



# Gedae Trace Table Users Manual

February 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](http://www.gedae.com)

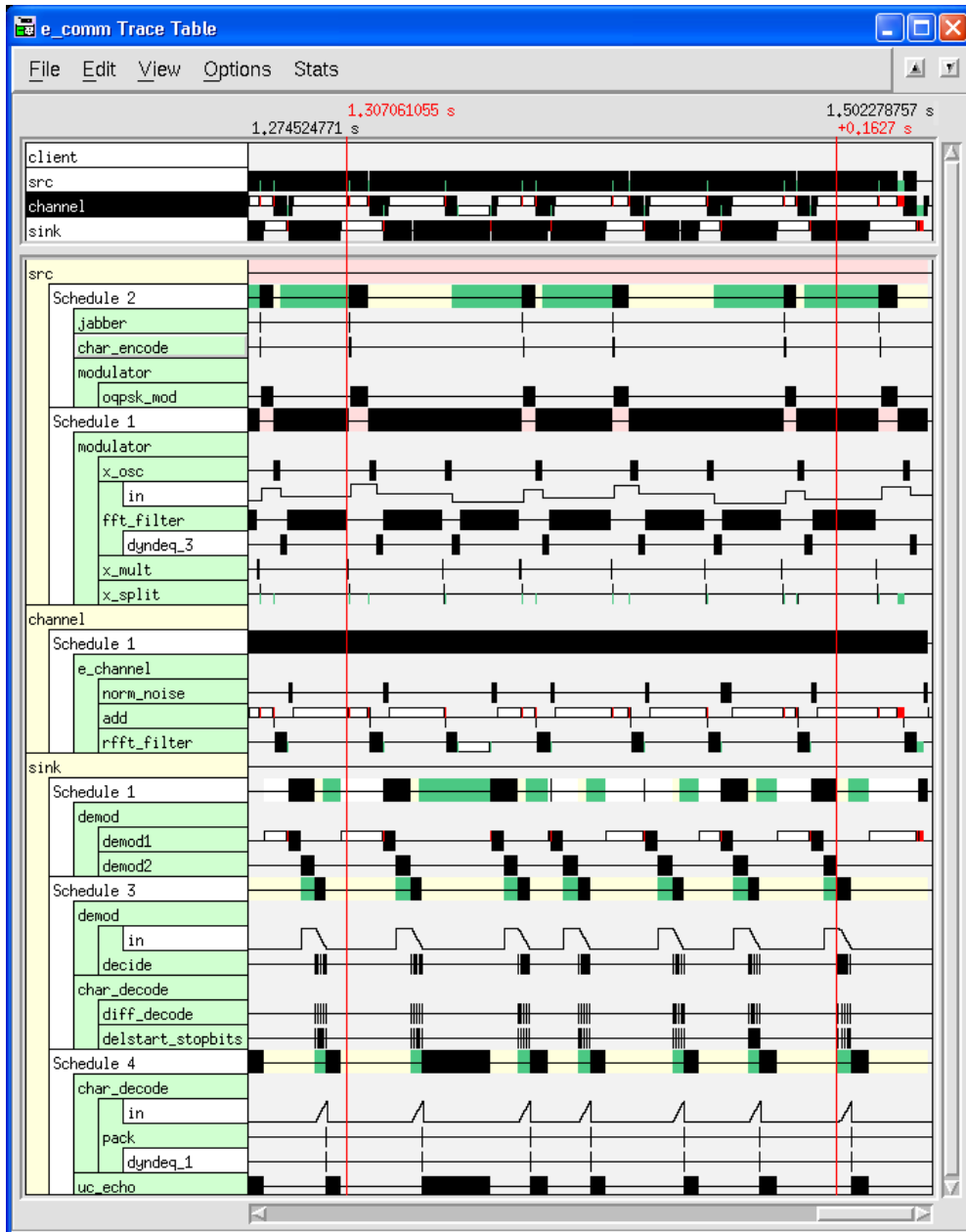
1	Preface.....	3
2	Introduction.....	4
3	Trace Event Filtering .....	6
4	Getting Trace Information from Non-Responsive Processors.....	10
	User Creation of Dump Files to View Trace and other Debug Information from a Target Executable that is not responding.....	10
	Automatic Creation of Dump Files to View Trace and other Debug Information from a Target Executable that is not responding.....	12
5	Unmapped Trace Memory .....	13
6	User Settable Data Trace Probes.....	14
7	Saving Trace Information to Comma Delineated File.....	18
	Saving the Trace Table Information .....	18
	Event File Format .....	19
	Index File Format.....	20

# 1 Preface

This document is an evolving document that presents various features of the Gedae Trace Table. While not complete at this time, we will add descriptions of new features as they are developed and also incrementally add descriptions of all the existing trace table features.

## 2 Introduction

The Gedae Trace Table allows users to view a timeline of the execution of an application graph. Events that are displayed include primitive execution including send and receive primitives, dynamic queues filling and emptying, change of static schedule thread state, segment boundaries added to queues and segment state of static schedule thread. Trace information is given on both a per processor basis and on a per primitive basis. A typical trace table is seen below:



In the above picture the top three lines represent a summary of the trace activity on the partitions named src, channel, and sink. The remainder of the trace table shows a more detailed breakdown of the activity on these three processors, including primitive firings (like the `fft_filter`), Schedule state changes (like `sink.Schedule1`) and queues firing an emptying (like the `queue demod.decide<in>`).

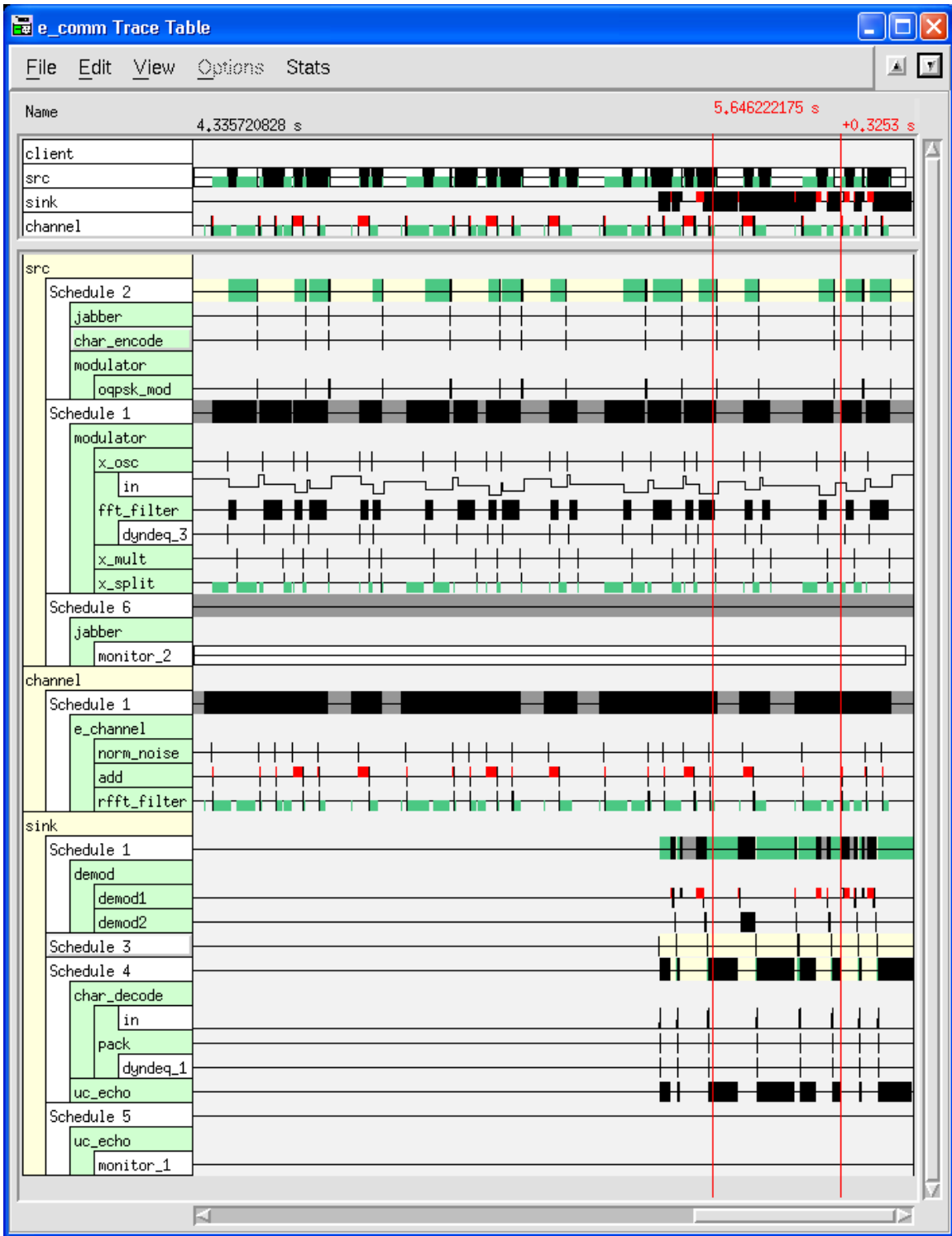
Using the trace table the user can determine why a graphs throughput is limited, get statistics on box firings, determine interprocessor communication delays, see primitives blocked waiting for I/O and analyze deadlock situations.

### 3 Trace Event Filtering

Trace event filtering reduces the amount of trace information that is collected allowing users to see longer execution time spans with smaller trace buffers. When the trace table is displayed the user will only see events that affect the dataflow behavior of the graph making it easier to debug dataflow problems. The user can turn on trace event filtering by starting Gedae with the `-tf` command line argument as for example:

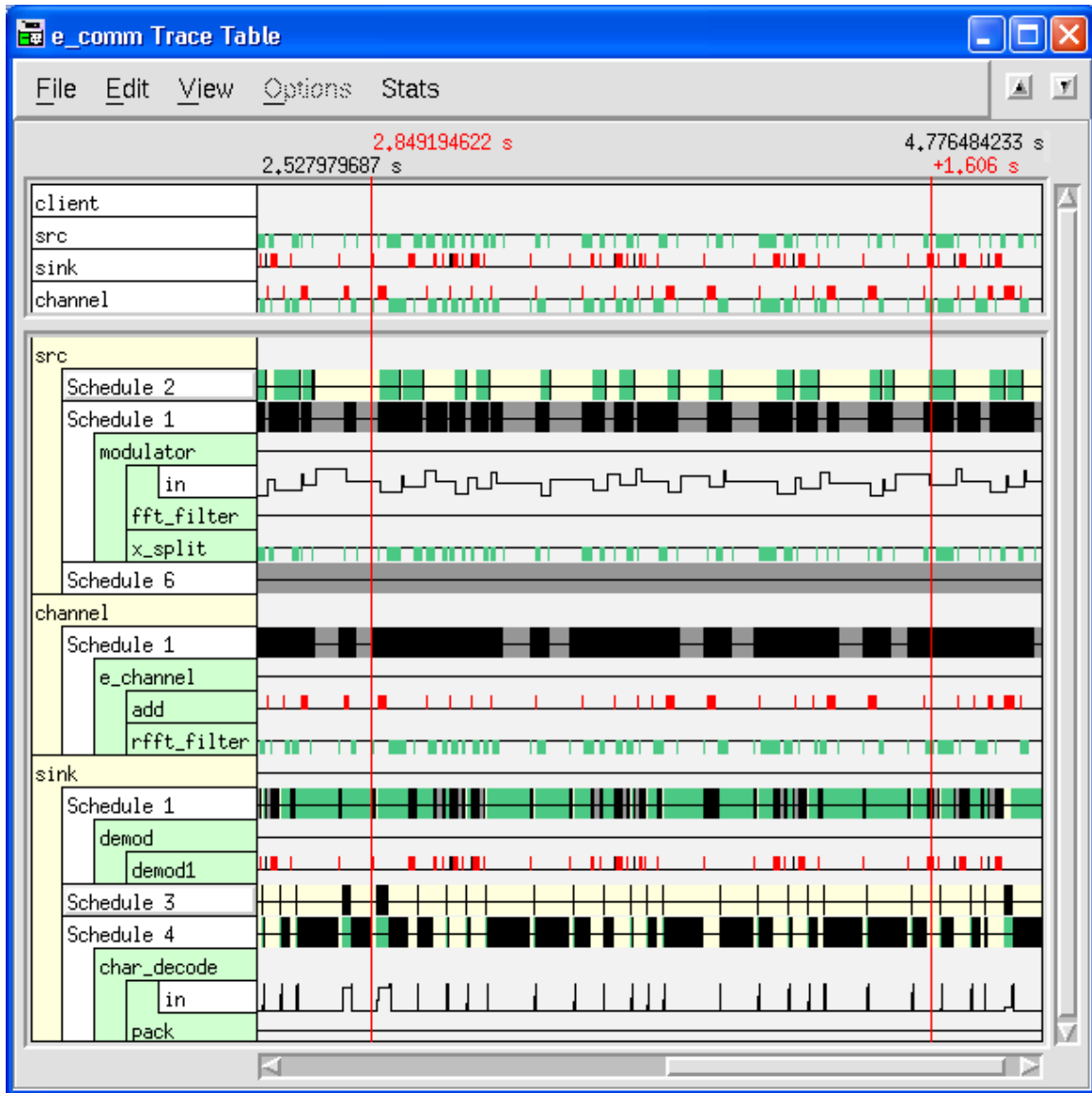
```
gedae -file demo/comm./e_comm -pa default -gr embedded -tr -tf -r
```

The events that are filtered out are the execution of primitives that cannot fail and thus add no information about the dataflow behavior of the application. Primitives like the automatically inserted send and receive primitives that can block are still recorded. And I/O primitives that can poll or pause waiting for an interrupt are also recorded. But boxes like the `vx_fft`, `add`, and `m_mult` that will fire as long as their input is available are not recorded. The two trace tables below show the advantage of using event filtering. The first table shows the `demo/comm./e_comm` graph running without event filtering:



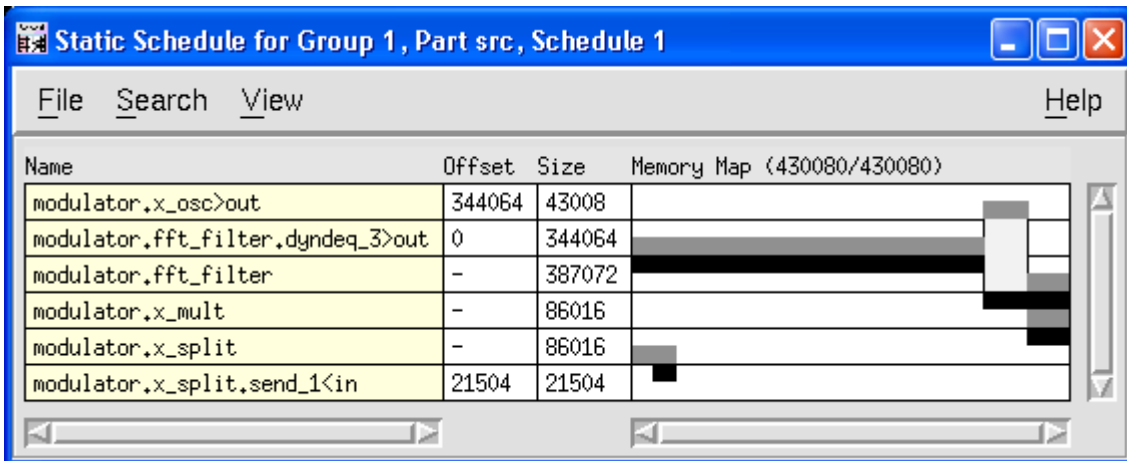
As can be seen the “sink” partition has overwrapped its trace buffer so all the events cannot be seen. Also the events such as the `fft_filter` that do not affect dataflow occupy a good deal of the vertical table space.

Below is the same trace table collected with the `-tf` parameter added to the command line:



As can be seen the sink partition can now display a full 2 seconds of trace information with the same size buffer that could only display about 0.6 seconds of trace information. Also the trace table is a good deal smaller and easier to analyze. The user can still see the order that boxes execute within a given schedule by clicking on the schedule (For example Schedule 1 above) and selection Options->Display Schedule from the Trace Table menu.



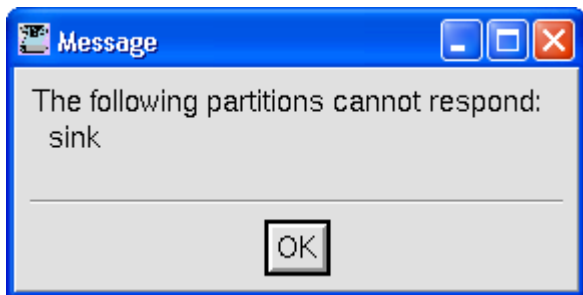


## 4 Getting Trace Information from Non-Responsive Processors

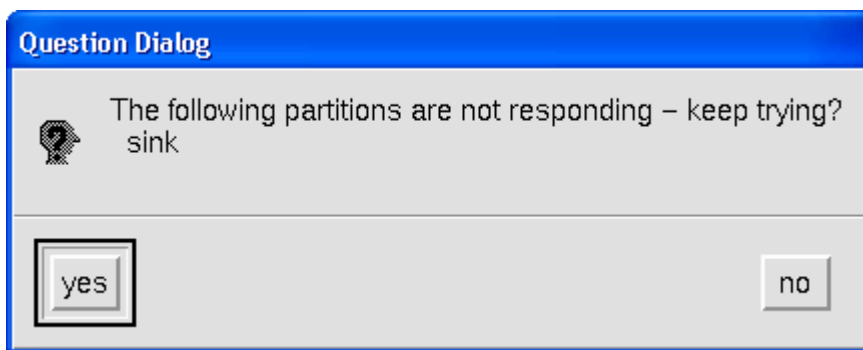
### User Creation of Dump Files to View Trace and other Debug Information from a Target Executable that is not responding.

Being able to view trace and other debug information from target executables that are not responding to commands from the development environment is possible if the target BSP allows it. This section describes how to view the trace information if the BSP requires the user to be involved in dumping memory from the target processor.

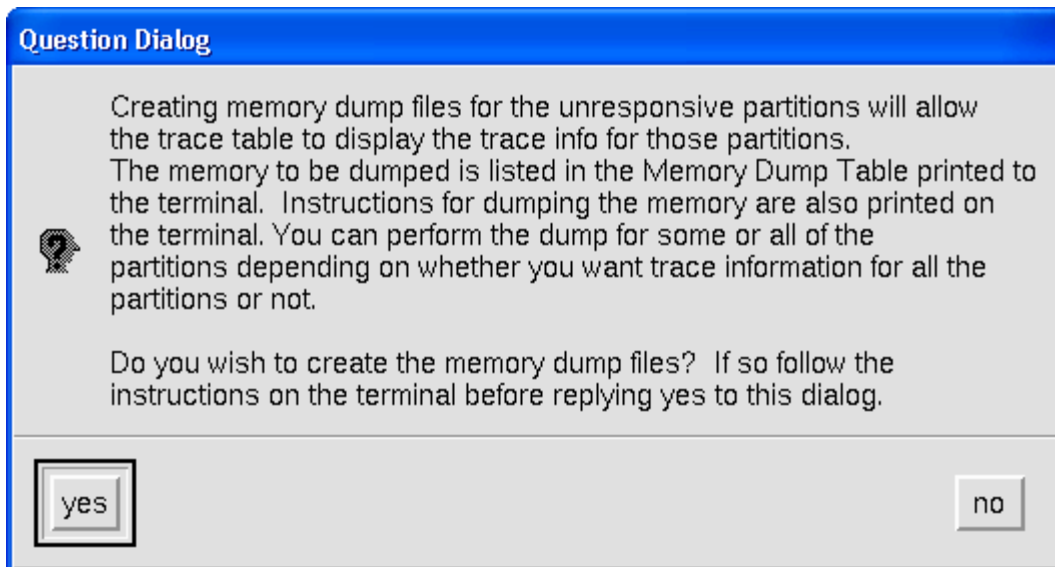
There are two ways that Gedae can detect that a partition is not responding. One way is that the Gedae Development Environment can no longer send a command to the target partition. In this case Gedae reports:



Another way is that the Gedae Development Environment does not receive a response from the target processor after a timeout period. In this case the user is asked to keep trying.



If the user is convinced that the non-responsive partition will not be able to respond and answers No to the above dialog then Gedae displays the following dialog asking the user to dump the trace information to a set of files.



The Memory Dump Table and Partition Info Tables printed to the terminal are:

Memory Dump Table			
Partition:	Base Address	Bytes	Filename
	sink: 0x00430048	200000	embedded/e_comm_1/DUMP_sink_1
	: 0x00460d90	6272	embedded/e_comm_1/DUMP_sink_2

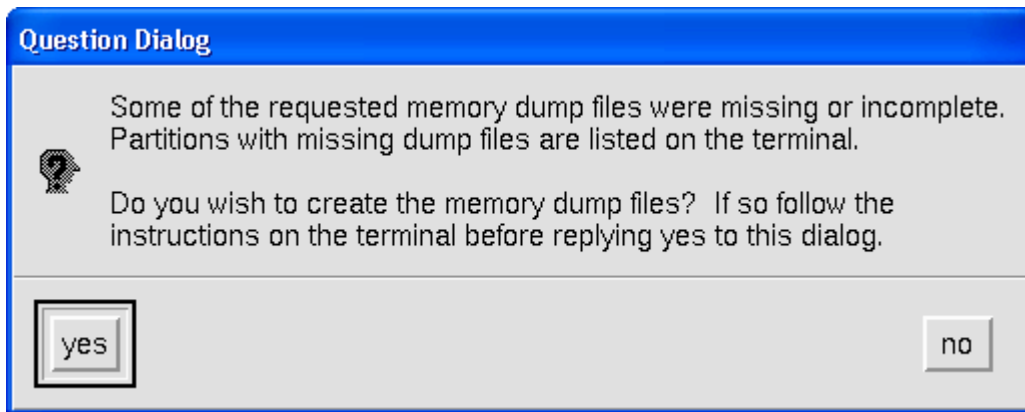
Partition Info Table		
Partition:	Physical Processor	Executable Pathname
sink:	local	embedded/e_comm_1\ent\exec-sink

To dump the memory requested in the Memory Dump Table, for each partition in the Partition Info Table dump the memory between <base\_addr> and <base\_addr>+<bytes> to <filename> where:

- <filename> is the Filename from the Memory Dump Table
- <base\_addr> is the Base Address from the Memory Dump Table
- <bytes> is the value of Bytes from the Memory Dump Table.

The above table tells the user to create two binary files. The first one should contain the 200000 bytes beginning at the sink processes base address 0x00430048. The second should contain 6272 bytes beginning at the sink processes base address 0x00460d90. Once these files are created according to the instructions printed to the terminal after the Memory Dump Table the user responds Yes to the above dialog and Gedae will then be

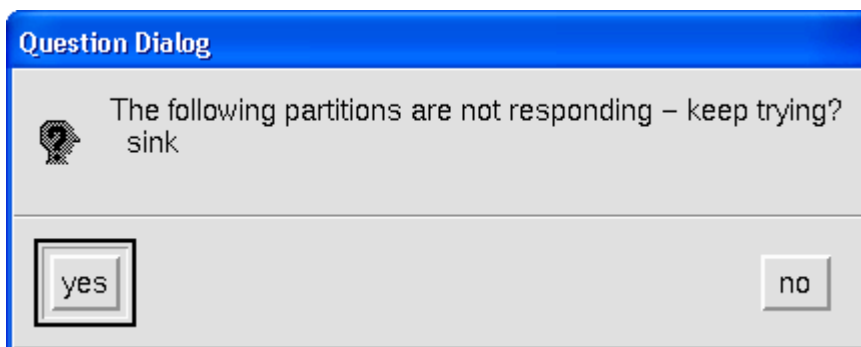
able to display the trace table. If the user responds Yes to the above dialog but the files were not created or were incorrectly created then the following dialog is displayed giving the user an additional chance to create the dump files.:



Replying No at any time will cause the Trace Table to be displayed with the memory partitions that were responding.

### **Automatic Creation of Dump Files to View Trace and other Debug Information from a Target Executable that is not responding.**

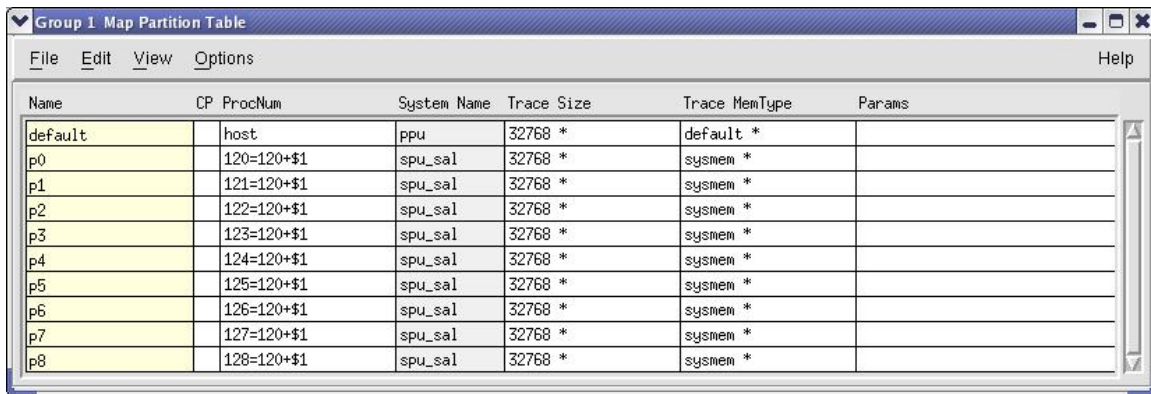
Some BSPs can create dump files of requested partition memory blocks without cooperation of the partitions target process. This feature supports getting the information needed to view the Trace Table of unresponsive partitions. When this feature is enabled if a partition is not responding as indicated by the dialog:



If the user answers no the memory needed to view the trace information will be extracted without cooperation of the partition process.

## 5 Unmapped Trace Memory

On the Cell B.E. processor each target processor (SPU) has only 256kbytes of memory. Since this small amount of memory must be used for program and data memory this often leaves little memory available for collecting trace events. To extend the trace memory a new feature has been added that allows the user to allocate the trace buffer in off chip system memory. To use this new feature the user needs to set the Map Partition Table field Trace MemType to an unmapped memory type. For example in the Map Partition Table viewed below the Cell SPU processors all have there memory types set to systemem which is the name given to the unmapped system memory for the Cell BSP. The Trace Size has been set to 32768 which requires 655360 bytes of memory – well over twice what would have been available in all of the SPU's on chip memory.



Name	CP ProcNum	System Name	Trace Size	Trace MemType	Params
default	host	ppu	32768 *	default *	
p0	120=120+\$1	spu_sal	32768 *	systemem *	
p1	121=120+\$1	spu_sal	32768 *	systemem *	
p2	122=120+\$1	spu_sal	32768 *	systemem *	
p3	123=120+\$1	spu_sal	32768 *	systemem *	
p4	124=120+\$1	spu_sal	32768 *	systemem *	
p5	125=120+\$1	spu_sal	32768 *	systemem *	
p6	126=120+\$1	spu_sal	32768 *	systemem *	
p7	127=120+\$1	spu_sal	32768 *	systemem *	
p8	128=120+\$1	spu_sal	32768 *	systemem *	

The unmapped memory trace table implementation is quite efficient. It uses a small double buffer of trace events on the target processor. When a buffer is filled a DMA transfer is kicked off to move the on chip trace events into system memory. This DMA transfer occurs rarely, is done in the background, and is of a relatively small amount of data compared with the amount of data transferred by the running application. As a result using unmapped memory has little impact on performance.

## 6 User Settable Data Trace Probes

A user can cause an application to be created that adds circular data trace buffers to any point in the application graph. These circular buffers keep a record of all data that passes through a static schedule thread, a primitive or a primitive input or output on the graph. While the graph is running the user can dump the data trace buffers for analysis with tools outside of Gedae.

To create data trace buffers the user must add a file to the <fgpath>/probes/<application path> directory that describes the probes where

<fgpath> is the path to the flow graph libraries directory. On Windows <fgpath> is the value given by the environment variable %fgpath%. On linux and solaris <fgpath> is typically the FGLibraries directory that is directly under the directory from which you run gedae.

<application path> is the path to the toplevel application under <fpath>/boxes. It is the path to the file that you open to load the toplevel graph.

For example if the graph that is running is demo/comm./e\_comm then a file default can be added to

<fpath>/probes/demo/comm./e\_comm/default

The probe point description file contains any number of lines of the form:

<probe type> <probe identifier> <N>

Where <probe type> is one of:

Proc Type	Description
schedule	create buffer for all data ports in schedule of size to record N executions of schedule
box-exec	create buffer for all inputs and outputs of box of size tokens produced by N executions of box
box-fire	create buffer for all inputs and outputs of box of size tokens produced by N firings of box
data-exec	create buffer for data of size tokens produced by N executions of parent box
data-fire	create buffer for data of size tokens produced by N firings of parent box
data-tokens	create buffer for data of size N tokens

The <probe\_identifier> identifies which schedule, box or data the probe or probes are to be created for. For the schedule type the identifier is the full application name of any

primitive in the schedule. For the two box types the identifier is the full application name of the primitive. And for the three data types the identifier is the full application name of the primitives data port.

The following is a probe file for the demo/comm./e\_comm application.

```
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 is loaded either using the `-probes gedae` argument list parameter as

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

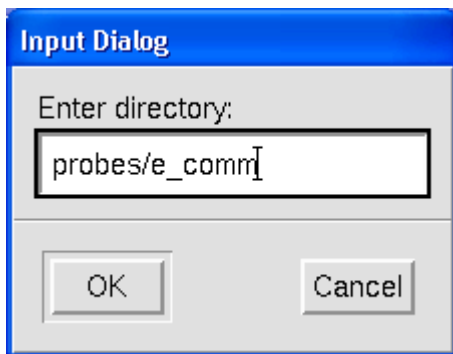
Or it is set using the toplevel application graph menu option:

Application->Open Probe Points

The probe points must be set before the application is compiled by the Gedae graph compiler. They can not be set once the application has been created and is running. To dump the probe points from a running application the user selects the menu item:

Application->Dump Probe Points

The user will be prompted for a directory in which the probe point dump files is to be created with the dialog:



The user should enter the directory that the probe files should be created in and the select OK. Gedae will create the director path if necessary and dump the probe files and a probe file dictionary to that directory. For example for the e\_comm demo using the example probe file shown above the following files are created:

```
> pwd
/cygdrive/c/gedev/user/nt/probes/e_comm
> ls -l
```

```

total 2581
-rwxr-xr-x    1 Kerry    None    2048116 Jan 14 16:09 d_probe_1
-rwxr-xr-x    1 Kerry    None      427 Jan 14 16:09 dictionary
-rwxr-xr-x    1 Kerry    None    1136 Jan 14 16:09 probe_1
-rwxr-xr-x    1 Kerry    None   41074 Jan 14 16:09 probe_2
-rwxr-xr-x    1 Kerry    None   41074 Jan 14 16:09 probe_3
-rwxr-xr-x    1 Kerry    None  430194 Jan 14 16:09 probe_4
-rwxr-xr-x    1 Kerry    None    8304 Jan 14 16:09 probe_5
-rwxr-xr-x    1 Kerry    None   10353 Jan 14 16:09 probe_6
-rwxr-xr-x    1 Kerry    None   10353 Jan 14 16:09 probe_7
-rwxr-xr-x    1 Kerry    None    1135 Jan 14 16:09 probe_8
-rwxr-xr-x    1 Kerry    None   41074 Jan 14 16:09 probe_9

```

The dictionary describes which probe file goes with which data element. For this graph the dictionary is:

```

probe_1: e_comm.char_decode.diff_decode>out
probe_2: e_comm.e_channel.add>out
probe_3: e_comm.e_channel.norm_noise>out
probe_4: e_comm.modulator.x_osc>out
probe_5: e_comm.char_encode.unpack>out
probe_6: e_comm.char_encode.addstart_stopbits>out
probe_7: e_comm.char_encode.diff_encode>out
d_probe_1: e_comm.modulator.oqpsk_mod>out
probe_8: e_comm.char_encode.unpack.recv_1>out
probe_9: e_comm.e_channel.add.recv_2>out

```

The probe files contain information in the following format

```

base_type      = <bt>
ndims          = <ndims>
dim1           = <dim1>
dim2           = <dim2>
dim3           = <dim3>
dim4           = <dim4>
typesize      = <typesize>
tokens_in_buf  = <tokens_in_buf>
tokens_in_stream = <tokens_in_stream>
<binary data>

```

Where

<bt> is the base type and is an integer value from 1 to 5 where 1 is a char, 2 is a short, 3 is an int, 4 is a float and 5 is a double.

<ndims> is the number of dimensions that the token type contains (0 for scalar token, 1 for vector, 2 for matrix, 3 for 3d matrix and 4 for 4d matrix)

<dim1>, <dim2>, <dim3> and <dim4> are the size of each of the 4 dimensions and are only printed out up to the number of dimensions of the token.

<typesize> is the number of bytes in the base type.

<tokens\_in\_buf> is the number of tokens recorded to the file



<tokens\_in\_stream> is the total number of tokens that have passed through this data stream. Only the last <tokens\_in\_buf> tokens that have passed through the stream will be recorded to the file.

<binary\_data> is the binary data that contains the values of the <tokens\_in\_buf> number of tokens. It should have the size <typesize> \* <tokens\_in\_buf> \* <dim1> \* <dim2> \* <dim3> \* <dim4> for 4 dimensional tokens and similarly for lower dimensional tokens removing the unused dimensions..

For example probe\_4 which is the complex x\_osc output has the preamble:

```
base_type      = 4
ndims          = 0
typesize      = 8
tokens_in_buf  = 53760
tokens_in_stream = 166656
```

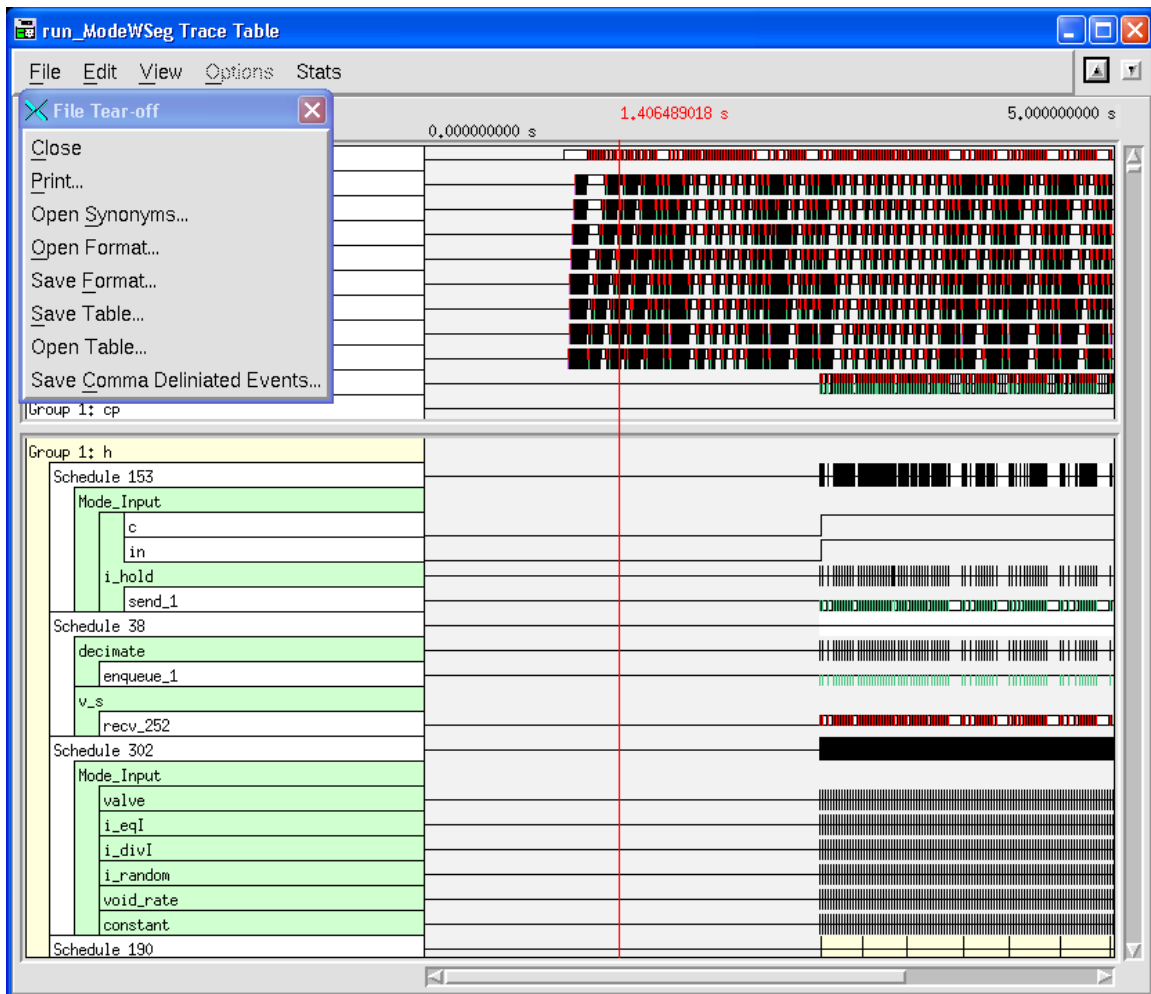
Note that though the base type is 4 (float) the typesize is 8 because each complex value consists of two floats. The preamble is followed by 8\*53760 bytes of data.

## 7 Saving Trace Information to Comma Delineated File

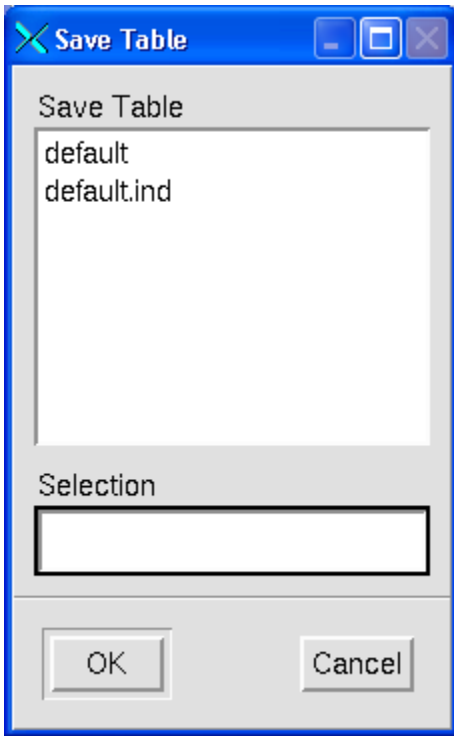
The trace table event information can be saved to a set of comma delineated files that completely describe the information in the trace table. These files are easily parsed and are can be used as inputs to third party offline analysis tools of the trace information. The comma delineated files can also be easily ready into Microsoft Excel.

### Saving the Trace Table Information

To save the trace table information the user must first bring up a Gedae Trace Table for the running application. The user then selects the trace table menu item File->Save Comma Delineated File.



The user is then prompted to supply the name of the file to be created



The user can either type a new filename or can select one of the filenames without the ind suffix. Two files will be created: the event file and the index file. The files are saved in the %fgpath%\cd\_events\

## Event File Format

Each line of the event file contains the following fields:

Field	Type	Description
group	int	the number of the Gedae group the event belongs to
part	string	the partition within the group the event belongs to
etype	string	the entity type – (box, queue, sched or peer)
index	int	the entities index – see the Index File
type	string	the event type
time1	double	time stamp at which event begins
time2	double	time stamp at which event ends (for interval events)
amount	int	granularity of box events words of queue event firings of peer events

acc	int	accumulated value of amount total number of times box fired for box events total number of tokens processed for queue event does not apply to peer events
segment	int	segment number of event
state	string	schedule state value
retry	int	true if schedule retry flag set
lock	int	true if schedule is locked
level	int	segmentation level of peer event
number	int	number of user event
value	int or double	value of user event

The first four fields uniquely identify the entity being traced. The entities name can be looked up using these four fields in the index file. Not all fields apply to all event types. Fields that do not apply are either indicated by commas with no data between or by an end-of-line occurring before the field is printed.

## Index File Format

The index file relates the indices in the event file to the named entities in the gedae application. The index file contains five comma separated columns which are:

Field	Type	Description
group	int	the number of the Gedae group the event belongs to
part	string	the partition within the group the event belongs to
etype	int	the entity type (box, sched, queue or peer)
index	int	the enties index
name	string	the full trace table name of the entity

The first four fields correspond to the first four fields in the event file. The trace table name is a period separated list of names that provides a hierarchical name for the event. Box, Queue and Peer event names contain the group and schedule names that the event belongs to. Using this information the events can be placed in the context of the static schedule thread to which the event belongs.