Gedae Primitive Function Reference Manual

June 25, 2008

Address:	Gedae, Inc. 1247 N Church St, STE 5 Moorestown, NJ 08057
Telephone:	(856) 231-4458
FAX:	(856) 231-1403
Internet:	www.gedae.com

Contents

Style Guide

In this document, the first use of any **term** appears in boldface font. Code, such as

```
x[i] = a[i]+1;
```

appears in bold Courier font. Built-in functions and Gedae language extensions, such as

```
Apply: {
    int i;
    amount(in,N);
    …
}
```

appear in bold blue Courier font. Gedae Run Time Kernel (RTK) library functions, such as the **embCalloc** call below:

```
float *x = embCalloc(N,sizeof(float));
```

appear in bold green Courier font.

Built-in Functions

The **built-in** functions of the **Gedae Primitive Language** are code-generated into statements that obtain information or modify the state of input and output **identifiers**. These functions always take an input or output identifier as one of their arguments.

Consider the following example:

```
Name: add
Type: static
Input: {
  stream float a;
  stream float b;
}
Output: {
  stream float out;
}
Apply: {
  int N = size(a);
  int i;
  for (i=0; i<N; i++) out[i] = a[i]+b[i];
}
```

The argument **a** is an identifier to the built-in **size** function and its use as identifier should be sharply distinguished from its use as an array of pointers in the **for** loop. When not used as identifiers, input and output variable names, such as **a** and **out,** refer to arrays of values of the base type. But when the variable names are passed as arguments to built-in functions, they allow the function to set the state and retrieve status information from the underlying input and output data ports. In the example above, **a[i]** refers to the ith element of the array **a**, but the built-in function **size(a)** is code-generated into the statement **_state->N_a.**

The Gedae built-in primitive functions are described in the following sections. This page describes the format of each function description.

Synopsis

The Synopsis section presents the calling syntax for the function including the declarations of the arguments and the return type. For example:

```
returntype appFunction(type1 arg1, type2 arg2);
```

The type specifiers can be any standard C data types or the type **`Identifier`**, which refers to an input or output identifier.

Box Types

The Box Types section names the types of primitives that may use this function. The types may include one or more of **`static`**, **`cyclic`**, **`eval`** or **`trigger`**.

Methods

The Methods section lists the methods that may use this function. The method may be one or more of **`Init`**, **`Start`**, **`Reset`**, **`ClassReset`**, **`Apply`**, **`Terminate`**, **`Destroy`**, **`Eval`**, **`Trigger`**, **`Save`**, or **`Restore`**.

Description

The Description section describes what the function does, what it returns and what side effects it causes.

Examples

The Examples section shows simple examples of how the function is used.

Example Primitives

The Example Primitives section lists some representative primitives from the Gedae Primitive Library that use the function.

See Also

The See Also section lists the primitive functions that are related to the function being described.

amount

Synopsis

```
int amount(Identifier id, int tokens);
```

Box Types

static

Methods

Apply

Description

The **amount** function specifies the data flow requirements (**tokens**) on a nondet input or output identifier, **id**. For nondet inputs, **tokens** specify how many tokens must be available. For nondet outputs, **tokens** specifies how much space must be available. If the queue requirements are met, then the **amount** function prepares the identifier **id** to be used and returns a nonzero value. If the queue requirements are not met, then the amount function sets the identifier **id** to zero, returns a value of zero and also sets the **queues_ready** flag to zero. It is the responsibility of the primitive programmer to detect that the **amount** failed by either checking its return value or checking the **queues_ready** flag. Multiple calls to **amount** can be made before the **queues_ready** flag is checked. If one or more amount calls fail, then the Apply method should exit without executing its algorithm. After the Apply method exits the primitive calling, it will be suspended and not called again until the data flow requirements specified by the **amount** calls have been met. If any amount call fails during processing of a segment of data that has an end-of-segment marker, then the failed call causes the primitive to move to the end-of-segment state.

Examples

The first example shows how the **amount** function is used with non-family and family inputs and outputs.

```
Input: {
  nondet stream float in;
  nondet stream float [F]fam_in;
  int N;
}
Output: {
  nondet stream float out;
  nondet stream float [F]fam_out;
}

Apply: {
  int f;
  amount(in,N);
  amount(out,N);
  for (f=0; f<F; f++) {
    amount(fam_in[f],N);
    amount(fam_out[f],N);
  }
  if (queues_ready) {
    /* only progress if all amount calls succeeded */
     .. can reference
      in[0]..in[N-1]
      out[0]..out[N-1]
      fam_in[0][0]..fam_in[F-1][N-1]
      fam_out[0][0]..fam_out[F-1][N-1]
  }
}
```

The simplest primitive that uses **amount** is the ndet_copy, which copies its nondet input to its static output.

```
Name: ndet_copy
Type: static
Input: {
  nondet stream float in;
}
Output: {
  stream float out;
}
Apply: {
  if (amount(in,granularity)) {
    e_vmov(in,1,out,1,granularity); /* vector move */
    consume(in,granularity);
  }
}
```

In this example, because only one **nondet** queue is involved, the value returned by **amount** can be checked directly using an **if** statement. When more than one amount function is called, it is more convenient to check that they are all successful using **if(queues_ready).**

The **amount** function will always succeed if it is called with **avail(id)** as the number of tokens. If in the above example instead of calling **amount(in,N)** we call **amount(in,avail(in))**, then the **amount** function is guaranteed to succeed.

Note that it is always necessary to call the **amount** function as it serves both the purpose of checking that the data flow requirements are met and preparing the identifier to be accessed. The following code would cause a segfault:

```
int ain = avail(in);
if (ain) {
  /* segfaults even though there are tokens on in */
  float x = in[0];
}
```

The following code, which uses the **amount** function before the variable **in** is dereferenced, corrects the problem.

```
int ain = avail(in);
if (ain) {
  float x,y;
  amount(in,ain); /* prepare in to be dereferenced */
  x = in[0];
  y = in[ain-1];
}
```

Note that there is no need to check the **queues_ready** flag in the above case as the **amount** function will not fail if it is passed **avail(in).**

An example of a complete function that uses the amount is seen below:

```
Name: latch
Type: static
Input: {
  nondet stream float in;
}
Local: {
  int ready;
  float last_in;
}
Output: {
```

```
  stream float out;
}
Include: {
#include <e_vfill.h>
}
Reset: {
  ready = 0;
}
Apply: {
  int ain = avail(in);
  if (ain) {
    amount(in,ain);
    last_in = in[ain-1];
    consume(in,ain);
    ready = 1;
  }
  if (ready) {
    int i;
    e_vfill(last_in,out,1,granularity);
  } else {
    amount(in,1);
  }
}
```

Example Primitives

embeddable/stream/logic/merge_c shows an example of how a control stream can determine the amount that must be available on two nondet input streams.

embeddable/stream/logic/umerge shows how an uncontrolled merge can be implemented.

embeddable/stream/distribute shows how a nondet output stream can be used to do dynamic load balancing.

See Also

queues_ready, avail

avail

Synopsis

```
int avail(Identifier id);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The function **avail** has two different meanings depending on whether the identifier is a nondet input or a nondet output dynamic data stream. For input data streams, the **avail** function returns the number of available input tokens on the input queue. For output data streams, the **avail** function returns the number of tokens that may be written to the output stream.

Note that **amount** must be called on the **nondet** streams if the primitive needs to access the data in the streams. If the return value from avail is passed to the amount function, then the amount function is guaranteed to succeed.

Examples

The following example shows how the **avail** function is used with nonfamily and family inputs and outputs.

```
Input: {
  nondet stream float in;
  nondet stream float [F]fam_in;
}
Local: {
  int fain[F];
  int faout[F];
}
Output: {
  nondet stream float out;
```

```
  nondet stream float [F]fam_out;
}

Apply: {
   int i;
   int ain = avail(in);   /* number of tokens on in */
   int aout = avail(out); /* space available on out */
   for (i=0; i<F; i++) {
      /* get number of tokens on each family input */
      fain[i] = avail(fam_in[i]);
      /* get available space on each family output */
      faout[i] = avail(fam_out[i]);
  }
  /* Process data based on availability of tokens
     on inputs and outputs */
  ...
}
```

Example Primitives

embeddable/stream/logic/umerge shows how an uncontrolled merge can be implemented by checking availability on input queues.

embeddable/stream/distribute shows how a nondet output stream can be used to do dynamic load balancing by checking availability on output queues.

See Also

amount

consume

Synopsis

```
void consume(Identifier id, int tokens);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The function **consume** notes how many tokens are to be consumed from a dynamic input stream identifier, **id**. For any identifier, **consume** should be called just once at the end of the function. The algorithm must keep a running total of the number of tokens to be consumed and pass this number to the **consume** function in the **tokens** variable. The identifier passed to the **consume** function must be that of a dynamic or nondet input stream. The **consume** function should not consume more tokens than are available in the input stream.

Examples

For a dynamic input stream with a decimate rate set to **D** declared as:

```
dynamic stream in(D);
```

or

```
dynamic stream in; /* D is 1 by default */
```

the call to **consume** should not consume more than **granularity*D** tokens.

For a nondet input stream that has successfully met the requirement of an **amount** function as:

```
amount(in,N);
```

the **consume** function should consume no more than **N** tokens.

Example Primitives

embeddable/stream/logic/umerge shows how an uncontrolled merge can be implemented by consuming only tokens available on the inputs.

embeddable/stream/logic/mergef_c shows how the number of tokens consumed can be determined from a control input stream.

See Also

avail, amount, produce

decode

Synopsis

```
void* decode(Identifier in, int *runlength, int
max_runlength);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **decode** function returns successive run-length/value pairs of the input encoded queue **in**. The value is returned as the return value of **decode** and the run-length is returned in the **runlength** parameter. The returned value of the **runlength** value never exceeds the value of **max_runlength**. On completion of each call to **decode**, **runlength** tokens are consumed from the encoded queue. If the encoded queue is empty, then the **decode** function returns 0 and sets the **runlength** to 0.

Examples

A data stream containing the values

1 1 1 1 2 2 2 3 3 3 3 3 4 4

can be encoded as run-length/value pairs as:

(4,1) (3,2) (5,3) (2,4).

given that a stream declared as

```
encoded stream int in;
```

has run-length/value pairs as described above. The code sequence that follows describes the effect of successive calls to the **decode** function.

```
int *x;
```

```
x = decode(in,&runlength,4);
/* at this point *x = 1, runlength = 4 and the queue is in state ((3,2) (5,3) (2,4) */
x = decode(in,&runlength,4);
/* at this point *x = 2, runlength = 3 and the queue is in state (5,3) (2,4) */
x = decode(in,&runlength,4);
/* at this point *x = 3, runlength = 4 and the queue is in state (1,3) (2,4) */
/* only 4 tokens were consumed from the runlength encoded value (5,3) */
/* leaving the value (1,3) at the head of the queue */
x = decode(in,&runlength,4);
/* at this point *x = 3, runlength = 1 and the queue is in state (2,4) */
x = decode(in,&runlength,4);
/* at this point *x = 4, runlength = 2 and the queue is empty */
x = decode(in,&runlength,4)
/* at this point x = 0 and runlength = 0 indicating the queue is empty */
```

Example Primitives

embeddable/stream/decode shows how an encoded stream can be converted to a normal stream.

See Also

decode, encode, peek

dirty

Synopsis

```
int dirty(Identifier id)
```

Box Types

```
trigger
```

Methods

```
Trigger
```

Description

The **dirty** function allows a trigger box to determine which trigger inputs were pushed from upstream primitives, which in turn, causes the trigger box to execute. The **dirty** function returns 1 if the identifier was pushed since the last call to the trigger box. Otherwise, it returns 0.

Examples

Given a trigger box with inputs:

```
trigger int in1;
trigger int in2;
```

The trigger method can be written as:

```
Trigger: {
  if (dirty(in1)) {
    /* handle in1 */
  }
  if (dirty(in2)) {
    /* handle in2 */
  }
|
```

Example Primitives

discrete/integer/I_Sync2 – shows a trigger box that copies the two inputs to the two respective outputs only after both of the inputs have been pushed (made dirty) by the upstream primitives.

See Also

push

drain

Synopsis

```
void drain(Identifier id);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **drain** function, when called on a **nondet stream** input identifier **id,** declares that any tokens currently in the input queue or any new queues arriving on the queue should be discarded until the beginning of the next segment of data. The **drain** function allows primitives with **nondet** inputs to move to the end-of-segment state, while tokens are still available on the input queue and before an end-of-segment has been issued on the nondet input.

The **drain** function should not be called if the primitive has made any progress, that is, produced data on dynamic outputs or filled in static outputs. The call to **drain** should be postponed until the next execution of the primitives **Apply** method.

Examples

Given that an input stream is defined as:

```
nondet stream float in;
```

then a primitive that determines that it doesn't need to process any more data from a segment can end the segment processing by calling:

```
drain(in);
```

Example Primitives

`test/segmentation/database` – shows a database primitive that drains its input queue (using the draining function) as soon as it detects that an end-of-segment marker has been issued on its input stream.

See Also

`draining`

draining

Synopsis

```
int draining(Identifier id)
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **draining** function returns the value 1 if an end-of-segment marker is on the **nondet input stream** identifier **id**. Otherwise, it returns a 0.

Examples

Given an input stream declared as:

```
nondet stream float in;
```

a primitive can move to the end-of-segment state when an end-of-segment marker is placed on the input queue by calling

```
if (draining(in)) {
   drain(in);
}
```

This code is primarily used to throw away tokens and is sometimes necessary when an application has a variable amount of data and needs to meet latency and throughput requirements.

Example Primitives

test/segmentation/database – shows a database primitive that drains its input queue as soon as it detects that an end-of-segment marker has been issued on its input stream.

See Also

drain

encode

Synopsis

encode(Identifier out, void *value, int runlength)

Box Types

static

Methods

Apply, EndOfSegment

Description

The **encode** function places run-length/value pairs on an encoded stream. The encoded stream is specified by the identifier **out.** The value and run-length are specified by the **value** and **runlength** parameters.

Examples

Example Primitives

See Also

set_ptr, produce_ptr

mem_type

Synopsis

```
char *mem_type(Identifier id)
```

Box Types

static

Methods

Start, Reset, Apply, Destroy, Terminate, EndOfSegment

Description

The **mem_type** function returns a character string that declares the type of memory of the input or output identifier **id**. The return value is a string that is either "default" or the value of a named memory type set by the user from the Gedae Development Environment. The identifier memory is set in the Development Environment by first partitioning the memory using the Group Setting Dialog's Partition Memory Dialog, and setting the partition to run in the named memory block using the Set Schedule Parameter Dialog.

The memory type name returned can then be passed to the **embCalloc**, **embFree** or the **embSbAlloc** functions. Passing the memory type to these functions gives the Development Environment user the ability to control the memory type that a primitive uses to do runtime memory allocation.

Examples

See Examples section of **embSbAlloc.**

See Also

embSbAlloc, embCalloc, embFree, n_dests

n_dests

Synopsis

```
int n_dests(Identifier id);
```

Box Types

static

Methods

Start, Reset, Apply, Destroy, Terminate

Description

The function **n_dests** returns the number of destinations to which the output identifier **id** is connected.

Examples

See Examples sections for the **embSbAlloc, embSbForward and embSbCopy** functions.

See Also

mem_type, embSbAlloc, embSbForward, embSbCopy

peek

Synopsis

```
void* peek(Identifier in, int *runlength, int index);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **peek** function allows a primitive to examine the runlength/value pairs on the encoded stream **in** without consuming them. The number of the runlength/value pair to examine is passed in the **index** parameter. The **index** parameter is zero based, that is, **index == 0** retrieves the first runlength/value pair from the queue. The runlength is returned in the **runlength** parameter, and a pointer to the value is returned as the functions return value.

Examples

A data stream containing the values

1 1 1 1 2 2 2 3 3 3 3 3 4 4

can be encoded as the run-length/value pairs:

(4,1) (3,2) (5,3) (2,4).

Given that an input stream declared as

```
encoded stream int in;
```

and has run-length/value pairs as described above. The code sequence that follows describes the effect of successive calls to the **peek** function.

```
int *x;

x = peek(in,&runlength,2);
/* at this point *x = 3 runlength = 5 */
x = peek(in,&runlength,0);
```

```
/* at this point *x = 1 and runlength = 4 */
```

Example Primitives

`embeddable/stream/logic/emerge` – shows how the encoded control queue for the merge box can be examined using **`peek`** before the queue is actually used and consumed by **`decode`**.

See Also

`decode, encode`

post

Synopsis

```
post(Identifier id);
```

Box Types

static

Methods

Apply

Description

The **post** function causes any change in the value of output parameter identifier **id** to be propagated to the destination parameters. The **post** function returns immediately before the parameter propagation is completed. Unlike the **push** function, the **post** function cannot be called from trigger primitives.

Examples

```
Output: {
  int Out;
}

Apply: {
  Out = 5;
  push(Out);
}
```

See Also

push

produce

Synopsis

```
void produce(Identifier id, int tokens);
```

Box Types

`static`

Methods

`Apply`

Description

The function **produce** notes how many tokens have been produced on a dynamic output stream identifier, **id**. For any identifier, **produce** should be called just once at the end of the function. The algorithm must keep a running total of the number of tokens to be produced and pass this number to the **produce** function in the **tokens** variable. The identifier passed to the **produce** function must be from a dynamic or nondet output stream. The **produce** function should not produce more tokens than there is space for on the output stream.

Examples

For a dynamic output stream with an interpolate rate of **I** declared as:

```
dynamic stream out(I);
```

or

```
dynamic stream out; /* I is 1 by default */
```

The call to **produce** should not produce more than **granularity*I** tokens.

For a nondet output stream that has successfully met the requirement of an **amount** function as:

```
amount(out,N);
```

the **produce** function should produce no more than **N** tokens.

Example Primitives

embeddable/stream/comm/oqpsk_mod shows how the amount of data to be produced can be calculated in a data dependent fashion.

embeddable/stream/logic/branchf_c shows how the number of tokens produced can be determined from a control input stream.

See Also

avail, amount, produce

produce_ptr

Synopsis

```
produce_ptr(Identifier id, void *ptr, int N,
            void (*release)(void *, void *),
            void *handle
```

Box Types

static

Methods

Apply

Description

The **produce_ptr** function sets the buffer of a dynamic or nondet output pointer stream **id** to be equal to **ptr** and notes that there are **N** tokens available in the buffer. When all of the downstream primitives are done using the pointer (as denoted by all the destinations having consumed all **N** tokens from the pointer stream), then the **release** function is called. The **release** function is passed two parameters: the pointer, **ptr** that was passed to **produce_ptr** as its second parameter, and the **handle** passed as the fifth parameter.

Calling **produce_ptr(out,ptr,N,release,handle)** on a pointer stream is logically equivalent to calling the following on a non-pointer stream:

```
memcpy(out,ptr,N*sizeof(*out)); /* copy data from
pointer to output */
produce(out,N);
release(out,handle);
```

The advantage of using **produce_ptr** is that it does not require copying data to the output stream. The call to release is postponed until the output stream is no longer needed by the downstream primitives.

Examples

Below is a code fragment that shows how **produce_ptr** can be used with an input device that can – when queried – return a pointer of varying length. A handle to the input device is opened in the **Start** method. The **Apply** method gets pointers from the input device and calls **produce_ptr** to pass that pointer to the output. The **produce_ptr** parameter **release** is a function that notes

how the pointer is to be returned to the input device when it is no longer needed. The **release** function will be passed both the **ptr** (the second parameter passed to **produce_ptr**) and the device **handle** (the fifth parameter passed to **produce_ptr**). The code fragment is based on a hypothetical input device that provides the function **getPtrFromHandle** that returns a pointer to values that are filled in by the device driver and also returns the number of values in the pointer that have been set.

```
Local: {
  void *handle; /* handle to an input device */
}
Output: {
  pointer stream float out(Nout);;
}

Include: {
  release(void *ptr, void *handle) {
     .. inform the device described by handle that
     the ptr is now free to write new data to ..
  }
}

Start: {
  handle = .. create handle to input device ..
}

Apply: {
  int Nvalues;
  float *ptr = getPtrFromHandle(handle,&Nvalues);
  if (Nvalues > 0) {
    .. a ptr to a vector of floats Nvalues in length
    was read from the input device...
    produce_ptr(out,ptr,Nvalues,release,handle);
  } else {
    embSuspendRetry("input device not ready");
  }
}
```

See Also

set_ptr, produce

push

Synopsis

```
push(Identifier id1, Identifier id2,…,Identifier idN);
```

Box Types

trigger
static

Methods

Trigger
Apply

Description

The **push** function simultaneously notes that the values of output parameter identifiers **id1, id2,** … **idN** have been changed and causes the effects of those changes to be propagated to the downstream primitives and derived parameter values. The **push** function does not return until its effects have completely propagated throughout the graph. All of the downstream primitives and derived parameter values affected by the **push** must be updated before the **push** returns. Since the **push** function doesn't return until the downstream graph has completely executed, the **push** function calls the downstream graph as though it were a function.

The **push** function can be called by trigger primitive **Trigger** methods or static primitive **Apply** methods. When called from an **Apply** method, the push function can currently take only one parameter value.

Examples

The following trigger primitive copies the input to the two outputs and pushes the outputs sequentially. All of the effects of **push(out0)** will occur before any of the effects of **push(out1)**.

```
Name: I_Seq2
Type: trigger
Input: {
  trigger int in ;
}
Output: {
  int out0 ;
```

```
  int out1 ;
}
Trigger: {
  out0 = in ;
  push(out0) ;
  out1 = in ;
  push(out1) ;
}
```

A static primitive example is given below. The push function called from a static primitive **Apply** method gives the **Apply** method the opportunity to convert streams to parameters. In this primitive, every floating-point input is pushed to the output parameter **Out**. All the effects of **push** must complete before **push** returns.

```
Name: push
Type: static

Input: {
  stream float in;
}

Output: {
  float Out;
}

Apply: {
  int i;
  for (i=0; i<granularity; i++) {
    Out = in[i];
    push(Out);
  }
}
```

Example Primitives

discrete/integer/I_Seq2 – a primitive that uses the push function to update its outputs in sequence.
discrete/stream/v_VA – a primitive that converts a stream of vectors into a pointer to G vectors. This primitive is used to allow vector streams to be sent to GUIs built out of trigger boxes.

See Also

post

segment

Synopsis

```
segment(Identifier id, SegmentType type);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The need often arises to process finite length data streams to completion. At the beginning of the stream, the processing must be reset and summary information often needs to be produced at the end of the stream. The **segmented data flow** capability in Gedae addresses the problem of directly processing finite streams and also offers opportunities to achieve efficient parallel execution of algorithms.

The **segment** function call puts a segment marker on the **segmented dynamic** or **segmented nondet** output stream specified by the identifier **id**. The segment marker can be either **SEGMENT_BEGIN** (begin a new segment) or **SEGMENT_END** (end the current segment).

Examples

Segmentation control begins with a box that produces segments on its output. Outputs of static box primitives can be marked as segment outputs using the keyword `segmented`. See the example below.

```
Output: {
  segmented dynamic stream float out;
}
```

All segmented outputs must be either **dynamic** or **nondet**. The beginning and end-of-segment are marked on the stream using the built-in **segment** function:

```
segment(out,SEGMENT_BEGIN); /* begin a new segment */
```

and

```
segment(out,SEGMENT_END); /* end current segment */
```

Any **produce** statements between the beginning and the end-of-segment put data into the current segment. If a **produce** statement occurs after a **SEGMENT_END** and before a **SEGMENT_BEGIN** is issued, then an implicit segment-begin is issued. As a result, issuing a segment-begin is not necessary.

An example box that produces segments is given below. The box drops the first **Where** tokens from its input data stream. After this, it cyclically begins a new segment, copies **Ton** tokens from the input to the output, ends the segment and drops the next **Toff** tokens from the input.

```
Name: segmenter
Type: static
Input: {
  stream float in;
  int Ton;
  int Toff;
  int Where;
}
Local: {
  int where;
}
Output: {
  stream segmented dynamic float out;
}
Reset: {
  where = -Where;
}
Apply: {
  int g;
  int N = 0;
  int k = 0;
  for (g=0; g<granularity; g++) {
    if (0 <= where && where < Ton) {
      out[k++] = in[g];
      N++;
    }
    where++;
    if (where == Ton) {
      produce(out,N);
      N = 0;
      segment(out,SEGMENT_END);
    }
    if (where == Ton+Toff) {
      where = 0;
    }
  }
  produce(out,N);
```

}

A segment is processed by the primitive or subgraph immediately following a segmented output. Therefore, the scope of the segmentation is the single flow graph or primitive immediately attached to the segmenter output. The subgraph calls its **`Reset`** methods at the beginning of segment execution and its **`Destroy`** and **`EndOfSegment`** methods at the end-of-segment execution. The **`granularity`** of the subgraph will be set to consume as much data as possible up to the end of the segment. In the example, graph and subgraph shown in the two figures below, the **`segproc3`** box is within the scope of the **`segmenter`** box output.

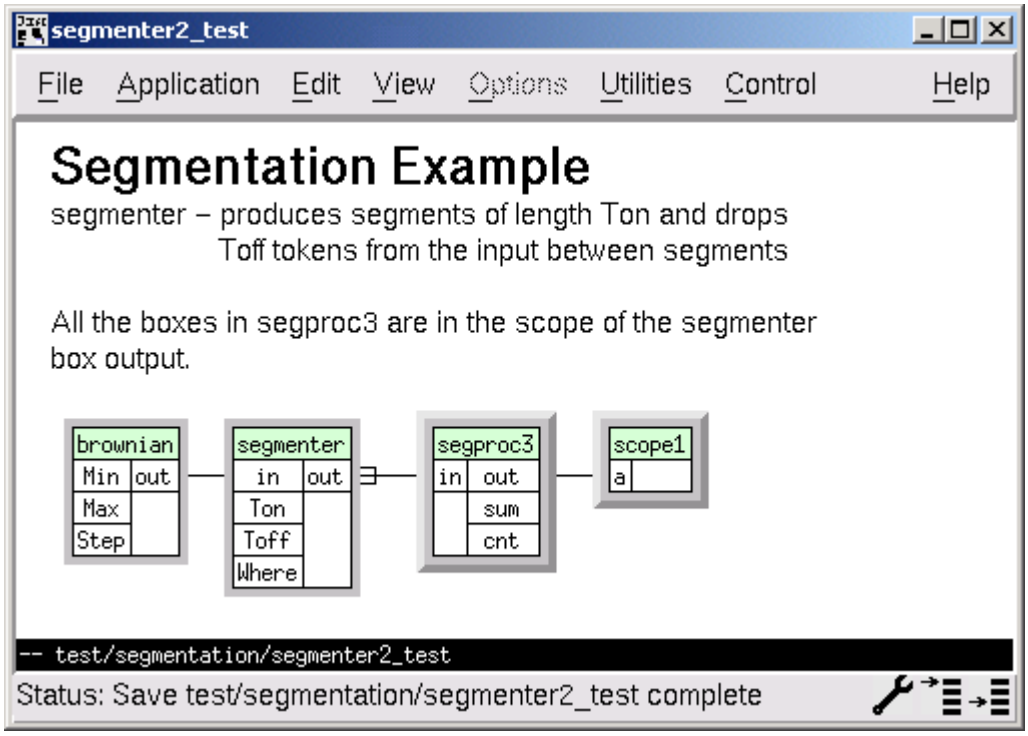


Figure 1Top-level segmentation control graph

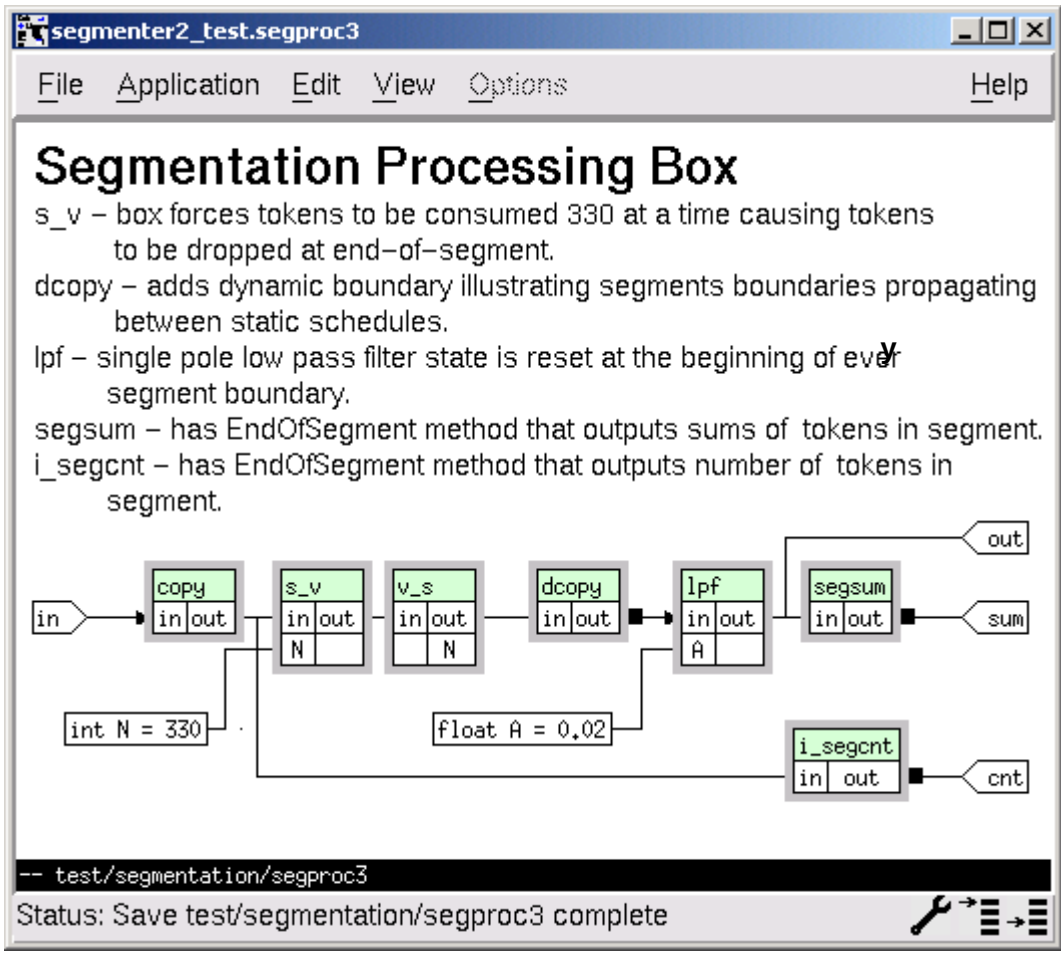


Figure 2 - Graph controlled by segmenter output

See Also

produce

set

Synopsis

```
set(Identifier id, void *value, int *nelems);
```

Box Types

trigger
eval

Methods

Init

Description

The **set** function allows the default value of a vector, matrix or higher level dimensional parameter input to be initialized in the primitives **Init** method. The id argument is the identifier to be set. The value argument is the values of the array and should be a flat array of size in bytes of:

nelems[0]*nelems[1]*…*nelems[ndims-1]*sizeof(Type)

In the above expression, **ndims** is the number of dimensions of the parameter and must be greater than or equal to 1. **Type** is the type of the parameter value to be set.

Examples

The following code initializes **Name** to a default value of **"hello world"** and **V** to a two-dimensional array of size 2x4 with values **{1,2,3,4}** in the first row and values **{11,12,13,14}** in the second row.

```
Input: {
  char Name[N];
  float V[P][Q];
}

Init: {
  char *str = "hello world";
  float x[] = {1,2,3,4,11,12,13,14};
  int len = strlen(str);
  int nelems[2] = {2,4}

  set(Name,str,&len);
```

```
    set(V,x,nelems);
}
```

Example Primitives

discrete/string/A_ResetK sets input strings to a default value of "" – the null string.

See Also

set_ptr

Synopsis

```
set_ptr(Identifier id, void *ptr,
        void (*release)(void *, void *),
        void *handle);
```

Box Types

static

Methods

Apply

Description

The **set_ptr** function sets the buffer of an output pointer stream **id** to be equal to **ptr**. When all of the downstream primitives are finished using the pointer, which means that all the destination primitives have executed to completion, then the **release** function is called. The **release** function is passed two parameters: the pointer, **ptr** that was passed to **produce_ptr** as its second parameter, and the **handle** passed as the fifth parameter.

Calling **set_ptr(out,ptr, release,handle)** on a pointer stream is logically equivalent to calling the following on a non-pointer stream.

```
memcpy(out,ptr,N*sizeof(*out));
release(out,handle);
```

The advantage of using **set_ptr** is that it does not require copying data to the output stream. The call to release is postponed until the downstream primitives no longer need the output stream.

Example Primitives

embeddable/stream/audio/audioIn_nt – this primitive shows how a pointer stream can be used to allow destination primitives to directly process data out of an audio device output buffer without requiring the data to be copied out of the buffer.

See Also

produce_ptr

size

Synopsis

```
int size(Identifier id);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **size** function returns the number of data elements of the base type available in a static input or output stream identifier. The number of data elements is the granularity of the box times the product of the stream's dimension values times the decimate or interpolate rate of the identifier.

Examples

Given streams defined as:

```
Input: {
  stream int in[R][C](D);
  int D;
  int I;
}

Output: {
  stream int out[R][C](I);
}
```

then in an **Apply** method where the built-in variable **granularity** is already defined, the statements:

```
int n_in = size(in);
int n_out = size(out);
```

are equivalent to:

```
int n_in = granularity*R*C*D;
int n_out = granularity*R*C*I;
```

Example Primitives

Using the **size** function – the following three primitives have identical **Apply** methods:

```
embeddable/vector/add - size(in)==granularity
embeddable/vector/v_add – size(in)==granularity*n;
embeddable/matrix/m_add – size(in)==granularity*n*m;
```

See Also

granularity

src_name

Synopsis

```
char *src_name(Identifier id);
```

Box Types

trigger
eval

Methods

Trigger
Eval

Description

The function **src_name** returns the hierarchical graph name of the source of the input identifier. If the identifier is unconnected, then **src_name** returns the string **"<no-source>"**.

Examples

```
Name: SrcName
Type: eval
Input: {
  int in;
}
Output: {
  char Out[200];
}
Include: {
#include <strings.h>
}
Eval: {
  strcpy(Out,src_name(in));
}
```

time

Synopsis

```
double time(Identifier id);
```

Box Types

trigger

Methods

Trigger

Description

The **time** function returns the time in milliseconds when the trigger input identifier **id** was set. The time value returned is useful for measuring elapse times since the last call but has no absolute interpretation.

Examples

The primitive discrete/float/Time shown below outputs the time in seconds that the input **in** was last updated.

```
Name: Time
Type: trigger
Input: {
  trigger int in;
}

Output: {
  float out;
}

Trigger: {
  out = time(in)*0.001;
  push(out);
}
```

See Also

dirty

tokensize

Synopsis

```
int tokensize(Identifier id);
```

Box Types

```
static
```

Methods

```
Start
Reset
Apply
Destroy
Terminate
EndOfSegment
Cycle
```

Description

The `tokensize` function returns the product of a stream's dimension values.

Examples

Given streams defined as:

```
Input: {
  stream int in[R][C](D);
  int D;
  int I;
}

Output: {
  stream int out[R][C](I);
}
```

Then:

```
int n_in = tokensize(in);
int n_out = tokensize(out);
```

are equivalent to:

```
int n_in = R*C;
```

```
int n_out = R*C;
```

To get the number of bytes in a token, multiply **tokensize(in)** by **sizeof(<type>)** where **<type>** is the data type of the input.

See Also

size, granularity

Built-in Variables

The built-in variables provide basic values that may be used in primitives. While they appear as variables in the Gedae Primitive Language, they get code-generated into more complicated expressions in the C code that implements the primitive.

firing

Synopsis

```
int firing;
```

Box Types

cyclic

Methods

Cycle

Description

The **firing** variable is used by the **Cycle** method of a cyclic box. Cyclic boxes exhibit predetermined data flow that changes in a cyclic fashion. The length of the cycle is specified in the **Length** section of the box. The **Cycle** method should calculate the total number of tokens consumed and produced on each static input and output after **firing** firings of the primitive have occurred. On the zeroth firing (**firing == 0**), the **Cycle** method should always set the consume and produce amounts to 0. On the last firing of a cycle (**firing == Length**), the **Cycle** method should set the consume and produce amounts to the full amount of data consumed and produced by the primitive in one complete cycle.

Examples

An example of a primitive that can be implemented as a cyclic box is a multiplexer. The function of a multiplexer is to interleave its input streams onto its output stream. A firing of a non-cyclic N input multiplexer requires one token on each of the N inputs. These N tokens from the separate input streams are then combined (multiplexed) into a stream of N tokens on the output. A non-cyclic implementation of a mux2 box (2 input multiplexer) is:

```
Name: mux2
Type: static

Input: {
  stream float a;
  stream float b;
}

Output: {
  stream float out(2);
```

```
}

Apply: {
  for (g = 0; g<granularity; g++) {
    out[2*g]   = a[g];
    out[2*g+1] = b[g];
  }
}
```

This implementation has the disadvantage that the output cannot be produced until both the **a** and **b** inputs have arrived. The latency of the primitive may be greater than necessary because the primitive must wait for both inputs to arrive. This problem is solved by implementing the multiplexer as a cyclic box as given below:

```
Name: mux2
Type: cyclic
Input: {
  stream float a(NA);
  stream float b(NB);
}
Local: {
  int even;
  int NA;
  int NB;
}
Output: {
  stream float out(NA+NB);
}
Include: {
#include <e_vmov.h>
}
Length: {
  return 2;
}
Cycle: {
  NA = (firing+1)/2;
  NB = firing/2;
}
Reset: {
  even = 1;
}
Apply:
  for (g = 0; g<granularity; g++) {
    if (even) {
      out[g] = *a++;
    } else {
```

```
        out[g] = *b++;
      }
      even = !even;
    }
  }
```

In the above example, the cyclic box **Cycle** method states how many tokens are consumed on the a and b inputs by a particular firing. The values of **NA** (the tokens consumed on a, **NB** (the tokens consumed on b) and **NA+NB** (the tokens produced on the output) are shown in the table below.

firing	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NA	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
NB	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
NA+NB	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

In the above table, every firing of the mux2 box copies a new value to the output (the number of tokens produced on the output increases by 1 on each firing of the box) but consumes data from only one of the inputs. The cyclic version of the mux2 needs to fire two times for every one firing of the non-cyclic version. On every even cycle, the cyclic version has moved **firing/2** tokens from each input to produce **firing** tokens on the output.

The following example shows how a cyclic multiplexer with a family of inputs can be implemented. In the multiplexer below, a family of **F** inputs is combined elementwise and sequentially placed on the output. For a noncyclic version of the box, one firing sets **out(i) = [i]in**. for **i=0**…**F**. For a cyclic version of the box, after firing **firing** times **(firing+M-i-1)/M** elements are copied from input **i** to the output. For example, if **M** is 3, then the following table shows how many elements are copied to each output after a total of **firing** firings of the box have occurred.

firing	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D[0]	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5
D[1]	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5
D[2]	0	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5
Sum(D[i])	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

So for any particular input, a token is consumed off an input every 3 firings of the primitives. The consumes from the inputs are offset from each other – that is, on the zeroth input tokens are consumed on firing 1, 4, 7,10 and 13 while the consumes on the first input are offset from the zeroth by 1 (consumes on firings 2, 5, 8, 11, and 14) and the consumes on the second input are offset from the zeroth input by 2.

```
Name: mux
Type: cyclic
Input: {
  stream float [i:F]in(D[i]);
}
Local: {
  int D[F];
  int where; /* the input from which the next */
             /* execution of the box begins to */
             /* copy data */
}
Output: {
  stream float out(sum(D,F));
}
Include: {
#include <e_vmov.h>
}
Length: {
  return F;
}
Cycle: {
  int i;
  for (i=0; i<F; i++) {
     D[i] = (firing+F-i-1)/F;
  }
}
Reset: {
  where = 0;
}
Apply: {
  int j;
  for (j=0; j<F; j++) {/* output offset by j */
    int f = where + j; /*family member to copy out */
    int len =(granularity+F-j-1)/F;
    if (f >=F) f-=F;
    e_vmov(in[f],1,out+j,F,len);
  }
  /* advance input by the number of firings */
  /* (granularity) mod F */
  where = (where+granularity)%F;
}
```

Example Primitives

embeddable/stream/mux – cyclic multiplexer
embeddable/stream/demux – cyclic demultiplexer

See Also

granularity

granularity

Synopsis

```
int granularity;
```

Box Types

```
static
```

Methods

```
Apply
```

Description

A **firing** of a primitive is the basic atomic data flow operation of a primitive. For fully static data flow primitives, each firing requires a fixed number of tokens on each input and produces a fixed number of tokens on each output. To fire, most primitives require one token on each input and produce one token on each output. An example of such a primitive is the embeddable/stream/add. Other primitives may require multiple tokens on some inputs and/or multiple tokens on some outputs. An example of such a primitive is the embeddable/stream/decimate that requires D tokens on its input (where D is a parameter of the decimate primitive) and produces 1 token on its output.

The built-in variable **granularity** indicates how many firings of a primitive one execution of the **Apply** method should implement. Any static or dynamic input streams will be scaled in size to provide enough data to allow the box to fire **granularity** times in one execution. Any static or dynamic output streams will provide enough output buffer space to allow the box to fire granularity times in one execution. The **granularity** keyword is available only in the Apply method and can vary from one execution of the Apply method to the next.

Examples

The following example uses two scalar stream inputs to select one element out of a matrix. The for-loop running from **g = 0..granularity-1** is typical of many primitives using the **granularity** keyword.

```
Name: m_sel_s
Type: static
Input: {
  stream float in[R][C];
  stream int r;
  stream int c;
```

```
 }
 Output: {
   stream float out;
 }
 Apply: {
   int g;

   for (g=0; g<granularity; g++) {
     *out++ = in[(*r++)*C+(*c++)];
     in += R*C; /* advance to next input token */
   }
}
```

Most primitives should either use **granularity** or **size** (which is proportional to **granularity**) so that the primitive fires enough times to insure that all the input tokens are processed; however, there are some cases when a custom primitive must execute at a granularity of 1. If that is the case, then the primitive **Apply** method should verify that the granularity is 1 and should call **embTerminateError** if it is not. For example:

```
 Apply: {
   if (granularity != 1) {
     embTerminateError("expected granularity of 1");
   } else {
     .. do normal operation of primitive ..
   }
 }
```

Example Primitives

embeddable/stream/repeat – repeats the first **R** tokens, which are recorded in a local variable. The **granularity** is unrelated to the value of **R**. For example **R** could be 501 and **granularity** 237. In this case, the box will record 237 tokens during the first firing, 237 during the second and 27 tokens during the third. It will start playing back the repeat buffer during the third firing after the last 27 tokens are recorded. The primitive is coded to perform the algorithm of repeating the first 501 input tokens independent of the primitive's execution **granularity**.

See Also

size

queues_ready

Synopsis

```
int queues_ready;
```

Box Types

static

Methods

Apply

Description

At the beginning of an Apply method's execution, the **queues_ready** flag is set to 1; however, if any calls to the **amount** function fail, then the **queues_ready** is set to 0. If the **queues_ready** flag was set to 0, then this is an indication that the primitive did not fire and will be put in the **queue-wait** state to wait for the amount of data requested by the calls to the **amount** function.

Examples

See the example in the section describing the **amount** function.

Example Primitives

embeddable/stream/logic/merge_c shows an example of how a control stream can determine the amount that must be available on two nondet input streams.

See Also

amount, avail

self_name

Synopsis

```
char *self_name;
```

Box Types

trigger
eval

Methods

Init
Reset
Destroy
Trigger
Eval

Description

The function **self_name** returns a string that is the hierarchical graph name of this instance of the primitive. The **self_name** does not include the top-level graph name.

Examples

The following code shows how the **Title** string of the **Shell** primitive is initialized to have a default value of the hierarchical graph name of this instance of the **Shell** primitive.

```
Name: Shell
Type: trigger
Input: {
  char Title[N];
}

Init: {
  int len = strlen(self_name)+1;
  set(Title,self_name,&len);
  shell = 0;
}
```

Example Primitives

`widget/Xm/Shell` – shows how the **`Shell`** primitive uses **`self_name`** to set the default **`Title`** to be displayed in the **`Shell`** banner.

See Also

`embBoxName`

Trigger Primitive Save/Restore Functions

Trigger primitives can have **Save** and **Restore** methods that are called when a Gedae user saves and restores parameter values from Gedae graph parameter files. The **Save** method saves the state of the primitive to the parameter file, and the **Restore** method restores the primitives state from the parameter file. For example, a Trigger primitive that implements a text entry window might have a **Save** method that saves the value of the text entered by a user and a **Restore** method that restores the value of the text. A total of eight functions have been provided to allow Trigger primitives to save and restore any information required. These functions are:

```
save_int(int);
save_float(float);
save_double(double);
save_string(char *);

restore_int(int *);
restore_float(float *);
restore_double(double *);
restore_string(char **);
```

Each function is described in the subsequent sections, although only **restore_int** is illustrated with a complete example.

restore_int

Synopsis

```
restore_int(int *value);
```

Description

When a parameter file is loaded and a trigger primitive's **Restore** method is called, the **restore_int** function restores the integer parameter **value** saved to the parameter file by the **save_int** function.

Example

This example shows how the save/restore functions are used. No other examples will be given by the other save/restore function descriptions.

The save functions may be called from a trigger box **Save** method, while the restore functions may be called from a trigger box **Restore** method. The save and restore functions should be called in the same order so that everything saved to the parameter file by the **Save** method is restored by the **Restore** method.

The following code saves the size and location of a text display in the **Save** method and creates, sizes and positions the text display in the **Restore** method.

```
Name: text_display
Type: trigger
Input: {
  trigger char in[N];
}
Local: {
  TextDisplay td;
}
Include: {
...
#include <td.h>
}
Reset:{
  td = ...create the text display here...
}

Trigger: {
  ... send data to the text display td here ...
```

```
}
Save: {
  /* save the location and size of the display */
  save_int(td->x);
  save_int(td->y);
  save_int(td->width);
  save_int(td->height);
}

Restore: {

  int x,y,width,height;
  /* create a text display if not already created */
  if (!td) {
    td = CreateTextDisplay(self_name);
  }
  /* read its size and location from the param file*/
  restore_int(&td->x);
  restore_int(&td->y);
  restore_int(&td->width);
  restore_int(&td->height);
  /* set the display to the new size and location */
  RestoreTextDisplay(td);
}

Destroy: {
  ... destroy the handle to the text display here ...
}
```

Example Primitives

The following are examples of trigger primitives that have **Save** and **Restore** methods and use save/restore functions.

widget/displays/text_display – the full implementation of the example above.
widget/displays/v_disp – shows how arrays of data can be saved.
discrete/string/A_Hold – a primitive that records the last string input to the trigger box as the default value out of the box.

restore_double

Synopsis

```
restore_double(double *value);
```

Description

When a parameter file is loaded and a trigger primitive's **Restore** method is called, then the **restore_double** function restores the double parameter **value** saved to the parameter file by the **save_double** function.

restore_float

Synopsis

```
restore_float(float *value);
```

Description

When a parameter file is loaded and a trigger primitive's **Restore** method is called, then the **restore_float** function restores the float parameter **value** that was previously saved to the parameter file by the **save_float** function.

restore_string

Synopsis

```
restore_string(char **value);
```

Description

When a parameter file is loaded and a trigger primitive's **Restore** method is called, then the **restore_string** function restores the zero terminated character string parameter **value** that was previously saved to the parameter file by the **save_string** function.

save_int

Synopsis

```
save_int(int value);
```

Description

When a parameter file is saved and a trigger primitive's Save method is called, then the **save_int** function saves an integer parameter **value** to the parameter file.

save_double

Synopsis

```
save_double(double value);
```

Description

When a parameter file is saved and a trigger primitive's Save method is called, then the **save_double** function saves a double parameter **value** to the parameter file.

save_float

Synopsis

```
save_float(float value);
```

Description

When a parameter file is saved and a trigger primitive's Save method is called, then the **save_float** function saves a float parameter **value** to the parameter file.

save_string

Synopsis

```
save_string(char *value);
```

Description

When a parameter file is saved and a trigger primitive's **Save** method is called, then the **save_string** function saves a zero terminated character string parameter **value** to the parameter file.

Functions

The functions listed in this section are directly available in the libraries against which Gedae primitives and other libraries are linked. A user may use these functions within primitives, but unlike the built-in functions, they also may be used within external libraries to be called by the primitives.

There are some restrictions on the calling context of the functions. Some of the functions can only be called from the calling tree of certain methods of certain primitive types. For example, the embTerminateError can only be called from stream primitive Start, Reset or Apply methods. Other functions can be called in any context. For example, the function embWallclock can be called from any function to get the current wallclock time.

embBoxName

Synopsis

```
char *embBoxName(void);
```

Box Types

```
static
cyclic
```

Methods

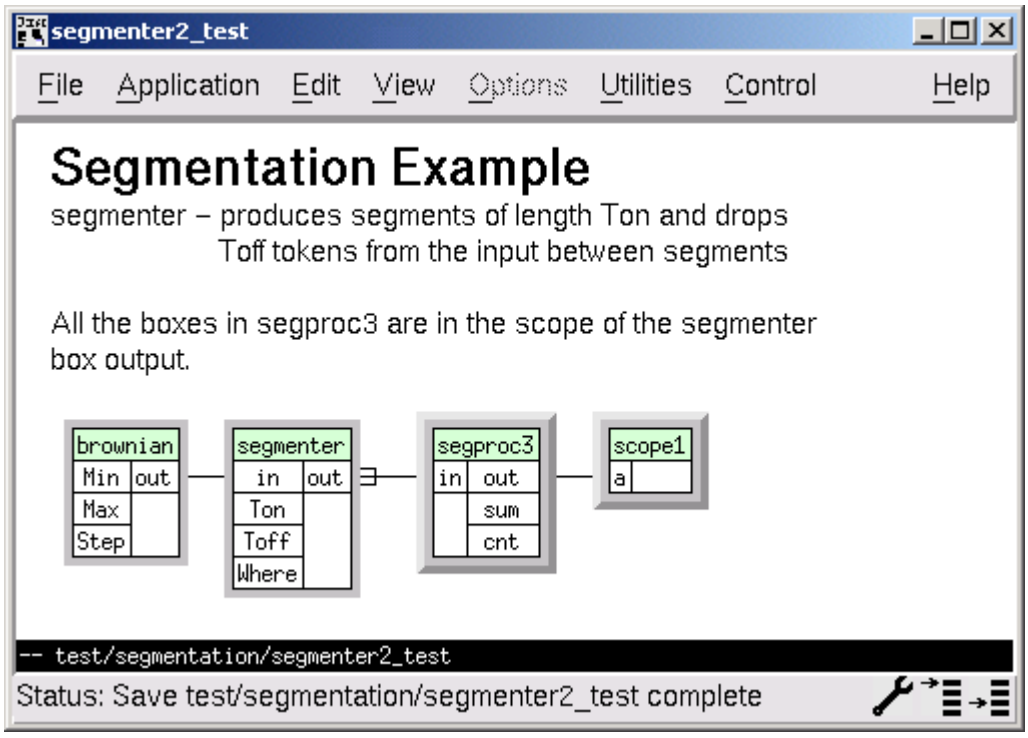
All methods

Description

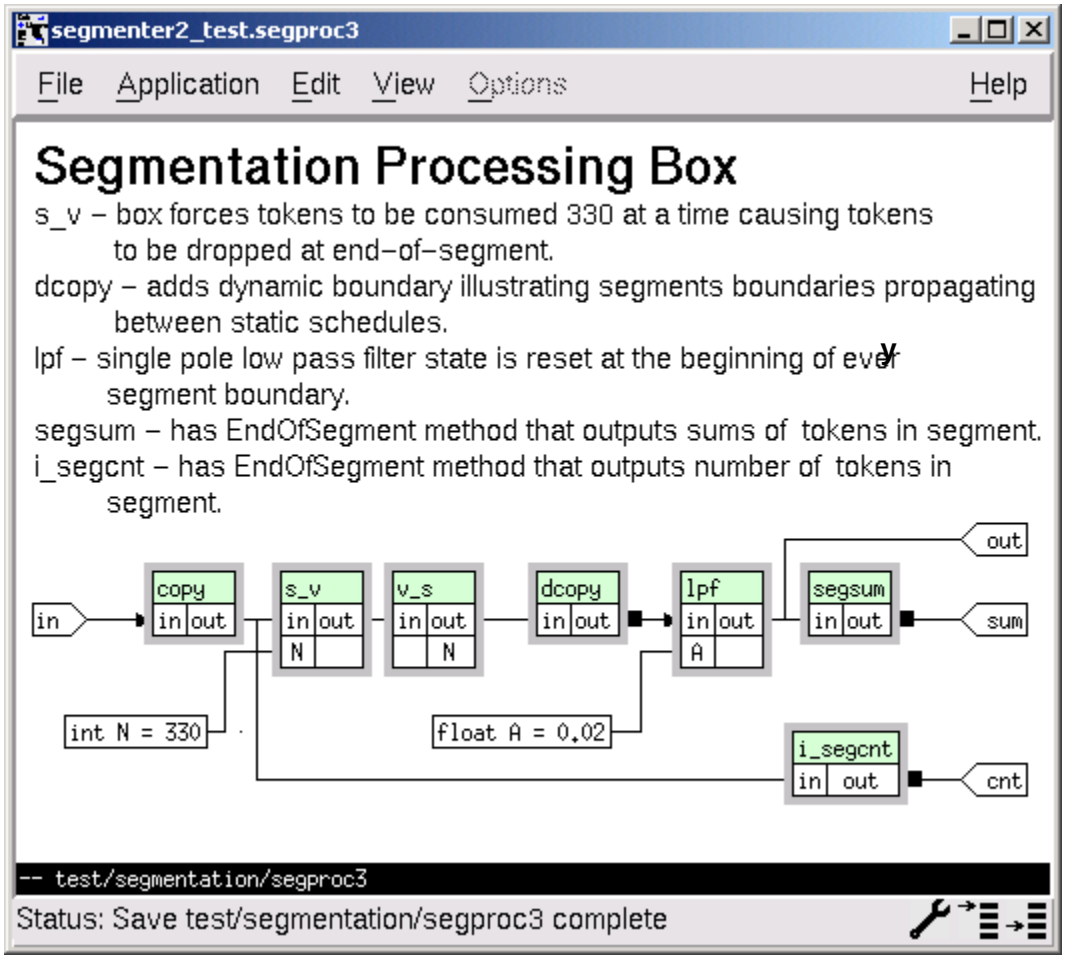
The function **embBoxName** returns the hierarchical name of the primitive from which it is called. The name does not include the top-level graph name. The name provides a unique identifier for any primitive instance in the graph.

Examples

In the following example graph:



Top-level segmentation control graph



A call to the **embBoxName** function from the **lpf** primitive would return the value **"segproc3.lpf"**.

Example Primitives

This function is primarily used for debugging, so there are no examples in the standard library.

embBreak

Synopsis

```
void embBreak(char *reason);
```

Box Types

static
cyclic

Methods

Apply

Description

The function **embBreak** causes a breakpoint to be invoked under program control. When the function executes it forces the partition executing the primitive to stop and brings up the Gedae development environment debug dialog. The debug dialog indicates which partition is stopped and which box invoked the break point.

Examples

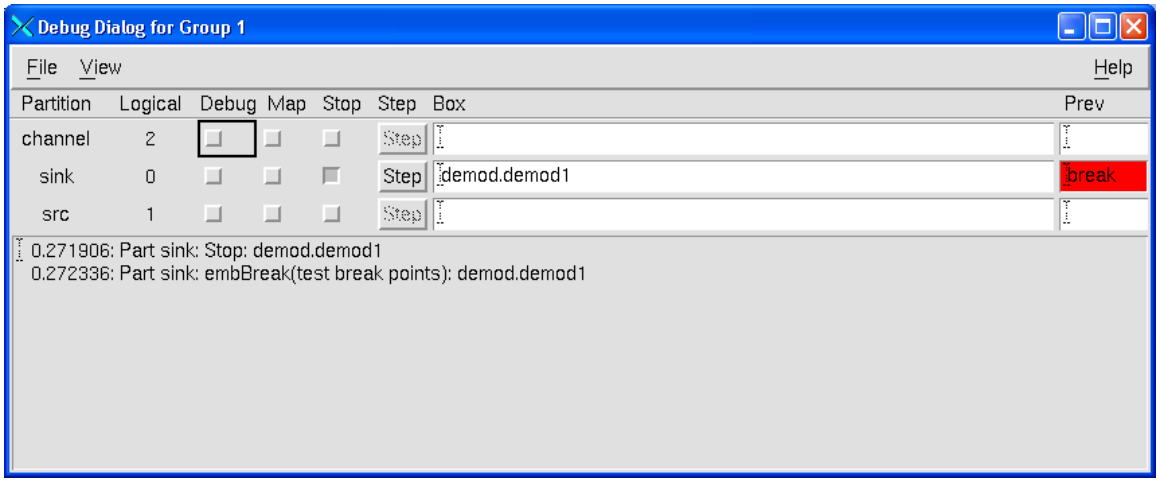
Running the graph

gedae –file demo/comm/e_comm –gr embedded

We then add a call to embBreak to the demod.demod1 primitive as:

```
Name: demod1
Type: static
Input: {
  stream float in(2);
  …
}
…
Output: {
  stream complex out;
}
…
Apply: {
  embBreak("test break points");
  oqpsk_demod1(in,out,size(in),Adc,Ga,Aref,&D_dc,(float
*)Dh,&D_AGC);
}
```

Running the application causes the debug dialog to pop up as below.



At this point the user can examine the sink partitions memory using the Map toggle and single step through the execution of primitives using the Step button. The graph can be restarted by toggling the Stop button.

embCalloc

Synopsis

```
void *embCalloc(char *memtype,
                int nelems, int token_size);
```

Box Types

All

Methods

All

Description

The **embCalloc** function returns a pointer to memory from the memory block named **memtype** of size **nelems*token_size**. The memory area is cleared to zero value by the function. The **embCalloc** function is identical to **calloc** with the exception of the **memtype** field. The **memtype** field's meaning is BSP specific. Many Gedae BSPs ignore this field and simply call **calloc**. Other Gedae BSPs provide different heaps that can be managed, and the heap from which the memory is allocated can be selected using **memtype**. The **memtype** **"default"** is always implemented and indicates allocation should be done off the standard heap. Memory allocated with **embCalloc** should be freed calling **embFree**.

Examples

```
float *x = embCalloc("default",10,sizeof(float));
```

creates a zero filled array of 10 floating point elements from the "default" memory bank.

Example Primitives

Generally, allocating and freeing of memory is less desirable than having the memory use preplanned. Gedae's standard library has no primitives that allocate memory.

See Also

mem_type, embFree

embFClose

Synopsis

```
void embFClose(int fd);
```

Box Types

All

Methods

All

Description

The function **embFClose** closes the file descriptor **fd** opened by **embFOpen**.

Examples

If a primitive's **Start** method opens a file descriptor using **embFOpen** as:

```
Local: {
  int fd;
}

Start: {
  fd = embFOpen(Filename,"r");
}
```

then the **Terminate** method of the primitive should close the file descriptor using **embFClose** as:

```
Terminate: {
  embFClose(fd);
}
```

Similarly, if the **Reset** method of a primitive opens a file descriptor using **embFOpen**, then the **Destroy** method of the primitive should close it using **embFClose**.

Example Primitives

embeddable/stream/source/read – reads a binary file, and when the file has been completely read it closes and reopens the file. The file is opened from the **Reset** method. As a result, this primitive would open the file once at

the beginning of execution if the primitive is not part of a segmented subgraph or at the beginning of each segment if the primitive is part of a segmented subgraph. The file descriptor is closed by the Destroy method at both the end-of-graph execution and at the end of every segment.

`embeddable/stream/sink/write` – writes a stream to a binary file.

See Also

`embFopen`

embFFlush

Synopsis

```
void embFFlush(int fd);
```

Box Types

All

Methods

All

Description

The **embFFlush** function flushes any data written to file descriptor **fd** opened by **embFOpen.**

See Also

embFOpen

embFOpen

Synopsis

```
int embFOpen(char *name,char *access);
```

Box Types

All

Methods

All

Description

The **embFOpen** opens up a file descriptor for the path given by name with access writes specified by access. This function causes a call to the C standard library function **fopen(name,access)** from the command program. As a result, the file will successfully open only if the **name** of the file is relative to where the command program was executed, and the command program has rights to open the file with the access requested. The access parameter is defined in the same way that it is defined for **fopen** on the host processor.

Examples

See **embFClose**

Example Primitives

embeddable/stream/source/read – reads a binary file
embeddable/stream/sink/write – writes a binary file
embeddable/stream/source/scanf – reads an ascii file of floats
embeddable/stream/sink/printf – write an ascii file of floats

See Also

embFClose, embFPrintChar, embFScanChar, embFWrite, embFRead

embFPrintChar

Synopsis

```
embFPrintChar(int fd, int nelems, char *buf);
```

Box Types

All

Methods

All

Description

The function **embFPrintChar** prints **nelem** ASCII characters from the buffer **buf** to the file descriptor **fd**, which was created with write access by **embFOpen**.

See Also

embFOpen

embFRead

Synopsis

```
int embFRead(int fd,int num,int size,void *buf);
```

Box Types

All

Methods

All

Description

The function **embFRead** reads binary data from the file descriptor **fd**. The number of binary elements read from the file is given by parameter **num** and each element is **size** bytes in length. The data is read into buffer **buf,** which must be large enough to contain **num*size** bytes. Function **embFRead** returns the number of elements actually read. The file descriptor that is passed to **embFRead** should have been opened with **embFOpen** with access writes of **"rb"**.

Examples

Given a file "xyzzy" containing 100 bytes, consider the following code:

```
int fd = embFOpen("xyzzy","rb");
int nelems = 200;
int bytes_per_elem = 4;
char *buffer = embCalloc("default",200,4);
int nread = embFRead(fd,nelems,bytes_per_elem,buffer);
embFClose(fd);
```

Since each element contains 4 bytes, the 100 byte file only contains 25 elements. While it was requested that 200 elements be read, the call to embFRead will return 25- the actual number of elements read.

Example Primitives

embeddable/stream/source/read

See Also

embFOpen, **embFWrite**

embFScanChar

Synopsis

```
int embFScanChar(int fptr,int size,char *buf);
```

Box Types

All

Methods

All

Description

The function **embFScanChar** attempts to read **size** characters from the file descriptor **fptr** into buffer **buf**. It returns the number of characters actually read. The file descriptor passed to embFScanChar should have been created with a call to **embFOpen** with access writes of **"r"**.

embFWrite

Synopsis

```
EXPORT void embFWrite(int fd,int num,
                      int size,void *buf);
```

Box Types

All

Methods

All

Description

The function **embFWrite** writes binary data from the file descriptor **fd**. The number of binary elements written to the file is given by parameter **num** and each element is **size** bytes in length. The data is written from the buffer **buf,** which must be **num*size** bytes in size. The file descriptor should have been opened with **embFOpen** with access writes of **"wb"**.

See Also

embFRead, **embFOpen**

embFree

Synopsis

```
void embFree(char *memtype, void *ptr);
```

Box Types

All

Methods

All

Description

Function **embFree** frees a block of memory pointed to by **ptr** that was previously allocated using **embCalloc**. The **memtype** field should be identical to the **memtype** field used by **embCalloc** when **ptr** was created.

Examples

```
float *x = embCalloc("default",10,sizeof(float));
```

creates a zero filled array of 10 floating point elements from the "default" memory bank.

To free this memory call

```
embFree("default",x);
```

See Also

embCalloc

embGlobalStop

Synopsis

```
void embGlobalStop(char *reason);
```

Box Types

static
cyclic

Methods

Apply

Description

The function **embGlobalStop** causes a breakpoint to be invoked under program control. When the function executes it forces the partition executing the primitive and all other partitions to stop and brings up the Gedae development environment debug dialog. The debug dialog indicates which primitive invoked the break point.

Examples

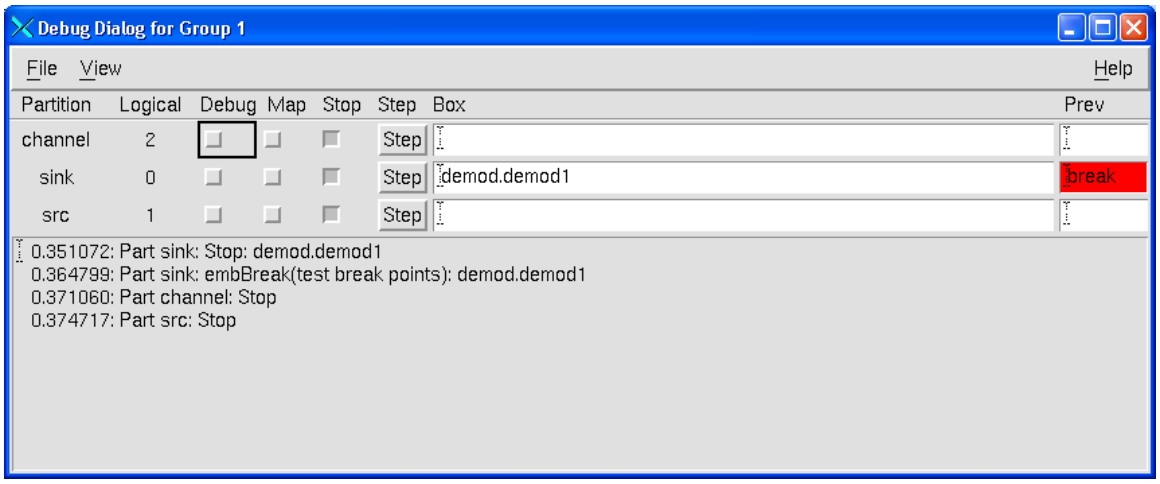
Running the graph

gedae –file demo/comm/e_comm –gr embedded

We then add a call to embGlobalStop to the demod.demod1 primitive as:

```
Name: demod1
Type: static
Input: {
  stream float in(2);
  …
}
…
Output: {
  stream complex out;
}
…
Apply: {
  embGlobalStop("test break points");
  oqpsk_demod1(in,out,size(in),Adc,Ga,Aref,&D_dc,(float
*)Dh,&D_AGC);
}
```

Running the application causes the debug dialog to pop up as below. In this case all the partitions are put into the Stopped state.



At this point the user can examine the partitions memory using the Map toggle and single step through the execution of primitives in any partition using the Step button. The partitions can be restarted by toggling the Stop button.

embNoteProgress

Synopsis

```
void embNoteProgress(void);
```

Box Types

static
cyclic

Methods

Apply

Description

The function **embNoteProgress** indicates that a primitive's **Apply** method was partially executed but did not complete. As a result of making this call, a primitive will release control to other static schedules containing primitives that are ready to fire. The primitive will be tried again at a later point and should record enough information in its **Local** variables so that it can resume execution where it left off. The **embNoteProgress** function causes identical behavior to the **embSuspendRetry** function but the Trace Table will show that the primitive has partially executed rather than blocked. Often at the same time the **embNoteProgress** function is called, the **embSetGranularity** function is also called to indicate how many firings an execution of the primitive successfully completed. This information is only used on the Trace Table and does not affect the behavior of the graph.

Examples

Input device primitives are typical examples of primitives that use **embNoteProgress.** Such a primitive might be able to read only N samples from the input device during an execution of the **Apply** method. If N is less than the granularity, then the primitive will not be able to complete. The primitive must keep track of how much data was actually read so it can begin where it left off on the next firing of the primitive.

In the following code we hypothesize the following two functions:

1. openInputDevice that creates a handle to an input device.
2. readDataFromInputDevice that has three parameters: a handle of an input device, a pointer to be filled with data from the device and the number of

samples to be read from the device. Any samples not read remain in the input device and are ready to be read on the next call to readDataFromInputDevice.

```
Local: {
  int where;
  void *handle;
}

Start: {
  handle = openInputDevice();
  where = 0;

}
Apply: {
  int N = readDataFromInputDevice(handle,
                                  out+where,
                                  granularity-where);
  if (N) {
    where += N;
    if (where < granularity) {
      embNoteProgress();
    } /* otherwise we successfully completed! */
    embSetGranularity(N);
  } else {
    embSuspendRetry("no data available");
  }
}
```

See Also

embSuspendRetry, embSetGranularity

embGetPeriod

Synopsis

```
void embGetPeriod(int *policy, int *units,
                  int *msw, int *lsw);
```

Box Types

static, cyclic

Methods

Reset, Apply

Description

The function **embGetPeriod** returns four values - **policy**, **units**, **msw** and **lsw** - that describe the scheduling policy and periodicity of the calling primitive's static schedule. The **policy** returned is either **DATAFLOW** or **PERIODIC**. If **DATAFLOW,** then the schedule will run when ready and the other fields are of no consequence. If **PERIODIC,** then the schedule is set to run with a periodicity described by the other three parameters. If **units** is **CLOCK_TIC** (the most typical case), then if ready, the schedule will run no more frequently than an elapse time in seconds given by the **msw + 1e-9*lsw**. Otherwise, the units are **SCHEDULE_TIC** and the schedule will run after **1000000000*msw+lsw** other schedules have executed.

The period can be set either from the Schedule Info Dialog or from a call from any primitive in the static schedule to **embSetPeriod**.

See Also

embSetPeriod

embGetPriority

Synopsis

```
int embGetPriority(void);
```

Box Types

static
cyclic

Methods

Start
Reset
Apply

Description

The function embGetPriority returns the priority of the calling primitive's static schedule as set by either a previous call to embSetPriority or as set by the application developer in the Schedule Info Dialog. The schedule that is ready to run and has the highest priority runs first.

See Also

embSetPriority

embGetSchedule

Synopsis

```
Schedule embGetSchedule(void);
```

Box Types

static
cyclic

Methods

Start
Reset
Apply

Description

The **embGetSchedule** function is usually called from the start method and returns a handle to the **Schedule** data structure for this primitive. The **embGetSchedule** function is usually used in conjunction with **embPause** and **embResume**. Schedules that are paused using **embPause** must be resumed by an interrupt handler, thread or callback routine that calls **embResume(thread,schedule)** when the primitive becomes ready to execute again. The **schedule** parameter should be set to the value returned by embGetSchedule, and the **thread** parameter should be set to the value returned by **embSelf**. Usually these values are registered as call-data to the interrupt handler, thread or callback started from the primitives **Start** method.

Examples

See **embResume**

Example Primitives

See **embResume**

See Also

embPause, **embResume**, **embSelf**

embMemcpy

Synopsis

```
void *embMemcpy(void *dest, void *src, int n);
```

Box Types

All

Methods

All

Description

The function **embMemcpy** copies **n** bytes from **src** to **dest**. This function is semantically identical to the standard C library function **memcpy**. The implementation is BSP specific. Each BSP should strive to make **embMemcpy** as efficient as possible given the alignment of **dest**, **src** and the value of **n**. For example, for 32 bit word aligned operands and **n** a multiple of 4, **embMemcpy** should call the vector library function **e_vmov**.

embName

Synopsis

```
char *embName(void);
```

Box Types

All

Methods

All

Description

The function **embName** returns the name of the partition in which the primitive is running.

embPause

Synopsis

```
void embPause(void);
```

Box Types

static
cyclic

Methods

Apply

Description

The function **embPause** puts a primitive in the paused state. The function is similar to **embSuspendRetry** except that the calling primitive will only be woken up by a subsequent call to **embResume**. The functions **embPause** and **embResume** provides an interrupt driven model of execution rather than the polling model provided by **embSuspendRetry**. When **embPause** is called, it signals the Gedae RTK that the primitive has not completed its execution. The primitive **Apply** method may need to record the fact that the **embPause** was called so that when the primitive is called again, it can continue execution from the point at which it stopped.

Examples

See **embResume**

Example Primitives

See **embResume**

See Also

embResume

embPrintChar

Synopsis

```
void embPrintChar(int size,char *buf);
```

Box Types

All

Methods

All

Description

The function **embPrintChar** prints a buffer **buf** having **size** characters to the terminal.

embProcStats

Synopsis

```
void embProcStats(double *total_time,
                  double *exec_time);
```

Box Types

All

Methods

All

Description

The function **embProcStats** sets the parameter's total, which is the total time in seconds since the last call to **embProcStats**. It also returns the parameter's exec, which is the time that the RTK spent running primitives since the last call to **embProcStats**.

Examples

```
double total_time;
double exec_time;
double percentage_utilization;
double idle_time;

embProcStats(&total_time,&exec_time);

idle_time = total-exec;
percentage_utilization = 100*exec_time/total_time;

printf("The processor was utilized %4.1f%%\n"
       "of the time over the last %g seconds\n"
       "and was idle for %g seconds.\n",
       percentage_utilization,
       total_time,
       idle_time);
```

Example Primitives

embeddable/stream/void/void_procstats – outputs the percentage processor utilization since the last call to the primitive.

embResume

Synopsis

```
void embResume(void *thread, Schedule s);
```

Box Types

static
cyclic

Methods

None – function called from interrupt handler, thread or callback registered by primitives **Start** method.

Description

The **embResume** function resumes a schedule previously paused by **embPause**. To use **embResume** the primitive must register an interrupt handler, thread or callback that will call **embResume** whenever the device moves into the ready state. When a primitive's **Apply** method detects that a device is not ready the **Apply** method calls **embPause** putting the primitive into the paused state. Then when the device becomes ready the interrupt handler calls embResume, which puts the primitive back in the ready state.

In order to resume a schedule, the **embResume** function must be passed the values returned by **embSelf** and **embGetSchedule**. These functions must be called from the primitives **Start** method and recorded in order to be passed to the callback or interrupt handler that will call **embResume**. Calling **embResume** when **embPause** was not previously called has no effect.

Examples

The following code shows how a typical input/output device can be made non-polling using **embPause** and **embResume**. We hypothesize some functions for a device as follows:

```
void *startDevice(void)
void registerDeviceCallback(
          void *handle,
          void (*callback)(void *),
```

```
          void *calldata);
int deviceReady(void *handle);
```

The startDevice function returns a device handle. registerDeviceCallback registers a callback function that will be called any time the device becomes ready. deviceReady is a function that returns true whenever the device is ready, that is, has data available to read. With these functions defined, we can now present our hypothetical device handler:

```
Name: device_handler
Type: static
Input: {
  ...device parameters...
}

Local: {
  void *handle;
  CallData cd;
}

Output: {
  stream float out; /* output data stream from device
*/
}
Include: {
  typedef struct {
    Schedule s;
    void *thread;
  } CallData;

  static void callback(CallData cd) {
    embResume(cd->thread,cd->s);
  }
}

Start: {
  handle = startDevice();
  cd->s = embGetSchedule();
  cd->thread = embSelf();
  registerDeviceCallback(handle,callback,cd);
}

Apply: {
  if (deviceReady(handle)) {
    ... get data from device and
    place in output data stream ...
  } else {
```

```
        embPause();
    }
}
```

In the above example, the **Start** method fills in the **CallData** structure **cd->s** and **cd->thread** elements by calling **embGetSchedule** and **embSelf**. It then registers the callback routine **callback** that is called when the device becomes ready. In the **Apply** method, if the device is ready, then the **Apply** method runs to completion, and if not, it calls **embPause**, which puts the primitive and its associated schedule in the paused state. The primitive will not execute again until the device becomes ready. When the device becomes ready, the callback routine calls **embResume**, thereby making the device ready.

It might seem that if **embResume** is called between the **deviceReady** returning **0** and the call to **embPause** that the **embResume** call might be missed. If the **embResume** were missed the primitive would go into the paused state with no chance to recover. This situation is avoided because when **embResume** is called before **embPause** the RTK records the call. The next call to **embPause** will see that **embResume** was called and cause the primitive to go into a polling state instead of going into the paused state – much as if **embSuspendRetry** had been called instead. Only after a second call to embPause will the primitive go into the paused state.

Example Primitives

embeddable/stream/audio/audioIn_nt – audio input device for windows PCs
vfg/vfg_video_in – video input device coded using the Gedae vfg library.

See Also

embPause, **embGetSchedule**, **embSelf**, **embSuspendRetry**

embSbAlloc

Synopsis

```
void *embSbAlloc(char *type, int bytes, int users);
```

Box Types

```
static
cyclic
```

Methods

```
Apply
```

Description

The **embSbAlloc** function is designed for use in **Apply** methods to create pointer values that are to be passed out of the **Apply** method to downstream primitives. It will be the responsibility of the downstream primitives to free the pointers using **embSbFree**,, to make copies of the pointers using **embSbCopy** or forward the pointers using **embSbForward**. The **embSbAlloc** takes the memory type as its first parameter, and the number of bytes to allocate as its second parameter. (The memory type is passed to **embCalloc**, which is called by **embSbAlloc**). The final parameter is **users**, which is the number of users of this pointer. If **users** is set to **N,** then **N** calls to **embSbFree** must be made before the pointer is actually freed using **embFree**.

The **embSbAlloc** type field is usually set by querying the type of the output using the built-in function **mem_type**. The **users** field should always be set by querying how many destination primitives are using the built-in function **n_dests**.

Examples

```
Name: vv_Float1d
Type: static
Input: {
  stream float in[n,MAX];
}
Output: {
  stream Float1dRec out;
}
```

```
Include: {
#include <valuetypes.h>
}

Apply: {
  int ndests = n_dests(out);
  char *memtype = mem_type(out);
  if (ndests) {
    int i;
    for (i=0; i<granularity; i++) {
      int bytes = n[i]*sizeof(float);
      out[i].data = embSbAlloc(memtype,bytes,ndests);
      out[i].n = n[i];
      embMemcpy(out[i].data,&in[i*MAX],bytes);
    }
  }
}
```

See Also

embSbFree, embSbCopy, embSbForward, embCalloc, embFree, mem_type, n_dests

embSbFree

Synopsis

```
void embSbFree(void *buf);
```

Box Types

static
cyclic

Methods

Apply

Description

The function **embSbFree** frees a shared pointer allocated with **embSbAlloc**. The **embSbFree** decrements the use count of the pointer and only calls **embFree** on the pointer when the use count goes to zero.

Examples

```
Name: Float1d_magn
Type: static
Input: {
  stream Float1dRec in;
}
Output: {
  stream float out;
}

Include: {
#include <valuetypes.h>
#include <math.h>
}

Apply: {
  int i;
  for (i=0; i<granularity; i++) {
    Float1d x = &in[i];
    float *data = x->data;
    int j;
```

```
        float sum = 0;
        for (j=0; j<x->n; j++) {
          sum += data[j]*data[j];
        }
        out[i] = sqrt(sum);
        embSbFree(data);
      }
    }
```

See Also

embSbAlloc, **embSbCopy**, **embSbForward**, **embFree**

embSbCopy

Synopsis

```
void *embSbCopy(void *buf, int users);
```

Box Types

```
static
cyclic
```

Methods

```
Apply
```

Description

The **embSbCopy** function returns a shared pointer the same size and memory type as the input pointer **buf**. If the use counter of the input buffer is 1, then **embSbCopy** returns the input buffer and sets the new use count of this buffer to **users**. If the use count of the input buffer is greater than 1, then the **embSbCopy** creates a new buffer and sets its user counter to **users**. The output buffer may actually be the same pointer as the input buffer, so the algorithm must allow the input buffer to be modified in-place. If the algorithm can only be performed out-of-place, then the output buffer should be created with **embSbAlloc.** After the algorithm has run, the input buffer should be freed using **embSbFree**.

Examples

In the following example, **embSbCopy** creates an output data buffer the same size as the input data buffer.

```
Name: Float1d_sqrt
Type: static
Input: {
  stream Float1dRec in;
}
Output: {
  stream Float1dRec out;
}

Include: {
#include <math.h>
```

```
}

Apply: {
  int i;
  for (i=0; i<granularity; i++) {
    Float1d x = &in[i];
    Float1d y = &out[i];
    float *indata = x->data;
    float *outdata = embSbCopy(indata,n_dests(out));
    int j;
    float sum = 0;
    y->n = x->n;
    y->data = outdata;
    for (j=0; j<x->n; j++) {
      outdata[j] = sqrt(indata[j]);
    }
  }
}
```

An example of an out-of-place version of the **Apply** method that uses **embSbAlloc** and **embSbFree** is given below:

```
Apply: {
  int i;
  for (i=0; i<granularity; i++) {
    Float1d x = &in[i];
    Float1d y = &out[i];
    float *indata = x->data;
    char *type = mem_type(out);
    float *outdata = embSbAlloc(type,n_dests(out));
    int j;
    float sum = 0;
    y->n = x->n;
    y->data = outdata;
    for (j=0; j<x->n; j++) {
      outdata[j] = sqrt(indata[j]);
    }
    embSbFree(in);
  }
}
```

See Also

embSbFree, embSbAlloc, **embSbForward**

embSbForward

Synopsis

```
void  embSbForward(void *buf, int users);
```

Box Types

```
static
cyclic
```

Methods

```
Apply
```

Description

The **embSbForward** function forwards the pointer **buf** from the input to the output. This function should be used only if the pointer is being forwarded without modification. The parameter users should be set to the number of output destinations using **n_dests(out)**. The use count of the buffer is incremented by **users-1**. If the number of destinations is **0**, then we may actually decrement the use count by **1**, and if the use count goes to zero, then **embFree** will be called on the buffer.

Examples

```
Name: Float1d_demux
Type: static
Input: {
  stream Float1dRec in(m);
}
Output: {
  stream Float1dRec [m]out;
}
Apply: {
  int i,j;
  for (i=0; i<granularity; i++) {
    for (j=0; j<m; j++) {
      Float1d outj = out[j];
      outj[i] = *(in++);
      embSbForward(outj[i].data,n_dests(out[j]));
    }
  }
}
```

See Also

embSbAlloc, **embFree**, **embSbCopy**, **n_dests**

embSbBytes

Synopsis

```
int embSbBytes(void *buf);
```

Box Types

static
cyclic

Methods

Apply

Description

The function **embSbBytes** returns the number of bytes allocated for a shared memory buffer by **embSbAlloc**.

See Also

embSbAlloc

embSbType

Synopsis

```
char *embSbType(void *buf);
```

Box Types

static
cyclic

Methods

Apply

Description

The **embSbType** function returns the name of the memory block passed to **embSbAlloc** when the buffer was created.

See Also

embSbAlloc, **embSbForward**, **mem_type**

embSelf

Synopsis

```
void *embSelf(void);
```

Box Types

static
cyclic

Methods

Start

Description

The **embSelf** parameter returns the thread of execution that can be passed to **embResume**. The **embSelf** function is BSP specific and is designed in conjunction with the BSP functions **embGoToSleep** and **embWakeup**. The function **embSelf** often returns a 0, as many BSPs do not require knowing the execution thread in order to wake up the process.

Examples

See **embResume**

Example Primitives

See **embResume**

See Also

embPause, **embResume**, **embGetSchedule**

embSetGranularity

Synopsis

```
void embSetGranularity(int granularity);
```

Box Types

static
cyclic

Methods

Apply

Description

The **embSetGranularity** function is called in conjunction with **embNoteProgress** to indicate how much progress a primitive made through the granularity loop during execution of its **Apply** method. The **embSetGranularity** function has no effect on behavior and is only useful for debugging purposes. For example, the user of the Gedae Trace Table can query any primitive execution displayed on the table for its execution granularity. If during the execution being queried the Apply method called **embSetGranularity(10)**, then the granularity displayed to the user is the value **10**. If the **Apply** method did not call **embSetGranularity** during the execution, then the value displayed to the user is the **granularity** of the **Apply** method.

Examples

see **embNoteProgress**

See Also

embNoteProgress

embSetPeriod

Synopsis

```
void embSetPeriod(int sec, int nsec);
```

Box Types

static
cyclic

Methods

All

Description

The **embSetPeriod** function sets a primitive's static schedule to run with a periodicity in seconds of **sec+1e-9*nsec**. When either **sec** or **nsec** are non-zero, **embSetPeriod** sets the scheduling policy to **PERIODIC** and the scheduling units to **CLOCK_TIC**. When **sec** and **nsec** are both zero, **embSetPeriod** sets the scheduling policy to **DATAFLOW**.

Examples

In the following example, **embSetPeriod** is called in both the **Start** and **Apply** methods. A possible modification of this function would be to remove the **Apply** method, in which case, the period for the schedule could not be changed once the graph has been reset and is running.

```
Name: void_rate
Type: static
Input: {
  stream void in;
  float T;
}

Start: {
  int sec = T;
  int nsec = (T-sec)*1e9;
  embSetPeriod(sec,nsec);
}

Apply: {
```

```
    int sec = T;
    int nsec = (T-sec)*1e9;
    embSetPeriod(sec,nsec);
}
```

Example Primitives

embeddable/void/void_rate – sets the period of a primitive to run every T seconds.

See Also

embGetPeriod

embSetPriority

Synopsis

```
void embSetPriority(int priority);
```

Box Types

static
cyclic

Methods

Start
Reset
Apply

Description

The **embSetPriority** function sets the priority of the primitive's static schedule. When multiple schedules are ready to run, the schedules with the highest priority will always run first. By default all schedule priorities are set to 0, thereby insuring that schedules set to priority 1 will run, when ready, before any other schedule with unset priorities.

Care should be taken when calling **embSetPriority** so that the calling schedule will not run to the exclusion of all other schedules. Such runaway behavior can occur when the schedule can run asynchronously from the other schedules (no data flow to any other schedules), and when the schedule has not been set to run periodically.

A schedule's priority can be changed dynamically by the **Apply** method or on segment boundaries by the **Reset** method, but it is most typically set once at start-up by the primitives **Start** method.

See Also

embSetPeriod, **embGetPriority**

embSuspendQueueWait

Synopsis

```
void embSuspendQueueWait(char *reason);
```

Box Types

static
cyclic

Methods

Apply

Description

If a primitive Apply method determines that the Apply method cannot complete execution until one of multiple nondet input or output queues becomes ready, then the primitive should call **embSuspendQueueWait**. The function **embSuspendQueueWait** puts the primitive in the queue-wait state. The primitive will become ready to fire when data arrives on any input or is consumed from any output.

There are two major ways nondet queues are used. In the deterministic method, the primitives with nondet inputs or outputs will call the amount function to say how much data is required on each input or how much space is required on each output. If any call to an amount function fails, then the primitive must exit and it will only be called again when all the queues for which amount was called have the amount of data specified. By contrast, the primitive that calls **embSuspendQueueWait** is advertising the fact that it may be ready to run when any input gets new data or any output gets more space. This method is nondeterministic as the order of arrival of data on the input queues (which may be affected by processor speeds in a distributed application) may change the order in which data is processed.

Examples

The following example shows how an uncontrolled merge is implemented using the **embSuspendQueueWait** function. In this code, the merge function is ready to fire when there is data available on any input. The variable **T** is the total amount of data available on all the input queues and is calculated by summing the values of **avail(in[i])** for all of the family members of **in**. If some data is available, then the primitive copies data from its inputs to the output and produces

T tokens on the output. If **T** is zero, however, the primitive goes into the queue-wait state by a call to **embSuspendQueueWait**. When any new data arrives on any of the input queues, the primitive is put back in the ready state and will be run.

```
Name: umergef
Type: static
Input: {
  nondet stream float [F]in;
}
Output: {
  dynamic stream float out(F);
}

Apply: {
  int T = 0; /* total data available on all inputs */
  int f;
  /* find the ready input queues and determine T */
  for (f=0; f<F; f++) {
    int ain = avail(in[f]);
    T += ain;                          /* update T */
    if (ain) amount(in[f],ain);    /* prepare in[f] */
  }
  if (T) {
    /* then the box is ready to fire */
    for (f=0; f<F; f++) { /* for each input */
      int ain = avail(in[f]);
      e_vmov(in[f],1,out,1,ain);
      out+=ain;
      consume(in[f],ain);
    }
    produce(out,T);
  } else {
    embSuspendQueueWait("waiting for input");
  }
}
```

Example Primitives

embeddable/stream/logic/umergef

See Also

amount, **avail**

embSuspendRetry

Synopsis

```
void embSuspendRetry(char *reason);
```

Box Type

static
cyclic

Methods

Apply

Description

The function **embSuspendRetry** notes that a primitive has not completed executing and makes a note of the reason that is passed in the **reason** parameter. The schedule that a primitive is part of is put into the pending retry state and will be executed again once the scheduling criteria for this schedule to retry has been met. The default policy is that every other schedule that is ready to fire will be given the chance before the schedule calling **embSuspendRetry** is called. Using **embSuspendRetry** is equivalent to polling on the primitive to find out when it is ready. This polling behavior is in contrast to the interrupt driven capability provided by **embPause** and **embResume**.

Examples

The following code shows a polling input device. In this example, the primitive is required to run at a granularity of 1, and the output interpolate value is set to BufferSize. As a result, the primitive will always produce BufferSize tokens on the output. This explicit setting of the number of tokens produced by the output is typical of I/O primitives, which then dictates the granularity of the boxes attached to them.

For this example, we hypothesize that there are functions:

1. initializeDevice which returns a DeviceHandle initialized to have internal storage of BufferSize.
2. deviceReady which returns true when BufferSize tokens are available in the input device.
3. copyDataFromDevice which will copy BufferSize tokens from the buffer represented by dh to the output pointer out.

```
Name: InputDevice
Type: static
Input: {
  char Name[N];
  int BufferSize;
}
Local: {
  DeviceHandle dh;
}

Output: {
  stream float out(BufferSize);
}
Start: {
  dh = initializeDevice(Name,BufferSize);
}
Apply: {
  if (granularity != 1) {
    embTerminateError("expected granularity of 1");
  }
  if (deviceReady(dh)) {
     /* copy BufferSize tokens from dh to out */
     copyDataFromDevice(dh,out);
  } else {
     embSuspendRetry("input device not ready");
  }
}
```

While the above is a typical example of an I/O primitive, there are many other ways they may be developed. The main purpose of this example is to illustrate a typical use of **embSuspendRetry** and not to exhaustively describe I/O primitive creation.

Example Primitives

Input/Output devices provided by Gedae have all been implemented using **embPause**/**embResume** so there are currently no examples in the Gedae library.

See Also

embPause, **embResume**

embTerminateError

Synopsis

```
void embTerminateError(char *reason);
```

Box Types

static
cyclic

Methods

Start
Reset
Apply

Description

The **embTerminateError** function indicates an exceptional condition has occurred and stops the graph from executing.

Examples

See **embTerminateNormal**

Example Primitives

embeddable/stream/source/read1

See Also

embTerminateNormal

Synopsis

```
void embTerminateNormal(char *reason);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

A call to the function **embTerminateNormal** signals that the calling primitive and the calling primitive's schedule is terminated. The dynamic scheduler will never schedule the calling primitive and its associated schedule again.

Examples

The following primitive reads the file named by **Name** until it reaches the end-of-file. At that time it calls **embTerminateNormal** to indicate that the primitive terminated in an expected fashion. In the **Start** method, **embTerminateError** is called, which indicates a fatal unexpected error that stops the graph from executing.

```
Name: read1
Type: static
 Input: {
   char Name[N];
 }
 Local: {
   int fd;
 }
 Output: {
   stream float out;
 }
 Start: {
   fd = embFOpen(Name,"rb");
   if (!fd) {
     embTerminateError("file open failed");
   }
 }
```

```
Apply: {
  int nread;
  nread = embFRead(fd,granularity,sizeof(float),out);
  if (numread < granularity) {
    embTerminateNormal("end of file reached");
  }
}

Terminate: {
  if (fd) {
    embFClose(fd);
  }
}
```

Example Primitives

embeddable/stream/source/read1

See Also

embTerminateError

embUserEvent

Synopsis

```
void embUserEvent(int number);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **embUserEvent** function adds a time-stamped trace event with that has recorded with it the user passed **number**. These events are visible on the trace table and the user can also set a break point on these events.

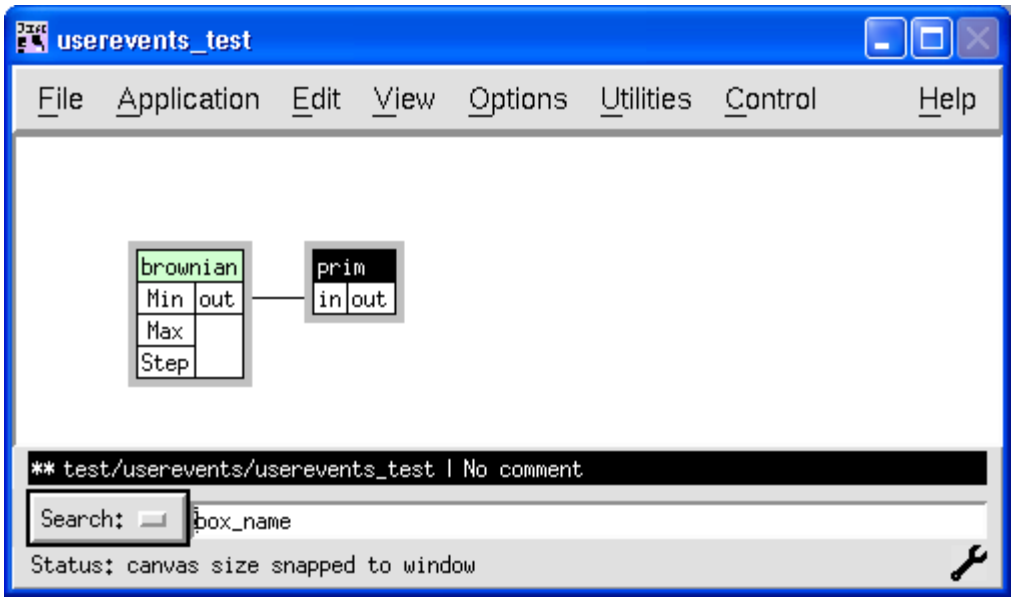
Examples

The following code fragment shows a user event being conditionally added to the trace table when a variable x takes on the value of 5.

```
Apply: {
…
  if (x == 5) {
   embUserEvent(1);
  }
…
}
```

Here is a more extended example of how this function and other associated functions can be used:

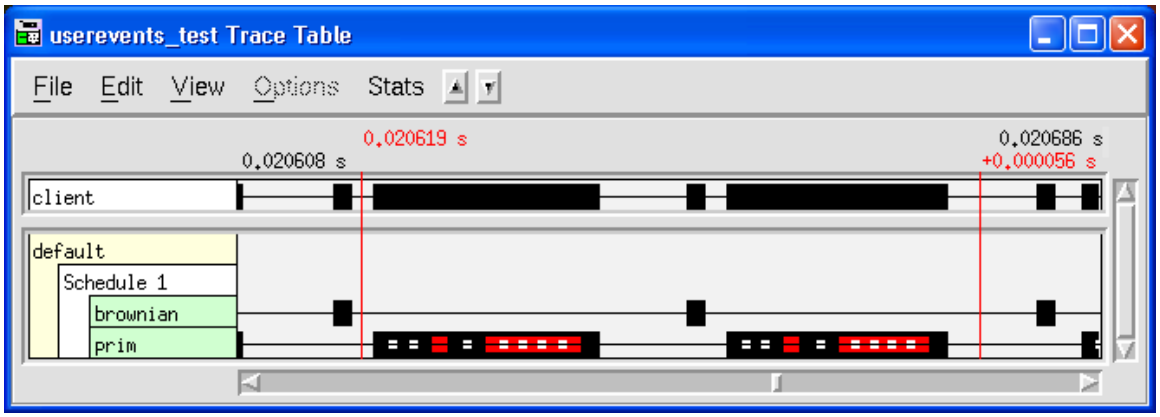
The prim function in the following graph creates each type of user event:



The **Apply** method for prim is:

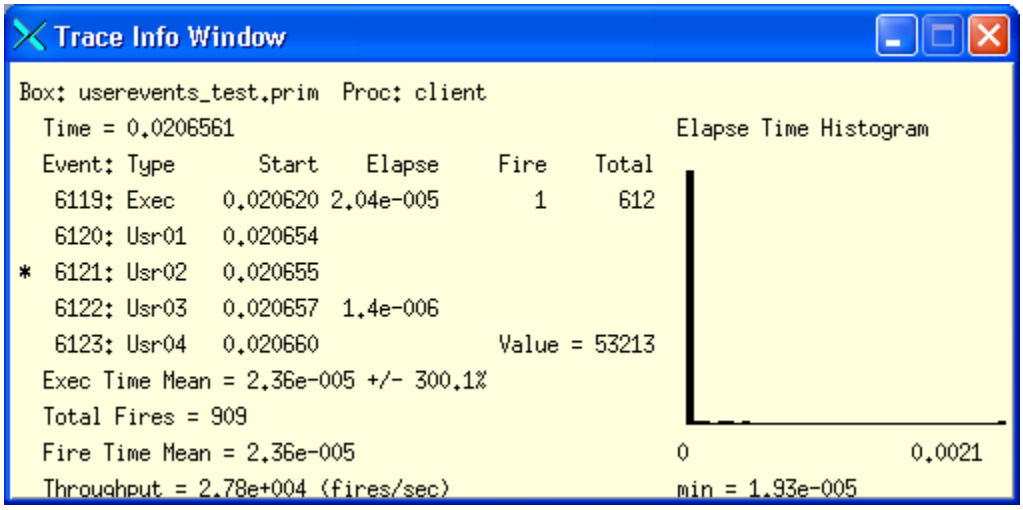
```
Apply: {
  embUserEvent(1);
  embUserEvent(2);
  embUserBeginEvent(3);
  embUserEndEvent(3);
  embUserIntEvent(4,cnt);
  embUserBeginEvent(9);
  embUserFloatEvent(5,3.7);
  embUserFloatEvent(6,3.7);
  if (cnt > 10000) {
     embUserEvent(7);
  }
  if (cnt > 10001) {
     embUserEvent(8);
  }
  cnt++;
  embEndUserEvent(9);
}
```

The firing of **prim** produces the following trace table.



The primitive firing is the black bar. Instantaneous events are the white square within the black rectangle. Interval events are the slightly larger red rectangles.

Clicking on the second white dot in one of the firings of **`prim`** shows the following display with more detailed information about the user events. From this display the user can read the even number, the elapse time (if it is an interval event) and the events value (if any).



Example Primitive

`test/userevents/prim`

See Also

`embUserBeginEvent, embUserEndEvent, embUserIntEvent, embUserFloatEvent`

embUserBeginEvent

Synopsis

```
void embUserBeginEvent(int number);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **embUserBeginEvent** function marks the beginning of a numbered interval event. This function is always paired with a following call to **embUserEndEvent** that is passed the same number. One event is recorded in the trace table event buffer that contains the event number, the event start time and the event stop time.

Currently intervals defined by **embUserBeginEvent** and **embUserEndEvent** can not overlap or be nested. However calls to any of the instantaneous event functions (**embUserEvent**, **embUserFloatEvent** or **embUserIntEvent**) can be made between calls to **embUserBeginEvent** and **embUserEndEvent**.

Examples

The following code fragment shows how **embUserBeginEvent** and **embUserEndEvent** can be used.

```
Apply: {
 …
    embUserBeginEvent(3);
 … code to be timed …
    embUserEndEvent(3);
…
}
```

See **embUserEvent** for a more extended example of the use of this function.

Example Primitive

```
test/userevents/prim
```

See Also

embUserEvent, embUserEndEvent

embUserEndEvent

Synopsis

```
void embUserEvent(int number);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **embUserEndEvent** function marks the end of a numbered interval event. This function is always paired with a preceding call to **embUserBeginEvent** that is passed the same number. One event is recorded in the trace table event buffer that contains the event number, the event start time and the event stop time.

Currently intervals defined by **embUserBeginEvent** and **embUserEndEvent** can not overlap or be nested. However calls to any of the instantaneous event functions (**embUserEvent**, **embUserFloatEvent** or **embUserIntEvent**) can be made between calls to **embUserBeginEvent** and **embUserEndEvent**.

Examples

The following code fragment shows how **embUserBeginEvent** and **embUserEndEvent** can be used.

```
Apply: {
 …
    embUserBeginEvent(3);
 … code to be timed …
    embUserEndEvent(3);
…
}
```

See **embUserEvent** for a more extended example of the use of this function.

Example Primitive

```
test/userevents/prim
```

See Also

embUserEvent, embUserBeginEvent

embUserFloatEvent

Synopsis

```
void embUserFloatEvent(int number, float value);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **embUserFloatEvent** works exactly like **embUserEvent** except that the recorded event contains an additional floating point value. This floating point value can be examined on the trace table and can be used when setting break points.

Examples

The following code fragment shows how **embUserFloatEvent** can be used.

```
Local:
  float x;
}
Apply: {
 …
   embUserFloatEvent(4,x);

}
```

The above code fragment shows the **embUserFloatEvent** function being used to register the value of a primitive's **Local** variable **x**.

See **embUserEvent** for a more extended example of the use of this function.

Example Primitive

```
test/userevents/prim
```

See Also

embUserEvent

embUserIntEvent

Synopsis

```
void embUserEvent(int number);
```

Box Types

```
static
```

Methods

```
Apply
```

Description

The **embUserIntEvent** works exactly like **embUserEvent** except that the recorded event contains an additional integer value. This value can be examined on the trace table and can be used when setting break points.

Examples

The following code fragment shows how **embUserIntEvent** can be used.

```
Local:
  int x;
}
Apply: {
 …
   embUserIntEvent(4,x);

}
```

The above code fragment shows the **embUserIntEvent** function being used to register the value of a primitive's **Local** variable **x**.

See **embUserEvent** for a more extended example of the use of this function.

Example Primitive

```
test/userevents/prim
```

See Also

embUserEvent

embWallclock

Synopsis

```
int embWallclock(int *sec);
```

Box Types

All

Methods

All

Description

The **embWallclock** function returns the wallclock time in seconds and nanoseconds. The nanoseconds are returned in the **embWallclocks** return value and the seconds are returned in the **sec** parameter. The **embWallclock** is used to measure elapse time between calls. The absolute meaning of the times (for example, seconds since Jan 1, 1975) is BSP dependent. The **embWallclock** function is used to collect trace events and processor statistics as returned by **embProcStats**.

Examples

The following code fragment can be used to time a section of code:

```
int start_sec,  stop_sec;
int start_nsec, stop_nsec;

start_nsec = embWallclock(&start_sec);
.. code to be timed ..
stop_nsec  = embWallclock(&stop_sec);

printf("Ellapse time of execution is = %g seconds\n",
(stop_sec-start_sec) + 1e-9*(stop_nsec-start_nsec);
```

See Also

embProcStats

Unmapped to Mapped Memory Transfer Functions

Synopsis

```
void e_putu    (void *mapped, unsigned long unmapped, int bytes);
void e_getu    (void *mapped, unsigned long unmapped, int bytes);
void e_putu_nw (void *mapped, unsigned long unmapped, int bytes,
    int wait_handle);
void e_getu_nw (void *mapped, unsigned long unmapped, int bytes,
                int wait_handle);
int  e_putu_cw (void *mapped, void *unmapped, int bytes);
int  e_getu_cw (void *mapped, void *unmapped, int bytes);
void e_putmu   (void *mapped, unsigned long unmapped,
                int R, int C_mapped, int C_unmapped, DmaListElem
dle);
void e_getmu   (void *mapped, unsigned long unmapped,
                int R, int C_mapped, int C_unmapped, DmaListElem
dle);
void e_putmu_nw(void *mapped, unsigned long unmapped,
                int R, int C_mapped, int C_unmapped, int
wait_handle,
                DmaListElem dle);
void e_getmu_nw(void *mapped, unsigned long unmapped,
                int R, int C_mapped, int C_unmapped, int
wait_handle,
                DmaListElem dle);
int  e_putmu_cw(void *mapped, unsigned long unmapped,
                int R, int C_mapped, int C_unmapped, DmaListElem
dle);
int  e_getmu_cw(void *mapped, unsigned long unmapped,
                int R, int C_mapped, int C_unmapped, DmaListElem
dle);
void e_wait    (int wait_handle);
int  e_cw      (void);
```

Description

Primitives that have declared inputs and outputs unmapped can only access the data in these streams using the primitive handling functions described below:

Unmapped memory refers to integer handles to allocated memory that a processor can not directly access using normal pointer dereferencing, but can only be access through the above function interface. There are 12 functions that are used to move data to and from unmapped memory. These function names have the form:

```
e_(put|get)(u|mu)[(_nw|_cw)]
```

Where the meaning of each part of the function name is:

`put` - Move mapped memory to unmapped memory
`get` - Move unmapped memory to mapped memory
`u` - Move bytes of contiguous data between memories. Parameter mapped indicates the beginning address of mapped memory and parameter unmapped indicates the beginning address of mapped memory.
`mu` - Move a matrix in mapped memory to/from a submatrix tile in unmapped memory. Parameter mapped indicates the address of a tile of size RxC_mapped in mapped memory and the parameter unmapped indicate the address of a matrix of size R*C_unmapped in umapped memory.
`_nw` - Begin the transfer and do not wait for the transfer to complete. To wait for the transfer completion at a later point call `e_wait(wait_handle)`
`_cw` - Begin the transfer and do not wait for the transfer to complete. Return an integer wait handle. To wait for the transfer completion at a later point call the e_wait function on the _cw functions return value. If neither the _nw or _cw prefix is added then the function waits for the transfer to complete.

In the following we refer to functions that have a component as a function of that component type. For example the function e_putmu_nw is a `put` function, an `mu` function and an `nw` function.

The parameters to the above functions are:

`mapped` - a pointer to the region of mapped memory to/from which data is to be moved.
`unmapped` - a long integer value representing a pointer to the unmapped memory region to/from which data is to be moved. The value `unmapped + N` represents the memory address that is `N` bytes offset from `unmapped`.
`bytes` - for `u` functions the number of bytes to transfer
`R` - for `mu` functions the number of rows to be moved between mapped and unmapped memory.
`C_mapped` - for `mu` functions the number of columns in the mapped matrix
`C_unmapped` - for `mu` functions the number of columns in the unmapped matrix. `C_unmapped >= C_mapped`.
`wait_handle` - for `nw` functions the handle returned by any of the `cw` functions (including `e_cw`) that the eventual call to `e_wait` will be passed to wait for completion of the transfers.
`dle` - A memory area of size `R*sizeof(DmaListElemRec)` that must is used as working memory of the mu functions. Typically this area is set in a primitives local section as:

```
Local: {
```

```
  DmaListElemRec dle[R];
}
```

In addition to the 12 functions for moving data between mapped and unmapped memory there are two additional functions that are used to allow programs to synchronize execution with transfer completion. These are the `e_cw` function that creates a `wait_handle` and the `e_wait` function that waits for completion of all functions whose transfers are associated with that wait handle. The `e_cw` function and all of `cw` functions return a `wait_handle` that can be passed to the functions ending in nw and to the `e_wait` function.. The program can wait for the completion of the `cw` function creating the wait handle and all of the `nw` functions passed the wait handle by calling `e_wait(wait_handle)`.

Return value

The `cw` functions return an integer wait handle that can be passed to any of the `nw` functions or the `e_wait` function.

Limitations

3. On the Cell/B.E. the unmapped parameter can only be a 32 bit unsigned long value.
4. There is no guarantee that the _cw functions return a unique wait_handle. As a result a call to e_wait on a particular wait_handle may end up waiting for more functions to complete than anticipated by the user.
5. The current Cell BSP also uses wait handles for send/recv boxes but these are generated differently than the wait handles for the e-functions. This may result in the inefficiency of an e_wait call waiting for both the e-function to complete and a BSP communication to complete when it should only wait for the e-function.
6. While the mu functions allow a submatrix from an unmapped matrix to be moved to a mapped matrix using list dma the converse can only be done by a series of calls to u functions.
7. As implemented on the Cell/B.E. the functions run most efficiently if the mapped and unmapped memory have the same 16 byte alignment. Otherwise an additional copy between SPE memory and a temporary buffer is performed to allow aligned transfers between the temporary buffer and unmapped memory.
8. On the Cell/B.E. list DMA is only used if the column lengths are a multiple of 16 bytes.

Examples

The following examples show how the `cw` and `nw` functions can be used. In the example assume that unmapped has been initialized to an unmapped pointer value of size `N*2*sizeof(float)`. The following data is declared for all the examples.

```
unsigned long unmapped;
float in1[N];
float in2[N];

float out1[N];
float out2[N];

int wh1;
int wh2;
```

In the first example, we kick off two transfers and wait for both to complete.

```
wh1 = e_cw();
e_putu_nw(in1,unmapped,N*sizeof(float),wh1);
e_putu_nw(in2,unmapped+N*sizeof(float),N*sizeof(float),wh1);
e_wait(wh1);
```

The second example is the same as above, but we use the `e_putu_cw` function to generate the wait handle.

```
wh1 = e_putu_cw(in1,unmapped,N*sizeof(float));
e_putu_nw(in2,unmapped+N*sizeof(float),N*sizeof(float),wh1);
e_wait(wh1);
```

In the final example, we use double buffering to overlap processing of unmapped memory with fetching unmapped memory.

```
/* initialization get two buffers */
wh1 = e_getu_cw(out1,unmapped,N*sizeof(float));
wh2 = e_getu_cw(out2,unmapped+N*sizeof(float),N*sizeof(float));
/* process next N-2 buffers */
for (i=0; i<N-2; i+=2) {
  e_wait(wh1);
  ... process out1 ...
  wh1 = e_getu_cw(out1,unmapped,N*sizeof(float));
  e_wait(wh2);
  ... process out2 ...
  wh2 = e_getu_cw(out2,unmapped+N*sizeof(float),N*sizeof(float));
}
/* process final 2 buffers */
e_wait(wh1);
... process out1 ...
e_wait(wh2);
... process out2 ...
```

Data Flow Parameter Functions

Any expressions that can be parsed by in the Gedae symbolic expression language can be used to express data flow parameters and dimensions in the Input, Output and Local sections of a primitive. These expressions are based on C syntax with a few functional extensions. This section presents some of the more important functions that can be used to describe dataflow.

part

Synopsis

```
int part(int f, int F, int N);
```

Description

The function call part(f,F,N) returns the value (f+1)*N/F – f*N/F; This function has the property that it divides N into as nearly equal as possible values that sum to N. For example if F = 8and N is equal to 35 then the values of part(f,F,N) take on the values 4,4,5,4,4,5,4,5 as f takes on the values 0,1,2,3,4,5,6,7.

The part function is available both to the dataflow expression parser and in the primitives Apply method. Note however that summing part(i,F,N) from i = 0..f yields f*N/F. This expression can be used in the Apply method of a primitive to calculate the offset of the fth part of N.

Example

```
Name: mzt_rpart
Type: static

Input: {
  stream float in_re[Rt:R][Ct:C];
  stream float in_im[Rt:R][Ct:C];
}

Output: {
  inplace pointer stream float
[f:F]out_re[part(f,F,Rt):R][Ct:C] = in_re;
  inplace pointer stream float
[f:F]out_im[part(f,F,Rt):R][Ct:C] = in_im;
}

Apply: {
  int f;
  for (f=0; f<F; f++) {
    int Roff_C = (f*Rt/F)*C;
    forward(out_re[f],in_re,in_re+Roff_C,1);
    forward(out_im[f],in_im,in_im+Roff_C,1);
  }
}
```

sum

Synopsis

int sum(int *X, int N);

Description

The function **sum** returns the sum of the vector elements **X[i]** as i ranges from **0** to **N-1**. For example **if X[] = {1,3,9,12,4}** then **sum(X,3) = 13** and **sum(X,5) = 29**.

Limitation

The function sum may only be used in the **Input**, **Output** and **Local** sections of the primitive. It is not available in the primitives **Apply** method.

Example

In the following example the sum function is used to indicate that the output row tile size is the sum of the family of input row tiles sizes.

```
Name: mt_rnconcat
Type: static
Input: {
  pointer stream float [f:F]in[Rs[f]:R][Cs:C];
}
Output: {
  inplace stream float out[sum(Rs,F):R][Cs:C] = in;
}

Apply: {
  int f;
  int Rsum = 0;
  for (f=0; f<F; f++) {
    set_ptr(in[f],out+Rsum*C);
    Rsum += Rs[f];
  }
}
```