



Gedae Debugging User's Manual

June 25, 2008

Address: Blue Horizon Development Software, Inc.
18000 Horizon Way, Suite 200
Mt Laurel, NJ 08054
Telephone: (856) 231-4458
FAX: (856) 231-1403
Internet: www.gedae.com

Table of Contents

| | | |
|---|---|----|
| 1 | Introduction..... | 4 |
| 2 | Probes..... | 5 |
| 3 | Break Points..... | 9 |
| | Break Point Files..... | 9 |
| | Example Break Point File..... | 11 |
| | Using Break Points to Create Automated Tests..... | 12 |
| 4 | Function embBreak..... | 16 |
| 5 | Ability to Dump Memory from Memory Display Structure..... | 17 |

1 Introduction

This is an evolving document that describes various debugging features in Gedae. For each release the document will include any feature that is new with that release. We will continue to update the document with existing features with each new release.

2 Probes

A user can specify various data streams in a graph that should have circular buffers attached that can record data. Such buffers are referred to as **probes**. The user can dump the probes to a file or (when using break points) compare the probes to existing probe files as a means of creating automated tests. Dumping probes can be initiated either through the Gedae user interface or in response to breakpoints set by the user that request the probes to be dumped. Comparing to existing probe files can only be initiated using break points and will be described in the section on setting break points. A user specifies the probes that should be active for a graph by creating a probe file.

A probe description file contains lines of the form:

```
<typename> <identifier> <value>
```

Where:

<typename> is the type of the probe and can be one of the following:

- `schedule` - create buffers for all data ports in a static schedule thread of a size to record <value> executions of the schedule
- `box-exec` - create buffers for all inputs and outputs of box of size tokens produced by <value> executions of box
- `box-fire` - create buffers for all inputs and outputs of box of size tokens produced by <value> firings of box
- `data-exec` - create a buffer for data of size tokens produced by <value> executions of parent box
- `data-fire` - create a buffer for data of size tokens produced by <value> firings of parent box
- `data-tokens` - create a buffer for data of size <value> tokens

<identifier> - For types `box-exec` and `box-fire` the identifier is the name of a primitive box element. For type `schedule` the identifier is the name of a primitive box element in the static schedule. For types `data-exec`, `data-fire` and `data-token` the identifier is the name of the data element connected to the stream.

<value> - an integer value that determines the size of the probe in bytes as described in the <typename> section.

Currently probe files must be created by the user outside of gedae using a text editor. An example probe file for the `demo/comm./e_comm` graph is:

```
box-exec modulator.x_osc 10  
box-fire e_channel.add 10240
```

```
schedule char_encode.unpack 1024
data-tokens char_decode.diff_decode>out 1024
```

The probe file should be stored in the `FGlibraries/probe/<graphname>` directory where `<graphname>` is the path to the graph in the `FGlibraries/boxes` directory. For example for the `demo/comm./e_comm` graph the probe file should be saved in the directory `FGlibraries/probes/demo/comm./e_comm`.

A user must load graph probes prior to graph compilation so that the Gedae graph compiler can automatically insert probe primitives in the graph to record the circular buffers of data described by the probe file. Probes may be loaded either through the gedae command line argument `-pr <filename>` or using the Flow Graph Editor menu item `Application->Open Probe Points`. For example if the above probe file were saved in `FGlibraries/probes/demo/comm./e_comm/default` the user could load probes from the command line as:

```
gedae -file demo/comm./e_comm ... -probes default ...
```

Or the user can select `Application->Open Probe Points` from the toplevel editor menu. When this menu is selected a dialog listing the probe description files for a graph pops up and allows the user to select the probe file to use.

A user can dump the probes using the `Application->Dump Probe Points` menu. When this is done the user is prompted for the name of a directory to which the probes are to be dumped. The directory path entered is relative to where the user started the Gedae development environment. A successful dump of the probes will create a dictionary file describing the various probe files and one probe file for each probe in the graph. For example dumping the probes described for the `demo/comm./e_comm` graph produces a dictionary file:

```
Probe Box : Data Element Name
  probe_1: e_comm.jabber>out
  probe_2: e_comm.char_decode.diff_decode>out
  probe_3: e_comm.e_channel.add>out
  probe_4: e_comm.e_channel.norm_noise>out
d_probe_1: e_comm.modulator.oqpsk_mod>out
  probe_5: e_comm.modulator.x_split>re
  probe_6: e_comm.modulator.x_osc>out
  probe_7: e_comm.char_encode.unpack>out
  probe_8: e_comm.char_encode.addstart_stopbits>out
  probe_9: e_comm.char_encode.diff_encode>out
```

Thus `probe_3` contains data from the `e_comm.e_channel.add>out` stream output.

The probe files contain information describing the probe data which is followed by a binary dump of the probe data. For example, probe_3 begins:

```
base_type      = 4
ndims          = 0
typesize      = 4
tokens_in_buf  = 10240
tokens_in_stream = 193536
```

```
□À□m□ÀQ→á¾Ý□@?5□!...
```

Where

- `base_type` - an enumerated type that indicates the basic C data type that was collected. Base types are:
 - 0 – token type not know
 - 1 – character token type
 - 2 – short token type
 - 3 – int token type (32 bit integer)
 - 4 – float token type
 - 5 – double token type
- `ndims` – the number of dimensions of the token. Scalars have 0 dimensions, vectors have 1 dimension and matrices have 2. `ndims` has a maximum value of 4.
- `dim1` – the first dimension (if any)
- `dim2` – the second dimension (if any)
- `dim3` – the third dimension (if any)
- `dim4` – the fourth dimension (if any)
- `typesize` – the number of bytes in a token element. Not that for single precision complex data this value will be 8 even though the base type is 4 (float) and the size of a float is 4 bytes. Thus the `typesize` is not redundant with the `base_type`.
- `tokens_in_buf` – the number of tokens that are recorded in the following binary dump. Note that the number of bytes in the dump will be the `typesize` times the product of the dims followed by the number of tokens in the buffer.
- `tokens_in_stream` – this is the last token recorded in the buffer. Since the buffer is of finite size and implemented as a circular buffer the `tokens_in_stream` and `tokens_in_buf` together describe which tokens are recorded. For example in the above example the tokens recorded are - given token numbering starts at 0 - $(193536-10240, 193536-1) = (183296, 193535)$. The information about the tokens actually stored in the buffer is useful when comparing probes. Only the token values that overlap are compared.

Following this header information is the binary dump of the data.

Probes are implemented as follows. The probes are added before the graph is compiled. Currently probes cannot be added interactively to running applications. The probes are added as primitives to the graph and the primitive has a local pointer "char buf[N]" where

N is large enough to store the number of tokens you specified for the probe. It also contains state information about the probe that indicates how much data is in the probe and which tokens are currently in the probe. The buffer buf is a circular buffer that the probe primitive updates every time the probe is executed along with the probe state variables. When a request to dump the probes occurs the probe data and probe state variables are retrieved from the running application by the host. This information is interpreted by the host and written out to the probe file if the data is to be dumped or the information is used to compare the currently collected data to existing probe files if the data is to be compared.

3 Break Points

Break Point Files

A user load a breakpoint description file into Gedae to break on primitive and queue events. Currently break point files must be created by the user. If the graph being tested is located in:

FGlibraries/boxes/<pathname>

(for example, FGlibraries/boxes/demo/comm/e_comm)

Then the breakpoint files must be found in the FGlibraries directory (\$FPATH on Windows):

FGlibraries/breakpoints/<pathname>

(for example, FGlibraies/breakpoints/demo/comm./e_comm/bkpts)

Each line of the breakpoint file describes a breakpoint. Primitive breakpoints are described as:

```
box=<boxname> [type=<event type>] [fired=<range specifier>]
[exec=<range_specfier>] [granularity=<range_specfier>] [stop-all] [exit] [dump =
<dirname>] [no-stop]
```

Where

- <boxname> is the hierarchical name of the box element in the graph (not including the top-level graph name). Only boxes that are executable may be used. Boxes such as vx_x and s_v that are not included in the end application may not be used to set breakpoints.
- type is the type of a box event
- <event_type> is one of
 - begin – event occurring immediately before a primitive executes
 - apply – event occurring after a primitive executes its Apply method
 - reset – event occurring after a primitive executes its Reset method
 - eos – event occurring after a primitive executes its EndOfSegment method
- fired is the total number of times the box has fired (accumulation of granularity for all executions of the box)
- exec is the total number of times the box's Apply method has been called
- granularity is the granularity at which the box is run
- stop-all indicates that all partitions should be stopped when the breakpoint is detected.

- exit indicates that gedae should exit after the breakpoint is detected and all other breakpoint commands have been executed. This is usually done in conjunction with the dump command
- dump = <dirname> indicates that probes should be dumped to the directory: FGLibraries/dump_probes/<graph_name>/<dirname>. If compare is enabled (see below) the probes are instead compared with the probes dumped to this directory.
- no-stop – do not stop execution when the breakpoint is encountered. Typically used in conjunction with the dump command to allow dumping probes when a breakpoint is encountered and then continuing execution.

We define the <range_specifier> by example:

| fired=<range_specifier> | interpretation |
|--------------------------------------|-----------------------|
| fired=10 | fired=10 |
| fired=(100,200) | 100<fired<200 |
| fired=[100,200) | 100<=fired<200 |
| fired=(100,200] | 100<fired<=200 |
| fired=[100,200] | 100<=fired<=200 |
| fired=(100,...) | 100<fired |
| fired=(...,100) | fired<100 |
| fired=[100,...) | 100<=fired |
| fired=(...,100] | fired<=100 |

Breakpoints can be set on user events as:

box=<boxname> type=user [number=<range_specifier>] [value=<range_specifier>]

Where

- number is the number passed as the first parameter to embUserEvent, embEndUserEvent, embUserIntEvent or embUserFloatEvent
- value is the value passed as the second parameter to embUserIntEvent or embUserFloatEvent

The second major breakpoint type is the queue breakpoint. A queue breakpoint has the form:

queue=<queuename> [type=<event_type>] [tokens=<range_specifier>]
[total_produced=<range_specifier>] [total_consumed=<range_specifier>]

Where

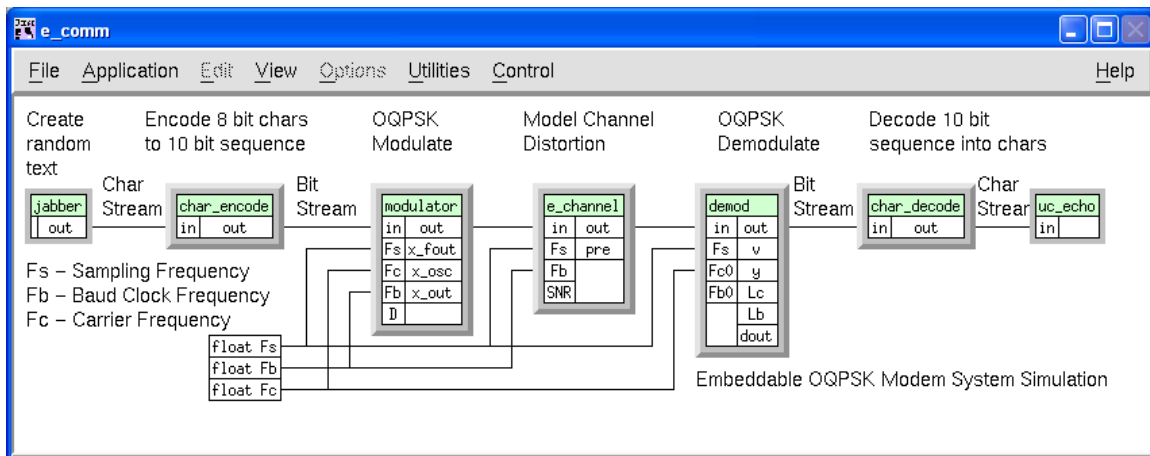
- tokens are the number of valid tokens in the queue
- total_produced are the total number of tokens that have been produced into the queue since the application started
- total_consumed are the total number of tokens that have been consumed from the queue since the application started

Example Break Point File

An example breakpoint description file for the graph demo/comm./e_comm is shown below:

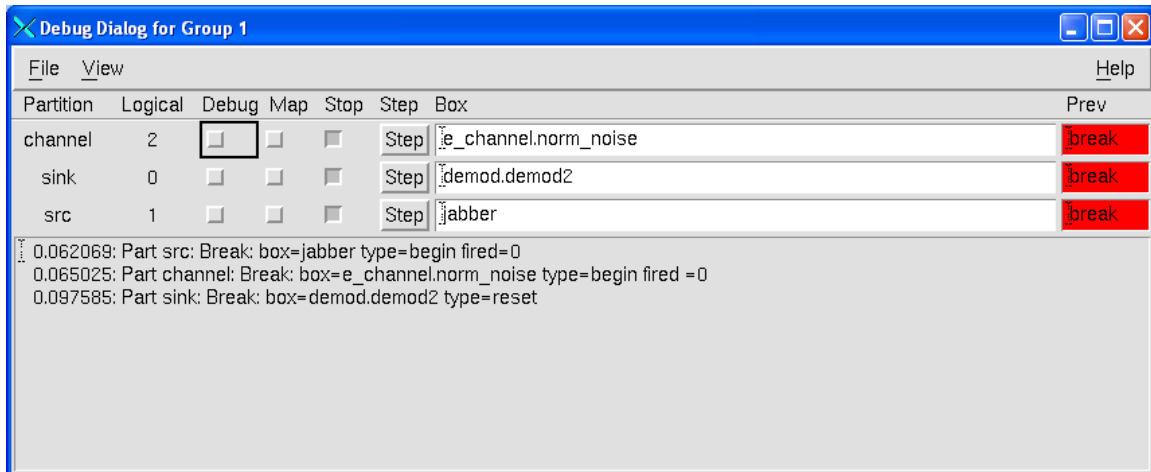
```
box=jabber type=begin fired=0
box=demod.demod2 type=reset
box=e_channel.norm_noise type=begin fired =0
box=modulator.fft_filter type=begin exec=[10,15]
box=e_channel.norm_noise type=apply fired=[40000,50000]
box=demod.decide type=apply fired=[10000,10020]
queue=modulator.fft_filter<in type=produce total_produced=[100000,150000]
queue=modulator.fft_filter<in type=consume total_consumed=[150000,200000]
queue=demod.decide<in type=consume total_consumed=[13000,13020]
box=uc_echo type=begin fired=[10000,10500] granularity=[1,50]
box=uc_echo type=begin fired=[10000,10500] granularity=[51,100]
box=uc_echo type=begin fired=[10000,10500] granularity=[101,150]
box=uc_echo type=begin fired=[10000,10500] granularity=[151,200]
box=uc_echo type=begin fired=[10000,10500] granularity=[201,256]
```

The e_comm graph is illustrated below



Note: Breakpoints only take effect when event tracing is enabled.

Running the e_comm demo with the above breakpoint file, with Trace enabled, and using a partitioning and mapping that maps the graph to three processors causes the breakpoint dialog to pop up as below:

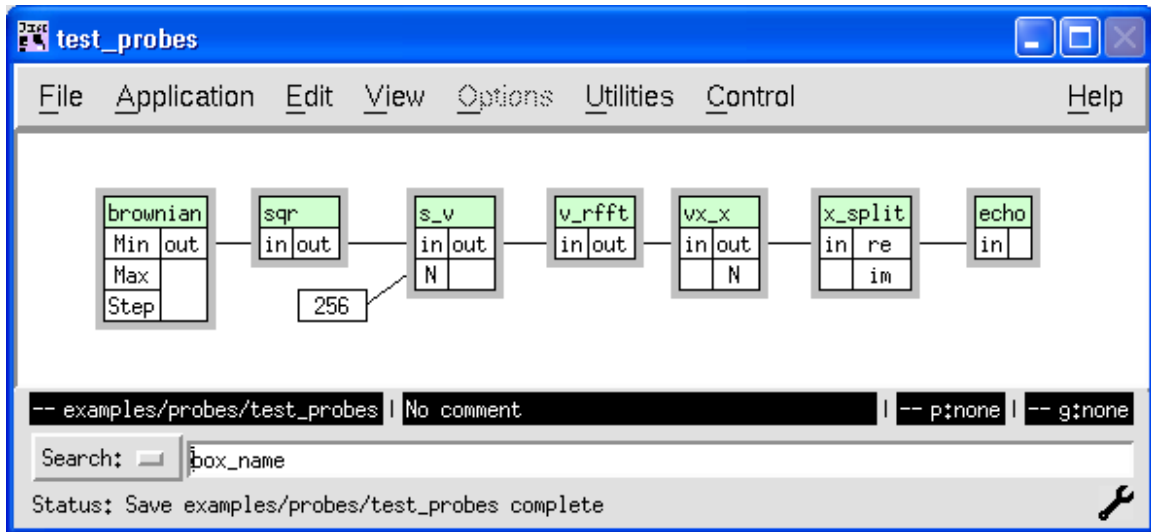


At this point all three partitions have stopped at different breakpoints, and the breakpoints where they stopped are listed in the text area at the bottom of the dialog. A user can now start any partition by unselecting the Stop toggle for that partition. A user can single step through the primitives in the partition, and a user can bring up the Structure display for the primitives in the graph to examine input and output data values.

Using Break Points to Create Automated Tests

One of the most useful features of breakpoints is the ability to dump or compare probes and then exit. This feature allows test scripts to be created that test a series of graphs against known past executions. This section will describe setting up a breakpoint and probe file to test a simple graph and describe how to use the file to dump the probes and use the same files to compare against existing probes. The comparison report that is generated for the probes will be described.

Typically a probe file and a breakpoint file will be designed in conjunction. The breakpoint file will contain a series of break points that dumps the probes described in the probe file. Each breakpoint prior to the last one will contain the no-stop command so execution will continue after the dump. The last breakpoint in the file should contain the exit command so the application will automatically exit. For example a break point file for the file `examples/probes/test_probes` seen below:



(note this file is not available in Gedae 5.5 but will be available in future releases – it is available to 5.5 users on request)

is

```
box=echo type=begin exec=1 dump=exec1 no-stop
box=echo type=begin exec=2 dump=exec2 no-stop
box=echo type=begin exec=3 dump=exec3 exit
```

This probe file will dump the input to the echo box prior to its first, second and third execution. Because the probes can be dumped at a known location in the data stream comparing against existing probe files should produce the same results if the graph behavior has not changed.

A probe file for this graph is:

```
box-exec echo 1
```

This creates a probe that has a buffer big enough to store the results of one execution of the echo primitive. Given that the breakpoint file is stored at `FGlibraries/breakpoints/examples/probes/test_probes/default` and the probes file is stored at `FGlibraries/probess/examples/probes/test_probes/default` a command line that will automatically dump the results of the above graph is:

```
gedae -file examples/probes/test_probes -probes default -bkpts default -tr -r
```

Running this graph creates three probe directories and then exits. Each directory contains a file `probe_1`. These files are found in:

```
FGlibraries/dump_probes/examples/probes/test_probes/exec1/probe_1
FGlibraries/dump_probes/examples/probes/test_probes/exec2/probe_1
```

FGlibraries/dump_probes/examples/probes/test_probes/exec3/probe_1

The header files of these three files contain the information:

```
probe_1:  
tokens_in_buf  = 128  
tokens_in_stream = 128
```

```
probe_2:  
tokens_in_buf  = 128  
tokens_in_stream = 256
```

```
probe_3:  
tokens_in_buf  = 128  
tokens_in_stream = 384
```

Indicating that the three probes were indeed collected prior to the first, second and third firing of the echo primitive.

A user can run the same application and compare against existing probe files by defining environment variable PROBE_COMP_REPORT. For example setting PROBE_COMP_REPORT to:

```
export PROBE_COMP_REPORT=compare.txt (in bash on linux/unix systems)
```

or

```
set PROBE_COMP_REPORT = compare.txt (on windows systems)
```

Then running the same gedae executable as above produces the following probe file:

```
-----  
c:\gedev\FGlibs/dump_probes/examples/probes/test_probes/exec1/probe_1:  
  Success.  
-----  
c:\gedev\FGlibs/dump_probes/examples/probes/test_probes/exec2/probe_1:  
  Success.  
-----  
c:\gedev\FGlibs/dump_probes/examples/probes/test_probes/exec3/probe_1:  
  Success.
```

However running the same graph with modified group settings to distribute the graph should yield the same results. In this way the user can test that different distributions of

the graph and mapping to different target processors do not change the results of execution.

Changing parameters on the Brownian primitive to deliberately introduce an error yields the resulting comparison file:

```
-----  
c:\gedev\FGlibs/dump_probes/examples/probes/test_probes/exec1/probe_1:  
  Buffer Error  
  byte 0 equals -128 in file buffer  
  byte 0 equals 62 in collected buffer
```

```
-----  
c:\gedev\FGlibs/dump_probes/examples/probes/test_probes/exec2/probe_1:  
  Buffer Error  
  byte 0 equals -116 in file buffer  
  byte 0 equals 92 in collected buffer
```

```
-----  
c:\gedev\FGlibs/dump_probes/examples/probes/test_probes/exec3/probe_1:  
  Buffer Error  
  byte 0 equals 0 in file buffer  
  byte 0 equals 20 in collected buffer
```

With the above capability a script to dump probes for many test graphs can easily be created and by setting the PROBE_COMP_REPORT environment variable the script can be rerun and a report of the success or failure of each graph can be created.

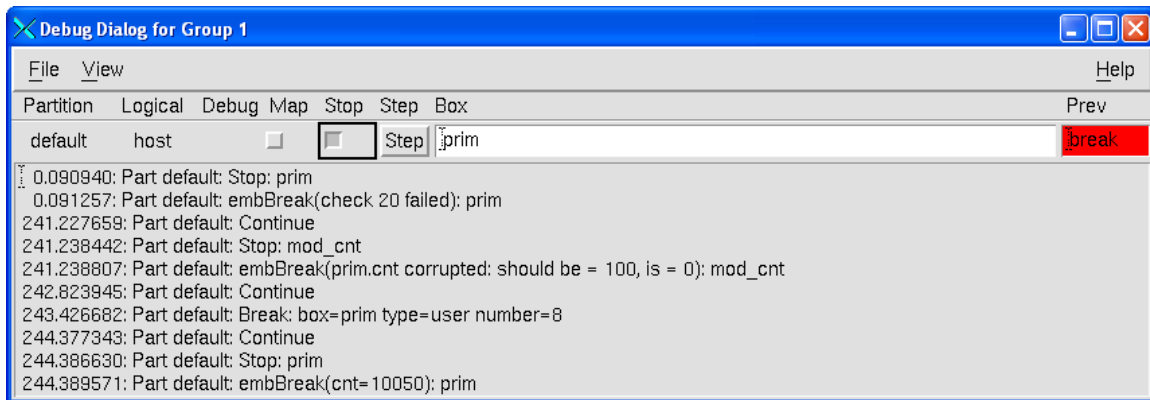
Limitation: Currently small floating point errors are reported as errors with no allowed tolerance on the results. More sophisticated error reporting will be provided in future releases.

4 Function embBreak

A user can force Gedae to invoke a breakpoint from a primitive by calling `embBreak`. For example, an `Apply` method can monitor a variable `cnt` and force a break as

```
Apply: {  
  ...  
  if (cnt == 10050) {  
    embBreak("cnt=10050");  
  }  
  cnt++;  
  ...  
}
```

Such programmed breakpoints can allow the user to force a breakpoint using more complicated conditional expressions than are available in the Gedae breakpoint file description language. It is anticipated that the `embBreak` function will be used for debug purposes only. When an `embBreak` point is hit, the debug dialog reports it as:



The above dialog shows several breakpoint occurrences. The bottom two lines show that primitive "prim" has called the function `embBreak` with the text string "cnt=10050".

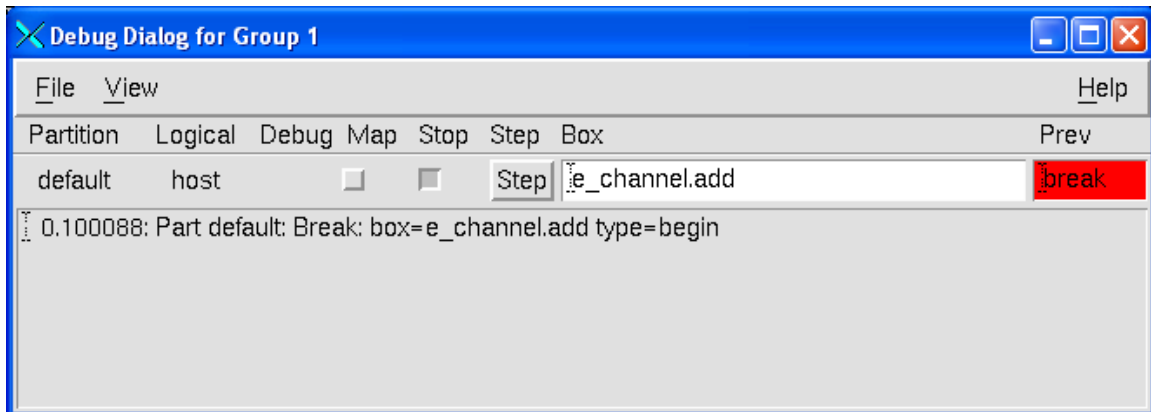
5 Ability to Dump Memory from Memory Display Structure

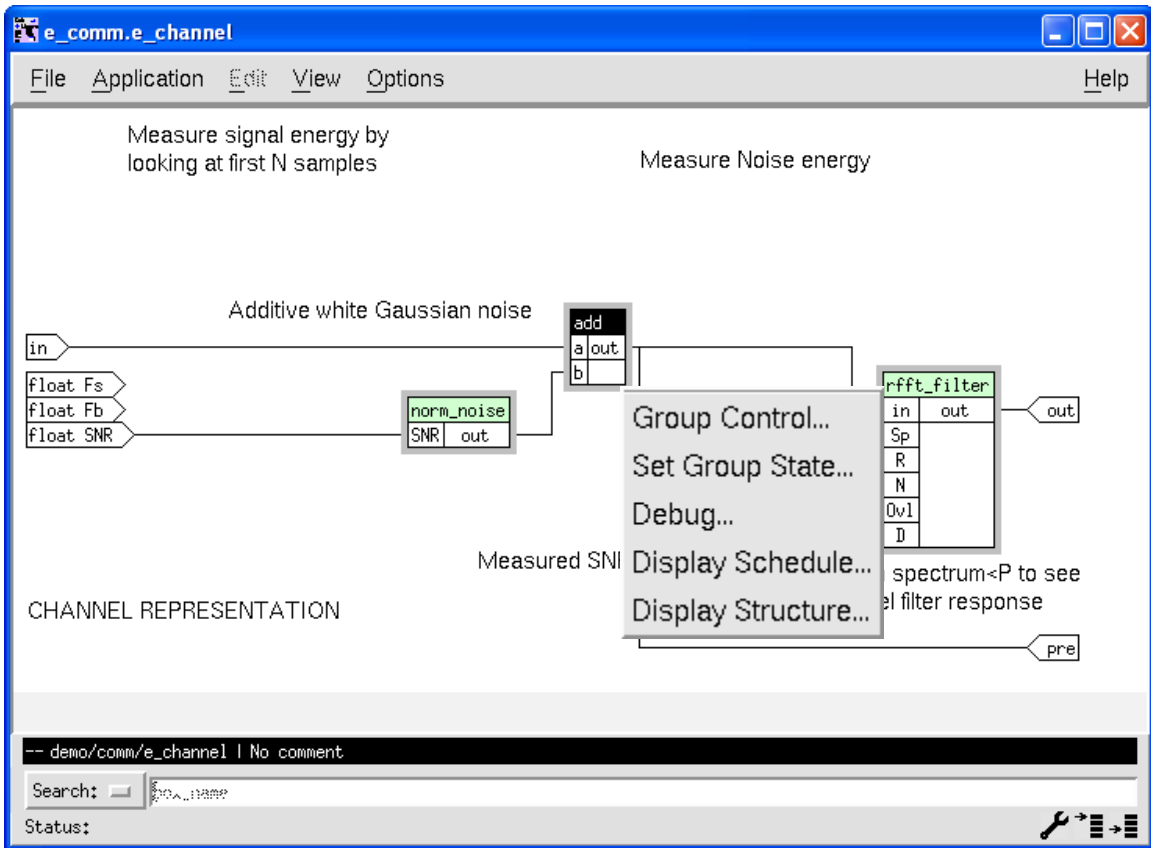
Limitations: This is a preliminary capability. We plan to use this capability as a starting point for being able to display or save any data values in the graph while the graph is running. Currently the dump capability may only be for primitives that are running within the development environment and can not be used on primitives that are mapped to target processors.

Primitive stream input and output values can now be dumped to a file from the Gedae structure display for that primitive. For example, if we run the demo/comm/e_comm graph using the following breakpoints file:

```
box=e_channel.add type=begin  
box=e_channel.add type=apply
```

Then when the graph is started (with trace enabled) the debug dialog pops up as:





Right clicking on the add box allows the user to select “Display Structure”, which brings up the display of the add boxes state structure:

Structure: e_comm.e_channel.add

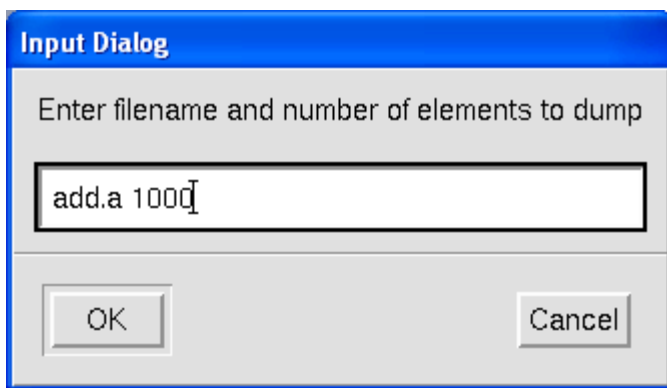
File Edit View Options

| Name | Value |
|-----------------------|------------------------|
| embeddable/stream/add | ... |
| int granularity | 5376 |
| float *a | 0x01809C18; -0.0243587 |
| int N_a | 5376 |
| float *b | 0x01804818; 0.0141357 |

This structure shows the value of input streams a and b before the box fires. We can right click on the a input and select the Dump To File item from the popup menu:

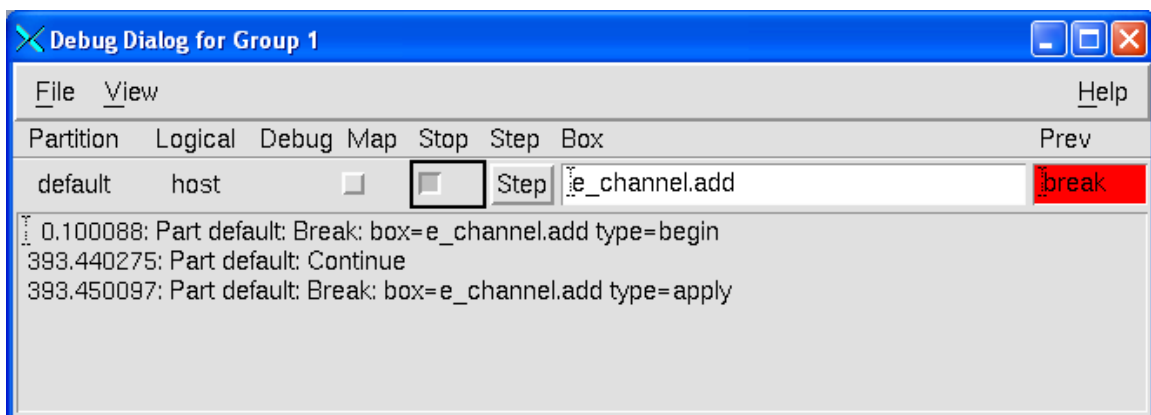
- Expand
- Collapse
- Hide
- Show All Boxes
- Expand Struct
- Expand Immediate
- Dump To File

We can then enter the name of the file that we want to dump the data to and the maximum number of values to be dumped:

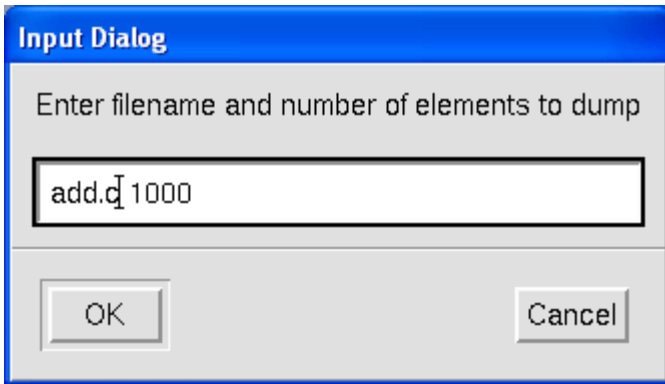


Similarly, we can dump 1000 values from the b input to file add.b

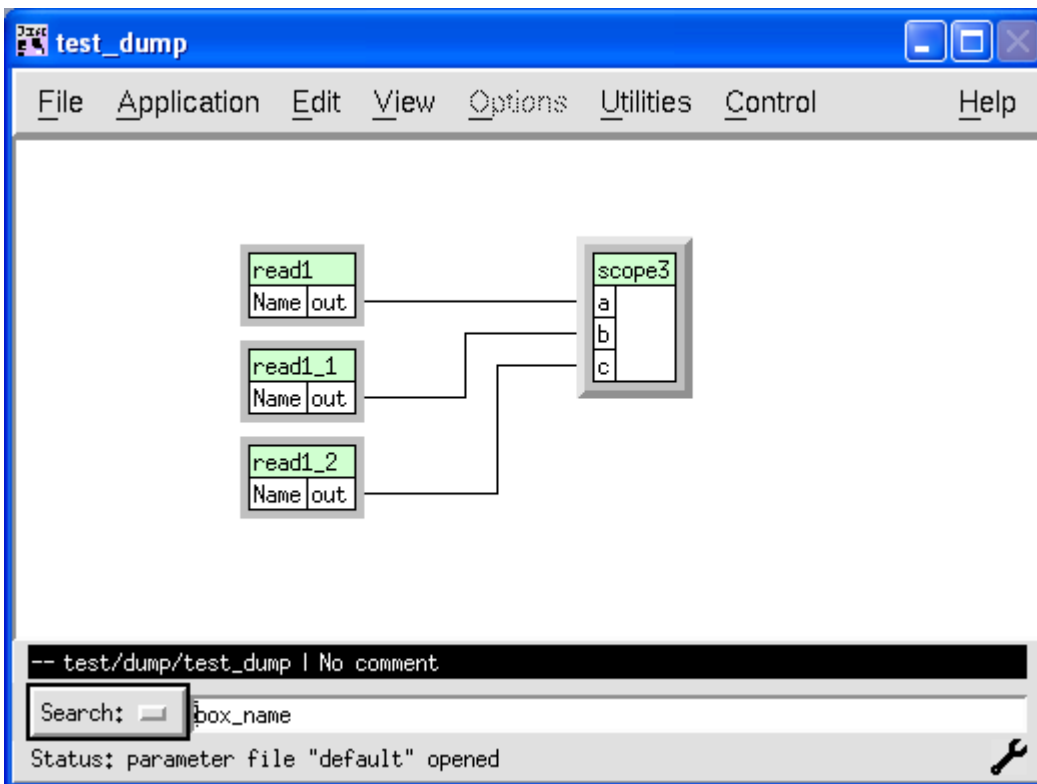
At this point, we hit the Stop toggle on the debug dialog and the graph proceeds to the next breakpoint.



This is the breakpoint that occurs after the box has fired. Since the a input is in place with the output the structure display now shows the output value of the add box in the a input. We select the a input and dump it to file add.c as:



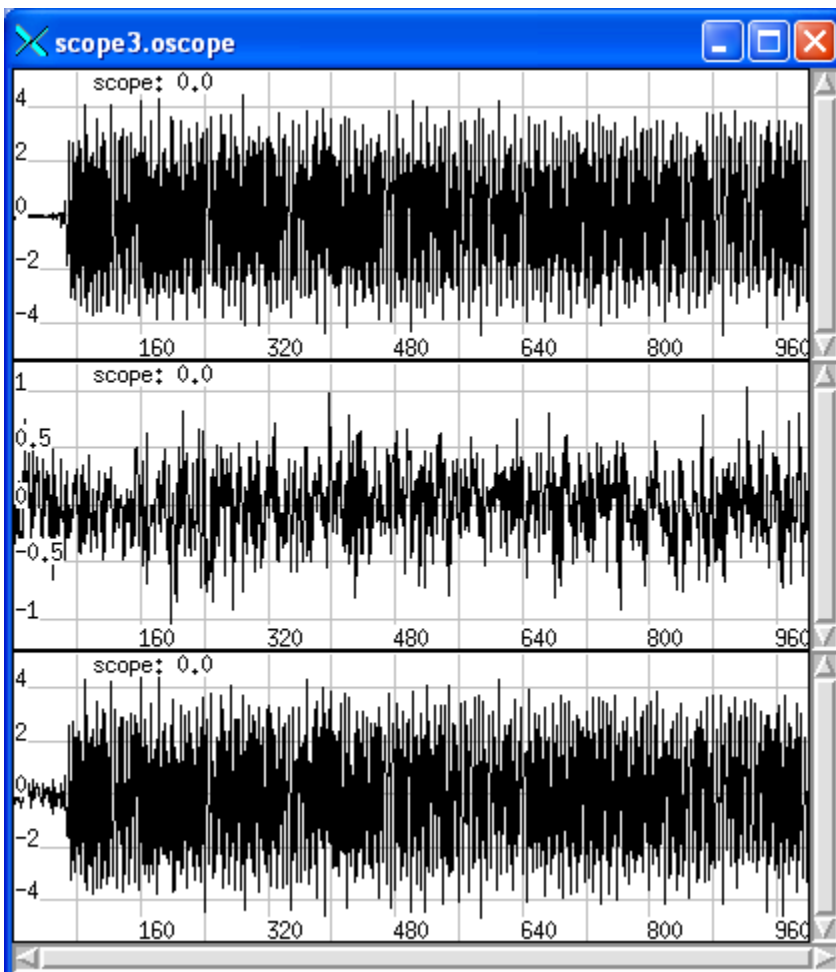
At this point, the input values to the add box should be stored in files add.a, add.b and the output value should be stored in add.c. Verify that the add box has executed correctly by creating the following graph:



With the parameters to the file read boxes set to read, the files add.a, add.b and add.c.

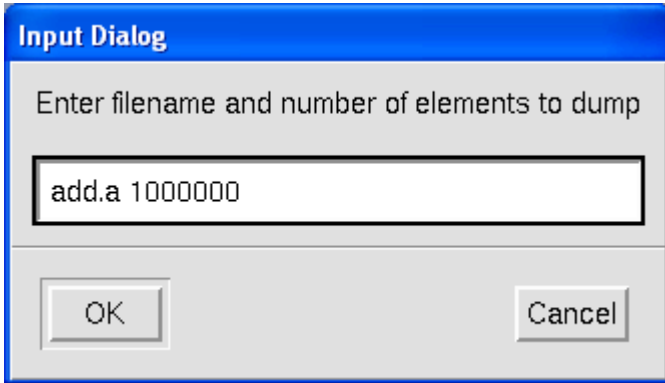
| Name | Value |
|--------------|-------|
| char Name[6] | add,a |
| read1 | |
| char Name[6] | add,b |
| read1_1 | |
| char Name[6] | add,c |
| read1_2 | |
| scope3 | |

Running this graph displays the results of adding input a to b to produce c:



Typically, the user can use the granularity of the primitive to determine how many values to dump. If more values than the amount that can be safely read from the memory

address are requested, then the user is informed of how many values were actually dumped. For example, requesting the values from the a input as:



The user is informed:

