (currently, the only valid value for this field is *HISTO*).

data — Points to the MetricData structure which contains the data of the metric.

Interface Routines

The `EVU_Init` routine initializes the EVU. It is not crucial that your EVU call this function, as the EVU library will automatically call it when necessary.

The `EVU_GetDesc` routine returns the descriptor of the named metric. Note that the *name* parameter is the name, not the title, of a metric. For more information on metric names, consult ???

The `EVU_GetOneMetric` routine is the simplest way for an EVU to retrieve metric data from IPS. The *type* parameter should always be *HISTO*. This routine requests the master to bring up a property sheet listing all metrics defined for the currently selected node, from which the user may select one metric. This routine returns a pointer to a **Metric** structure which provides the name of the currently selected node, the descriptor for the chosen metric, and the metric data.

The `EVU_GetMetrics` call is similar to `EVU_GetOneMetric`. The only difference is that the user may select several metrics from the property sheet. This routine returns an array of Metric structures, and *nmetrics* is set to the number of Metrics in the array.

Using the `EVU_GetThisMetric` routine, the EVU can ask for data for a specific metric, data for more than one node, or both. The *desc* parameter is the descriptor of the requested metric, as returned by `EVU_GetDesc`. If this parameter is NULL, then the master will bring up a property sheet from which the user can select one metric. The *name* parameter names the node or nodes for which the metric is to be retrieved. If this parameter is NULL, then the metric is retrieved for the currently selected node. Currently, the *type* parameter must always be *HISTO*.

It is possible to request data for several nodes by setting one or more of the components of *name* to the wildcard constant *ANY* (defined in *evu_lib.h*). Thus, this parameter should be thought of a template, in that data for the requested metric will be retrieved for all nodes that match the template, and for which the given metric is defined. If *desc* is NULL, then the property sheet generated by the master will list all metrics that are defined for all of the nodes that match the template. Note that all the nodes that match a given template will be at the same level. This routine returns an array of **Metric** structures, and *nmetrics* will be set to the length of the array. Note that if both *desc* and *name* are NULL, then this function is equivalent to `EVU_GetOneMetric`.

The *EVU_GetSelectedNode* function returns a **NodeName** structure corresponding to the name of the currently selected node.

The *EVU_GetNodeInfo* function returns a **NodeInfo** structure containing some miscellaneous information about the requested node. In some cases, the fields will be filled in with the data for requested node's parent. For example, it doesn't make sense to ask for the pid of an *IPS_PROCEDURE*, but the pid field will contain the pid of the process in which the procedure executed. If the appropriate field cannot be determined from any of a node's parents, NULL is returned. For example, if information is requested about an *IPS_MACHINE* node, the pid field would be NULL. **NodeName** structure corresponding to the name of the currently selected node.

EVU_Close terminates the connection with the master. Because the master will be notified when the EVU exits, it is not necessary for the EVU to make this call. It is provided in case the EVU wishes to terminate the connection before exiting.

EVU_perror prints its *str* argument on the diagnostic output followed by a message corresponding to the last error encountered by an EVU library routine. It is analogous to *perror(3).*

EVU_strerror returns the error message corresponding to error number (analogous to *strerror(3)*). This function can be used to format error messages to be printed somewhere besides the diagnostic output. Note that

```
EVU_perror("Error in EVU library");
```

is equivalent to

```
fprintf(stderr, "Error in EVU library: %s\n",
        EVU_strerror(EVU_errno));
```

Init_nodename Initializes a node name with the given values.

The *fprint_nodename* routine formats *name* and prints the result to the given stream. This function inserts a colon ‘‘:’’ character between each consecutive pair of labels. If any of the components of the name are then they are formatted as ‘‘*’’. If *level* is less than then the name will be formatted as a name of level This feature can be used to generate the name of one of the node's ancestors. *fprint_nodename* does not print a newline character after the name.

sprint_nodename formats the given node name as per *fprint_nodename,* leaving the result in If *buf* is NULL, then a buffer of the appropriate size is allocated with *malloc(3).* The formatted buffer is returned.

EVU_errno, analogous to *errno(2),* is a global error indicator which is set whenever an error is encountered by an EVU library call. This value can be used in calls to *EVU_strerror* to generate an error message.

The global variables *ips_levels* and *Ips_levels* are arrays of strings corresponding to the levels of the IPS tree. The strings in the former are all lowercase, and the strings in the latter are capitalized. These arrays can be indexed by the *level* field of the **NodeName** structure, or by the constants and

RETURN VALUES

On error, all of the functions that normally return structures will return NULL, with the global constant EVU_errno set to indicate the error. The following error values are defined in

EVU_OK - No error

EVU_CANCELLED - Request cancelled by user
: This error indicates that the request was cancelled by the user, for instance, by pressing the *cancel* button of a property sheet. This error will also be the result if the user presses *accept* with no metrics selected.

EVU_ERRORATNODE - Error condition at node
: There is an error condition at the selected node. This error covers such conditions as the exiting of a slave analyst process while the instrumented program was running.

EVU_INVALIDNAME - Invalid node name
: No nodes match the given node name template.

EVU_MASTERNOTRESPOND - Master not responding
: The IPS process is not responding to requests from the EVU. This usually indicates that the master has exited.

EVU_NODATAATNODE - Can not find data for selected node and metric
: The requested metric can not be retrieved from the slave. This usually indicates that a slave analyst process exited after the program finished running.

EVU_NOEDCUHIST - No histograms for EDCU nodes
: A request was made for histogram data from an EDCU node.

EVU_NOHISTOS - No histogram metrics defined for the selected node
: There are no histogram metrics defined for the selected node.

EVU_NOSUCHMETRIC - No such metric
: The given name does not correspond to any known metric.

EVU_NOTDEF - Metric not defined at selected node
: The given metric is not defined at the given node.

EVU_NOTHINGSELECTED - Nothing selected
: No node is currently selected.

EVU_TABLEONLY - Metric is tableonly at node
: Histogram data was requested for a metric which is only available as a table metric at the given node.

EVU_TYPENOTSUPPORTED - Metric type is not supported
: A request was made for a metric type other than

EVU_NOTUNIQUE - Not unique node specification
: A request was made for NodeInfo, but the name passed in did not uniquely specify a node.

EXAMPLES

This section gives some examples of common uses for the functions in the EVU library. The first example shows how to have to user select a single metric, and then print the histogram bins on the standard output:

```
#include <stdio.h>
#include "evu_lib.h"

main(void)
{
    Metric *metric;
    HistoData *hdata;
    int i;

    /*
     * Request IPS to bring up a property sheet and return the
     * selected metric at the selected node
     */
    if((metric = EVU_GetOneMetric(HISTO)) == NULL){
    EVU_perror("EVU_GetOneMetric");
    exit(1);
    }

    /*
     * Output something like: "prog:ips2(CPU Time)"
     *
     * (Note that it is always safe to print a node name at level
     * IPS_NLEVELS, as name->level will always be less than this.)
     */
    fprint_nodename(stdout, metric->name, IPS_NLEVELS);
    printf("(%s)\n", metric->desc->title);

    /*
     * Print the bins
     */
    hdata = &(metric->data->hdata);
    for(i = 0; i < hdata->numbins; ++i)
        printf("%d: %f\n", i, hdata->bins[i]);
}
```

The next example shows how you might retrieve CPU time for every process-level node:

```
#include <stdio.h>
#include "evu_lib.h"

#define METRIC_NAME  "ips_cpu_time"

main(void)
{
    NodeName name;
    MetricDesc *desc;
    Metric *metrics;
    int nmetrics;

    /*
     * Create a template to match all process-level nodes
```

```
     */
    init_nodename(&name, IPS_PROCESS, ANY, ANY, ANY, NULL);

    /*
     * Get the descriptor for CPU Time
     */
    if((desc = EVU_GetDesc(METRIC_NAME)) == NULL){
        EVU_perror("EVU_GetDesc");
        exit(1);
    }

    /*
     * Get the metrics
     */
    if((metrics = EVU_GetThisMetric(desc,
                                    &name,
                                    HISTO,
                                    &nmetrics)) == NULL){
        EVU_perror("EVU_GetThisMetric");
        exit(1);
    }

    ...
}
```

The third example shows how to have the user select a metric which is defined for all of the children of the selected node:

```
#include <stdio.h>
#include "evu_lib.h"

main(void)
{
    NodeName *name;
    Metric *metrics;

    /*
     * Get the name of the currently selected node
     */
    if((name = EVU_GetSelectedNode()) == NULL){
        EVU_perror("EVU_GetSelectedNode");
        exit(1);
    }

    /*
     * Transform the name into a template which matches all of its
     * children
     */
    if(++name -> level == IPS_NLEVELS_){
        fprintf(stderr, "Procedure-level nodes have no children.\n");
        exit(1);
    }
    name -> components[name -> level] = ANY;
```

```
    /*
     * Get the metrics
     */
    if((metrics = EVU_GetThisMetric(NULL,
                                    name,
                                    HISTO,
                                    &nmetrics)) == NULL){
        EVU_perror("EVU_GetMetrics");
        exit(1);
    }

    /*
     * Output something like "Process-level children of prog:ips2"
     */
    printf("%s-level children of ", Ips_levels[name -> level - 1]);
    fprint_nodename(stdout, name, name -> level - 1);
    printf(".\n");

    ...
}
```

NOTES

All structures returned by the EVU library are allocated with malloc(3).

Link load your EVU with

Do not compile your EVU with the ips instrumentation (compiler switch -pi).

BUGS

Only histogram metrics are currently supported. Table metrics and critical path data may be some day.

Currently, the wildcard feature does not provide a way to ask only for visible nodes.

Various attributes of a metric can change after the metric has been retrieved. For example, the title of a metric can change, and even the values can change, if it is a derived metric and the user changes the metric's formula interactively. Currently, there is no way for an EVU to find out about these changes.

There is no way for the EVU to find out that the master has exited (short of requesting a metric and finding the master not there).

Currently, it is possible to retrieve several metrics for a single node (using EVU_GetMetrics), or one metric at several nodes (using EVU_GetThisMetric and a node name with wildcard components). There is no way to ask for lots of metrics at lots of nodes with a single call (one might argue that this is feature, not a bug).

If the user presses the *accept* button on a property sheet without having selected any metrics, it will be handled as if the user had pressed the *cancel* button.

The deep nesting of structures means that it requires a long string of field names to get to some pieces of data (such as histogram bins).

The strings to be passed to EVU_GetDesc are ‘‘magic’’, in that there is no well defined place from which to get them.

The **NodeName** structure is inconsistent with the **HistNode** structure of the EDCU library, which is used for the same purpose.

SEE ALSO

errno(2), malloc(3), perror(3), strerror(3)

NAME

histogram – ips external histogram files.

DESCRIPTION

Histograms are ASCII files produced by IPS external data collector units (EDCUs) and processed by the IPS slave analyst. This format is still under development.

The first line of the file contains the version number of the file as an integer. The second line of the file contains the node specifier for the metric. The node specifier is four space separated strings. Strings that are asterisks denote wild cards. The third line of the file contains the name (identifier) of the metric represented by the histogram. The fourth line file contains the start time, bucket width, and number of buckets of the histogram. The following lines contain the bucket values in order.

The start time is represented as seconds and microseconds since midnight January 1, 1970. The bucket width is represented in seconds and microseconds. The number of buckets in the histogram is represented by an integer. All of the values on the fourth line of the file are separated by commas.

The bucket values are floating point numbers separated by commas.

EXAMPLE

The following example shows a histogram of pageins over 9 seconds starting Thursday, July 26, 1990 at 17:05:19 (330544 microseconds).

```
2
* * * *
pageins
649029919,330544,0,500000,18
0.50,0.50,1.96,2.02,0.06,0.00,0.00,0.00,0.00,0.00,7.51,8.08,6.16,6.06,
4.16,4.04,2.18,2.02
```

SEE ALSO

gettimeofday(2), edcu(3), vmcollect(1)

Index

Table of Contents