

01000111 01100101 01100100 01100001 01100101  
01000111 01100101 01100100 01100001 01100101  
01000111 01100101 01100100 01100001 01100101  
01000111 01100101 01100100 01100001 01100101

## Gedae 4.4 Release Notes

February 2004

Address: Gedae, Inc.  
18000 Horizon Way, Suite 200  
Mt Laurel, NJ 08054  
Telephone: (856) 231-4458  
FAX: (856) 231-1403  
Internet: [www.gedae.com](http://www.gedae.com)

# 1 New Features

Gedae 4.4 provides three related features targeted at allowing Gedae graphs to be interrupt driven rather than polling. Interrupt driven graphs improve dynamic scheduling performance, reduce power consumption, and provide better sharing of the CPU with other processes. Gedae 4.4 also includes enhancements to the Dy4av2 BSP that now supports the Starlink drivers and provides a DSA transfer mechanism for high-speed communication.

## Dy4av2 DSA Implemented

A new communication method – `dsa_bulk` – that uses the Dy4av2 bulk transfer mechanism was added to the Dy4av2 BSP. This method allows processing and data transfers to be done concurrently. Users can set the number of send and receive buffers for each `dsa_bulk` communication port allowing double, triple or higher level buffering on both the sender and receiver.

## Dy4av2 Starlink Drivers supported

The Dy4av2 BSP has been modified to run correctly with or without the Starlink drivers installed. Thus, both single board and multiple board configurations of the Dy4av2 are now possible. Appendix A shows an example of the Dy4av2 efficiently executing a graph on eight processors.

## Pause and Resume

Four new functions, `embPause`, `embResume`, `embSelf` and `embGetSchedule` have been added to the primitive box API to support interrupt driven primitive execution. The function `embPause` is called from the `Apply` method to signal when a primitive is inactive. A call to `embResume` from an interrupt handler or callback resumes execution of the primitive. Functions `embSelf` and `embGetSchedule` provide the information necessary for the `embResume` function to wake up the paused primitive.

### Synopsis:

```
void *embSelf();
Schedule embGetSchedule(void);
void embPause(void);
void embResume(void *thread, Schedule s);
```

## Description:

The `embPause/embResume` feature eliminates the need to poll on primitives that are waiting for a resource. Because such polling is time consuming, calling `embPause` and `embResume` improves execution performance.

Some I/O or external resources allow interrupt handlers or callbacks to be registered to signal when the resource is available. Typically, the developer can register a callback function along with `calldata`. When the external resource is available, the callback function is called and passed the `calldata`. To use the `embPause` and `embResume` capability, typically an interrupt handler calling `embResume` is registered in the primitive's `Start` method. Also registered during the `Start` method is the `calldata` containing the information returned by `embSelf` and `embGetSchedule`.

An example from a primitive encapsulating the NT audio input as a data source (primitive `embeddable/stream/audio/audioIn_nt`) is described below. The example begins with a description of how `embResume` is registered to be called. The registration of the primitive's callback routine is done in the primitive's `Start` method. For the `audioIn_nt` primitive, the `Start` method is:

```
Start: {
    as = calloc(1, sizeof(AudioStructRec));
    as->s = embGetSchedule();
    as->self = embSelf();
    ...
    error = waveInOpen(..., waveInFunc, as, ...);
    ...
}
```

The `Start` method registers a callback function, `waveInFunc`, that is invoked whenever the audio device has new data ready. The callback function is passed `calldata` that includes the schedule to which the primitive belongs (returned by `embGetSchedule`) and the currently active thread (returned by `embSelf`).

Both the `waveInFunc` and the type `AudioStructRec` are defined by the developer in the primitive's `Include` section and are:

```
Include: {
    typedef struct {
        ...
        Schedule s;
        void *self;
    } AudioStructRec, *AudioStruct;

    void CALLBACK waveInFunc(..., DWORD dwInstance, ...) {
```

```

    AudioStruct as = (AudioStruct)dwInstance;
    ...
    if (as->bufferReady >= as->buffersNeeded) {
        embResume(as->self, as->s);
    }
}
}

```

The callback function that wakes up the primitive contains the call to `embResume`. The function `embResume` must be called with the pointer returned from `embSelf` and the schedule returned from `embGetSchedule`. These values were recorded in `as-self` and `as->s` in the primitive's `Start` method. The value returned by `embSelf` is the thread that may need to be woken up when `embResume` is called (see the following section on Sleep and Wakeup). The function `embGetSchedule` returns the static schedule that contains the primitive. The dynamic scheduler will only resume scheduling of the static schedule (after a call to `embPause`) when `embResume` is called with the schedule as its second parameter.

The purpose of calling `embResume` is to resume execution of the schedule paused by a call to `embPause`. Function `embPause` is called from a primitive's `Apply` method when it is determined that the `Apply` method cannot complete execution until the resource the primitive is accessing becomes available. For the `audioIn_nt` primitive, pseudocode for the `Apply` method is:

```

Apply: {
    if (input is available) {
        move data from input device to primitive output
    } else {
        embPause();
    }
}
}

```

If input is not available, then the `Apply` method calls `embPause` to put the primitive and its associated static schedule in the paused state. There is no restriction on when `embResume` can be called. Geda provides necessary protection so that an interrupt handler calling `embResume` during a call to `embPause` will not cause a problem. A call to `embResume` when the schedule is not in the paused state is legal and has no effect.

### **BSP Developer Considerations:**

BSP developers should consider modifying their BSPs to call `embPause` and `embResume` from any nonblocking communication routines. Such calls will put schedules sending or receiving data from other target processes in a paused state when

the sends or receives are blocked. Appendix C at the end of these release notes describes how the ent BSP was modified to call `embPause` and `embResume` from the nonblocking read and write calls.

## Sleep and Wakeup

Gedae BSPs can now optionally provide the ability to go to sleep. When all schedules have gone to the paused state, a call to the BSP will allow the Gedae process to go to sleep. When `embResume` is called later, the Gedae process can be woken up. The benefit to general users is that if the sleep/wakeup capability is implemented, then Gedae applications waiting for I/O resources will use less power and less CPU. Currently only the ent BSP has implemented the ability to go to sleep. The following description of sleep/wakeup functions is provided for BSP developers that must add these functions to the BSP they are supporting:

### Synopsis:

```
void embGoToSleep(int sec, int nsec);
void embWakeup(void *thread);
void *embSelf(void);
void embSetHostPortSchedule(Schedule s);
```

### Description:

Four new BSP functions `embGoToSleep`, `embWakeup`, `embSelf` and `embSetHostPortSchedule` allow Gedae processes to go to an idle mode. The function `embSelf` is the only one of the four that needs to be called by primitive developers. All four of the functions need to be provided by BSP developers and must be added to any BSP that is being upgraded to Gedae 4.4. A default set of stub functions that can be included in any BSP are described at the end of this section.

When all schedules have gone to the paused state a call to `embGoToSleep` puts the process in sleep mode. If parameters `sec` and `nsec` are both set to zero, then the process goes to sleep until the next call to `embWakeup`. If either the `sec` or `nsec` parameter are nonzero, then the process goes to sleep until the next `embWakeup` call or when time  $sec+1e-9*nsec$  seconds have expired – whichever comes first.

The function `embWakeup` is called by `embResume` and may be called before a call to `embGoToSleep`. To avoid missing a wakeup call, the wakeup and sleep functions should be implemented so that an `embGoToSleep` immediately following a call to `embWakeup` will not go to sleep. Also `embGoToSleep` and `embWakeup` should be

protected so that they will work correctly if called simultaneously. Appendix C shows how these calls have been implemented for ent.

Function `embSelf` should be designed in conjunction with `embWakeup`. The return value of `embSelf` obtained during the call to the primitive's `Start` method should be passed as the first parameter to `embResume`. Function `embResume` will automatically pass the value to `embWakeup`. For many BSPs the value of `embSelf` can be zero, and `embWakeup` will simply ignore the parameter. For some thread based BSPs the value of `embSelf` is a pointer to the thread that `embWakeup` must wake up. Having `embSelf` return a nonzero value is sometimes necessary when multiple processes or threads can be mapped to the same target processor.

The function `embSetHostPortSchedule` is called from the `internal/monitor` primitive to associate the monitor schedule with the control port opened using `embOpenControlPort`. This function is necessary because the control port is opened before the monitor `Start` method is called, therefore, – unlike the other ports that are opened during the `Start` methods of the `send` and `recv` primitives – there is no opportunity to associate the schedule with the port during the port creation procedure.

The following example shows the port creation procedure for ent :

```
static Port createPort(void) {
    Port p = calloc(1, sizeof(PortRec));
    p->thread = embSelf();
    p->s = embGetSchedule();
    p->mutex = PTHREAD_MUTEX_INITIALIZER;
    p->cond = PTHREAD_COND_INITIALIZER;
    return p;
}
```

The function `embGetSchedule` returns a zero when `embOpenControlPort` is called. Thus, to associate the schedule with the control port `embOpenControlPort` saves the control port in a global variable `G_HOST_PORT` and then function `embSetHostPortSchedule` associates the monitor schedule with the port as:

```
void embSetHostPortSchedule(Schedule s) {
    if (G_HOST_PORT) {
        G_HOST_PORT->s = s;
    }
}
```

### **Stub Functions:**

BSPs that are migrating to Gedae 4.4 need to provide the following stub function until a full implementation of the functions is provided:

```
void embGoToSleep(int sec, int nsec) {}  
void embWakeup(void *thread) {}  
void *embSelf(void) { return 0; }  
void embSetHostPortSchedule(void *s) {}
```

## Monitor Schedule

Previously to Gedae 4.4, calls to the target processor command handler were inserted at various places in the target processor runtime kernel to catch and handle commands from the host. If this design were maintained, then it would require the Gedae BSP to poll on the control port for commands even when there were no schedules to run. In Gedae 4.4, the command handler has been made into a separate schedule that contains the single primitive `internal/monitor`. The `internal/monitor` primitive by default runs at a periodicity of 0.5 seconds. This period can be set from the schedule info dialog. If BSP ports have been developed using `embPause` and `embResume`, then the periodicity is unimportant because the `internal/monitor` box will go to the paused state until commands from the host processor arrive. Since the command handler is now just another schedule, when it and all the other schedules have gone to the paused state, the process will call the `embGoToSleep` function and go to sleep.

## 2 Bugs Fixed

This is a partial list of the bugs fixed in this release. For the current status of Gedae bug fixes see: <http://gedae.com/SUPPORT/buglist.html>

### **SCR1421.2: Trace Table Synchronization Error**

The Trace Table time synchronization did not work if the application was terminated and rerun. The synchronization is now correctly reset at the beginning of a new run.

### **SCR1415.2: Graph Using Max Function on Array arg Crashes**

Graphs with graph variables containing expressions like:

```
int Maxoffset[b] = max(A,N)
```

where A is an array that caused Gedae to crash. The problem was the array argument to max was handled incorrectly. The problem has been fixed.

### **SCR1410.2: Peer-to-Peer Communication Problem**

A problem existed in the peer-to-peer communication between the Nt host and the Dy4av2 processor. The problem was a result of the fact that all control structures passed between the processors were 4 byte swapped. However, some of the control structures contained short integers that gave incorrect results when 4 byte swapped. The problem was fixed by changing all kernel data structures to use ints instead of shorts.

### **SCR1409.2: Test and Fix Launch Package Creation**

Some of the options in the launch package creation menu did not work. A test procedure was developed to insure that all of the possible options were tested, and the options that were not working have been fixed.

### **SCR1408.2: Primitive Output Dimension Assert Error**

A primitive output dimension equation containing a parenthesis caused an assert error. For example the `embeddable/var_matrix/complex/vmx_rhalve` box has an output dimension with a parenthesized expression:

```
stream complex a[p0, (Max0+1)/2][q0,Max1]<[(Max+Max1)/2]>;
```

The first term: " $(Max0+1) / 2$ " caused an assert error during the phase of trying to evaluate output expressions based on parameter inputs. The problem occurred only when `Max0` was an input dimension and not a parameter to the box. This problem has been fixed.

### **SCR1407.2: `embTerminateNormal` on Fully Static Partitions.**

On partitions that are fully static (no dynamic queues) the `embTerminateNormal` function did not stop the execution of the calling primitive. This problem has been fixed.

### **SCR1406.2: Unconnected Output Queue Segfault**

An unconnected output queue caused a segfault during schedule creation. This problem has been fixed.

### **SCR1405.2: Graph Editable while Running or Continuable**

It was possible to do various edits of a graph while the graph was running, which frequently would cause the graph to crash during execution. This problem has been fixed by disabling all edits during running. The graph execution must be terminated (not merely stopped with the possibility of continuing) to resume editing.

### **SCR1403.2: `gsim` Crashes After run-stop-run**

If the user stopped or terminated a graph and then reran it, then `gsim` could crash. Threads were freed twice if enqueued on `gsim` thread queue more than once. This problem is now prevented.

### **SCR1402.2: `appFree` Error**

Calling `appFree` could call segmentation violation. This problem has been fixed.

### **SCR1401.2: Launch Package Code-gen Error**

When code-generating a copy-data-base, the copy blocks were named after the memory area in which they were allocated. If two blocks were allocated in the same memory area, then they got the same variable name causing a compilation error. The problem was fixed by adding a serial number to the block name.

## SCR1400.2: External State Transfers Not Efficiently Scheduled

External state is moved between exclusive branches using autoscheduled `recvState` and `sendState` boxes. For efficient scheduling, the `recvState` should be scheduled as late as possible, and `sendState` should be scheduled as early as possible. The `recvState` was scheduled correctly by setting its priority lower than anything else. The `sendState` box was given a higher priority than anything else, however, just setting the `sendState` box priority was insufficient. For the `sendState` to be scheduled early, the Gedae scheduler now does the following: when the `recvState` is scheduled all boxes leading up to the `sendState` are now given a higher priority than boxes that aren't.

## SCR1399.2: Evaluating Dimensions Can Be Slow

A graph containing a box with a parameter output, such as:

```
Output: {  
    int M = N;  
}
```

that drives a parameter input of another box could cause a slow parameter evaluation. The parameter outputs caused parameter evaluation upstream in the graph to fail until the box was explicitly evaluated. Further, if there were multiple paths back to this box, then many boxes evaluations were attempted. This problem has been fixed.

## SCR1393.2: Number of Segment Levels Limited to Four

The maximum number of segment levels has been increased from four to six.

## SCR1392.5: Matlab Library Should Open Engine in Start

Matlab library opened the engine in `Reset` method. If the Matlab boxes were controlled by a segmented stream, then the library did not close the engines properly, leaving Matlab licenses checked out until the processes were killed. The problem was fixed by moving the code to the `Start` method. The `Start` method is the appropriate method to use for code that should get called one time at the beginning of application execution, whether it is segmented or not.

## SCR1389.5: Complex Matlab Boxes Use mxREAL Data

The boxes `mx_mlsave1` and `mx_mlsave2` incorrectly used `mxREAL` data. All uses of `mxREAL` were switched to `mxCOMPLEX`.

## **SCR1388.5: FGU Segfaults**

When FGU was run from the command line, it sometimes segfaulted when restoring archives. The fault only occurred when archive had no common parent directory. The problem has been fixed.

## **SCR1387.5: Canvas Resize**

If the canvas was Ctrl-I resized and the graph did not fit in the window, then the window cut off graph objects so they were not viewable. If a user exited and re-entered, then the canvas was resized so that all objects are viewable by default. This resizing caused scroll-bars to appear (which is a problem for consistent autodocumentation). The solution is to not allow Ctrl-I to cut off objects. The canvas must fit all objects in the graph's bounding box.

## **SCR1386.2: Periodic Schedules Don't Work Correctly in Gsim**

There were two problems with periodic schedules in gsim. First, if a gsim thread became idle and there was a periodic schedule, then instead of waking up when input arrived, the process delayed until the periodic schedule was ready to run. Second, if a process that waited in this way had a delay multiplier of say 0.1 (-d 0.1 set in processor parameters), then the process would wake up prematurely. Both of these problems have been fixed.

## **SCR1382.2: Reset Called Twice at Startup**

This problem was related to `Reset` methods that were not contained within a segmented subgraph. All top-level – unsegmented - `Reset` methods are now only called once at the beginning of graph execution.

## **SCR1381.2: Missing \$GEDAE/include/mcos\_altivec/e\_cvsub.h**

The file `$GEDAE/include/mcos_altivec/e_cvsub.h` was missing. This problem has been fixed.

## **SCR1038: Nt uses 100% of CPU**

On NT, Gedae used 100% of the CPU when idle. This problem has been fixed.

## 3 Known Bugs

The following known bugs are listed by their Software Change Request (SCR) number. Please include the SCR number if you wish to report additional information about this bug. For an online list of SCRs see

<http://gedae.com/SUPPORT/buglist.html>

### SCR1016: Unterminated Comments

Unterminated comment causes the Gedae parser to segfault.

### SCR1097: Parser Problem

The Gedae parser doesn't handle an odd number of quotes (") well.

### SCR1168: Problem with Variable Vectors and Delay

If a dynamic variable vector input is preceded by a `vv_delay`, then the graph segfaults during scheduling.

### SCR1207: Arrays of Strings not Allowed

Gedae currently allows string array graph parameters to be declared as:

```
const string X[] = {"hello", "world"}
```

or

```
string X[i] = [i]Y
```

where `Y` is a family of strings.

In either case, the values so declared are not set correctly, and therefore, should be considered illegal.

### SCR1214.2: Inconsistent Data Type Declarations

Gedae aborts when the same stream type is multiply defined. For example, suppose two primitives define data types with the same name but different definitions. If the primitives use those types in their `Input`, `Local` or `Output` sections, then Gedae aborts.

## **SCR1223.2: Function `appFree` Memory Leak**

A command program running on VxWorks does not free all the resources allocated (memory, sockets, etc). The `appFree` function must release everything allocated. Gedae should automatically generate a call to `appFree` for the standard `exec-host` command program.

## **SCR1258.2: External Code Does Not Recompile**

`Make` is not called after a successful run, so code changes to code listed in the `Personal_Emb_Obj_List` do not get recompiled. To force the recompile, it is currently necessary to change something from the Gedae GUI - like saving a primitive or toggling the Group "Run On Embedded" toggle off and on.

## **SCR1261.2: Large Graphs Fail to Display on Flattened Graph**

If a graph is too large, then it cannot be displayed on the flattened graph. That is, if the flattened width or height exceeds the allowable pixmap width or height.

## **SCR1263.5: Primitive Cannot Recompile**

If a primitive Input, Output or Local section is modified at runtime, then Gedae segfaults when the primitive is recompiled, and the graph is rerun. The user must currently exit Gedae after a primitive Input, Output or Local section has been modified.

## **SCR1277.2: DSA With Fan Out Does Not Work for Some BSPs**

If one box output fans out to several boxes mapped to several different processors, then the DSA communication mechanism does not work correctly for Mercury and Sky BSPs.

## **SCR1291.2: VxWorks `embWallclock` Function Misses Wrap**

When collecting trace information the `embWallclock` function timer can wrap without being detected. This failure to detect the timer wrap causes VxWorks processor timelines to appear compressed.

## **SCR1300.2: Trace Table Saved on NT not Readable on Solaris**

Files saved from NT are byte reversed from what is expected on Solaris. Files saved on a big-endian platform cannot be read by Gedae from a little-endian platform.

## **SCR1326.5: FFT Primitives Only Work with Power of 2 Sized Vectors**

The FFT boxes do not support non-power-of-2 lengths, however, the comments make no mention of this fact. If these boxes only support a power-of-2, it would be useful to have a separate set of boxes that support a non-power-of-2.

## **SCR1379.2: Queues Don't Autoresize When Opening Group Settings**

If a queue needs to be autoresized and the autoresize option is turned on in the group-setting file, then the autoresize does not occur. A workaround for this problem is to toggle the Group Control Dialog Autoresize toggle off, and then on, and then resave the group settings.

## **SCR1386.5: Cannot Put { or } Inside Single Quotes**

A line like `c = '{';` inside a primitive causes an error during primitive parsing. The parser will count the `'}'` as a bracket and fail to find a matching closing bracket. Because brackets inside double quotes are ignored, the expression `"{"` will not cause a problem.

## **SCR1387.2: Set Partition by Equation Error**

When setting a partition by equation on a subgraph family, and if any of the contained boxes are families, then the \$1 variable applies to the deepest family member giving the wrong results.

## **SCR1388.2: Constants Propagated Through typedef Boxes**

Constants propagated through typedef boxes cannot be used for instantiation.

## **SCR1395.2: Partially connected State variable**

State variables should not have to be connected to a primitive in every branch of an exclusive output. Currently, they must be connected when the branches are distributed.

## **SCR1411.2: Gain Propagating Through Internal Queue**

A gain error can result if a gain is propagated through a dynamic queue internal to the static schedule. Gains should not be propagated through dynamic boundaries.

## **SCR1412.2: Segmented Schedules with nondet Inputs**

If a segmented static schedule has only nondet inputs queues, then it will not respond to segment ends correctly. This behavior was originally an intended feature of segmentation but needs to be either enforced during development or changed.

## **SCR1416.2: Primitives with EndOfSegment and no Apply**

Primitives that have an `EndOfSegment` method but don't have an `Apply` method do not get included; therefore, the `EndOfSegment` method does not run. A workaround for this problem is to include an empty `Apply` method in the primitive.

## **SCR1417.2: Stream Box with push in hostless Launch Package**

If a stream box contains a call to `push` and it is made part of a hostless launch package, then the launch package will fail to compile, as the code for the `push` is not included in the standalone library.

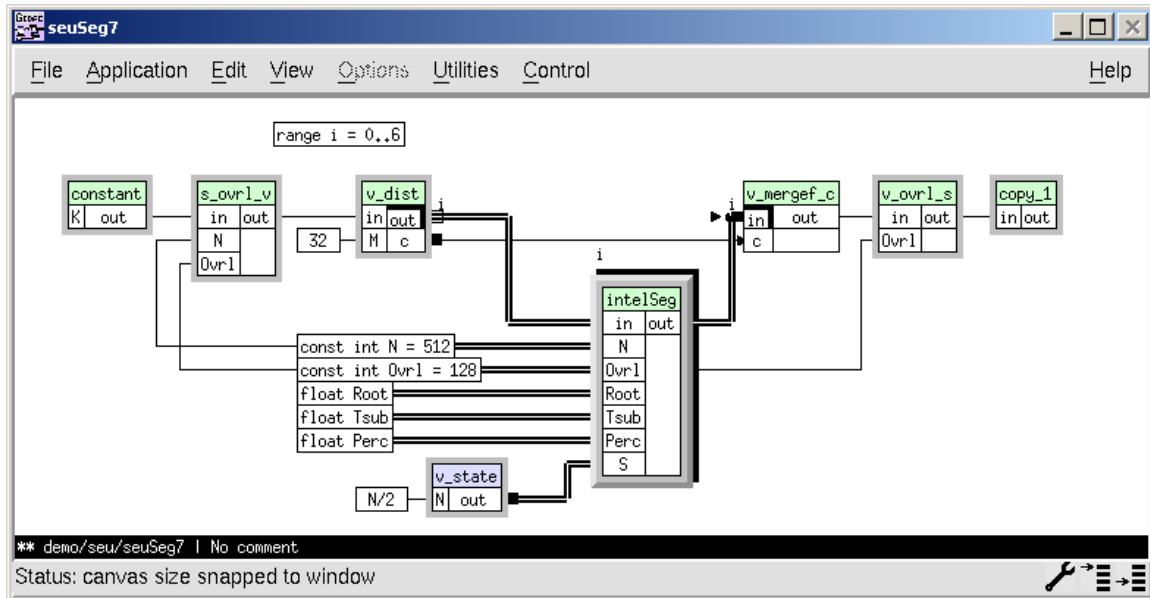
## **SCR1422.2: avail on Variable Vectors is Unreliable**

On variable vectors the `avail` function only returns the number of tokens available on the vector and not on the variable part. Usually these are the same, but when the variable vector is coming from another processor over a dynamic queue they may be updated at different times and may be different. As a result, `avail` may return a number too large so that when the data is processed the variable part may not actually be ready causing a segfault. A workaround is to

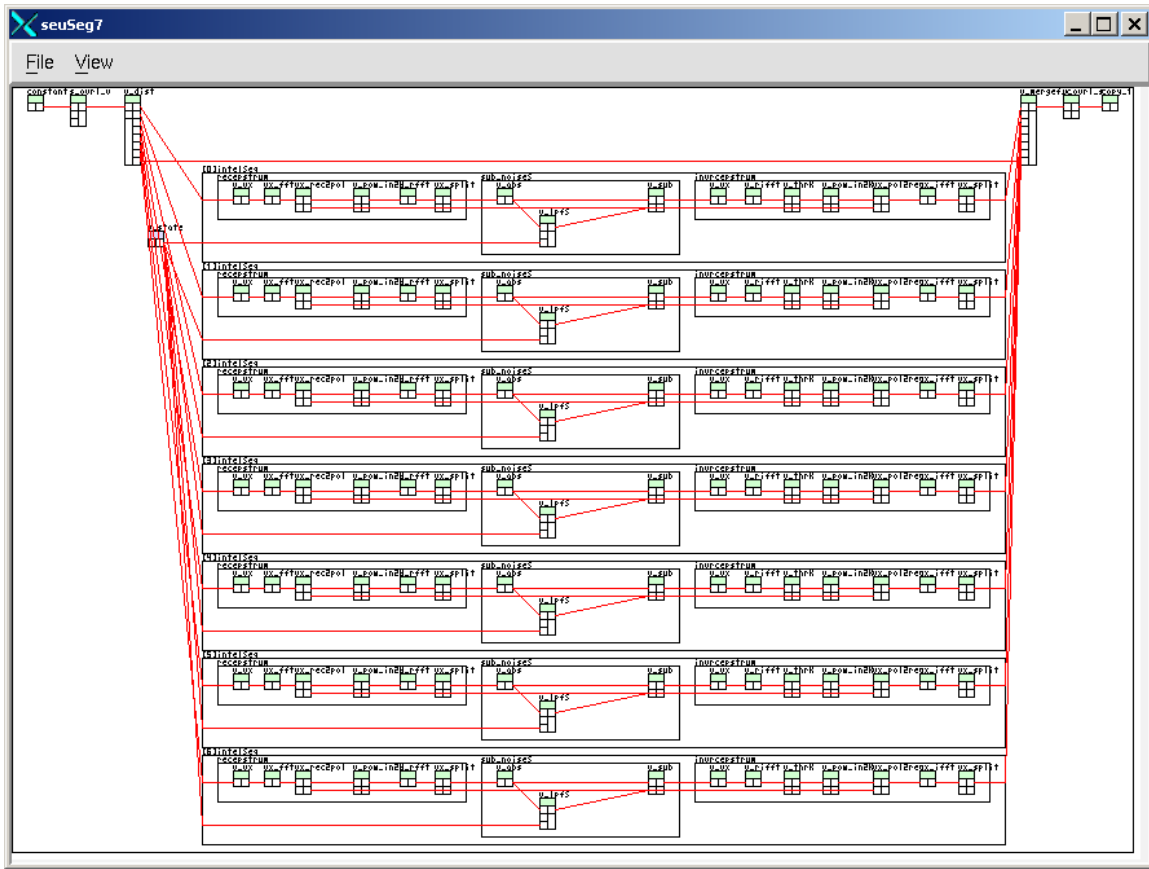
check the `queues_ready` flag after the call to `amount` and only continue with execution if the `queues_ready` flag is non-zero.

## Appendix A: Example Graph on Dy4av2

An example graph running on the Dy4av2 using the Starlink drivers to distribute a graph to eight processors and using the `dsa_bulk` transfer mechanism is shown below. The graph is a speech noise removal algorithm developed by AFRL at Rome Labs, NY. The graph uses a distributed state variable that must be communicated between the seven processors on which the seven intelSeg subgraphs are running



Below is the flattened graph of the seuSeg7 graph. The flattened graph shows seven subgraphs that are mapped to seven processors. It also shows the front-end and back-end processing that consists of the data source, distributor and merge. These functions are all mapped to the eighth processor.



The graph is divided into eight partitions using the partition table as follows:

The 'Group 1 Partition Table' dialog box is shown with a menu bar (File, Edit, View, Options) and a table with three columns: Name, Part, and SubSched. The table lists various segments and their corresponding partition assignments.

Name	Part	SubSched
constant	default *	
s_ovrl_v	default *	
v_dist	default *	
v_mergef_c	default *	
v_ovrl_s	default *	
copy_1	default *	
[0]intelSeg	p0 = "p"+\$1	
[1]intelSeg	p1 = "p"+\$1	
[2]intelSeg	p2 = "p"+\$1	
[3]intelSeg	p3 = "p"+\$1	
[4]intelSeg	p4 = "p"+\$1	
[5]intelSeg	p5 = "p"+\$1	
[6]intelSea	p6 = "p"+\$1	

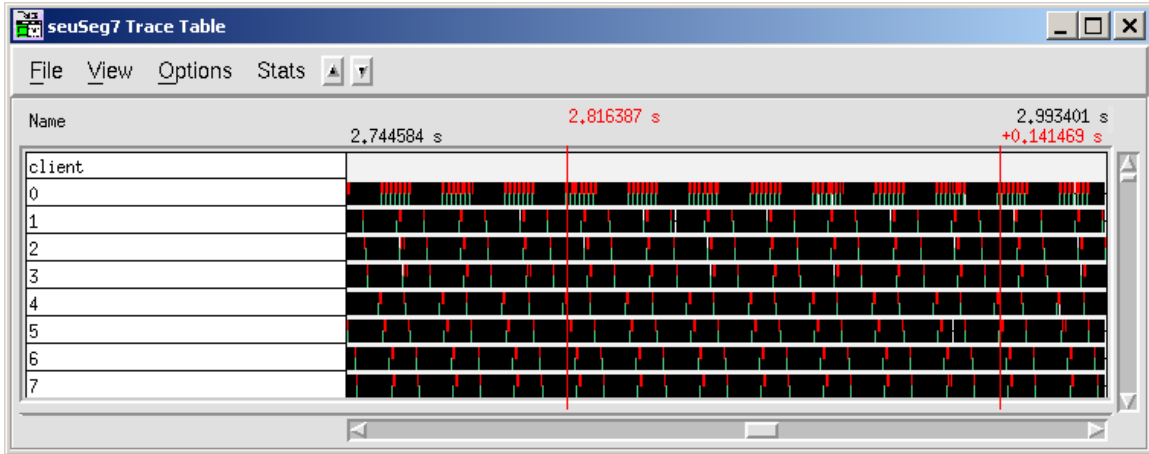
And the eight partitions are mapped to eight Dy4av2 target processors using the mapping table as follows:

Name	Proc Num	System Name	Trace Mem	Trace Size	Params
default	0	dy4av2	default	100000	
p0	1	dy4av2	default	10000	
p1	2	dy4av2	default	10000	
p2	3	dy4av2	default	10000	
p3	4	dy4av2	default	10000	
p4	5	dy4av2	default	10000	
p5	6	dy4av2	default	10000	
p6	7	dy4av2	default	10000	

In the transfer table below, the data transfer mechanism types are all set to `dsa_bulk` with double buffering on the sender and triple buffering on the receiver.

Name	Source	Dest	Xfer Type	NBsize	Send Bufs	Recv Bufs
v_mergef_c<[0]in	1	0	dsa_bulk		2	3
v_mergef_c<[1]in	2	0	dsa_bulk		2	3
v_mergef_c<[2]in	3	0	dsa_bulk		2	3
v_mergef_c<[3]in	4	0	dsa_bulk		2	3
v_mergef_c<[4]in	5	0	dsa_bulk		2	3
v_mergef_c<[5]in	6	0	dsa_bulk		2	3
v_mergef_c<[6]in	7	0	dsa_bulk		2	3
[6]intelSeg.recepstrum.v_vx<in	0	7	dsa_bulk		2	3
[5]intelSeg.recepstrum.v_vx<in	0	6	dsa_bulk		2	3
[4]intelSeg.recepstrum.v_vx<in	0	5	dsa_bulk		2	3
[3]intelSeg.recepstrum.v_vx<in	0	4	dsa_bulk		2	3
[2]intelSeg.recepstrum.v_vx<in	0	3	dsa_bulk		2	3
[1]intelSeg.recepstrum.v_vx<in	0	2	dsa_bulk		2	3
[0]intelSeg.recepstrum.v_vx<in	0	1	dsa_bulk		2	3

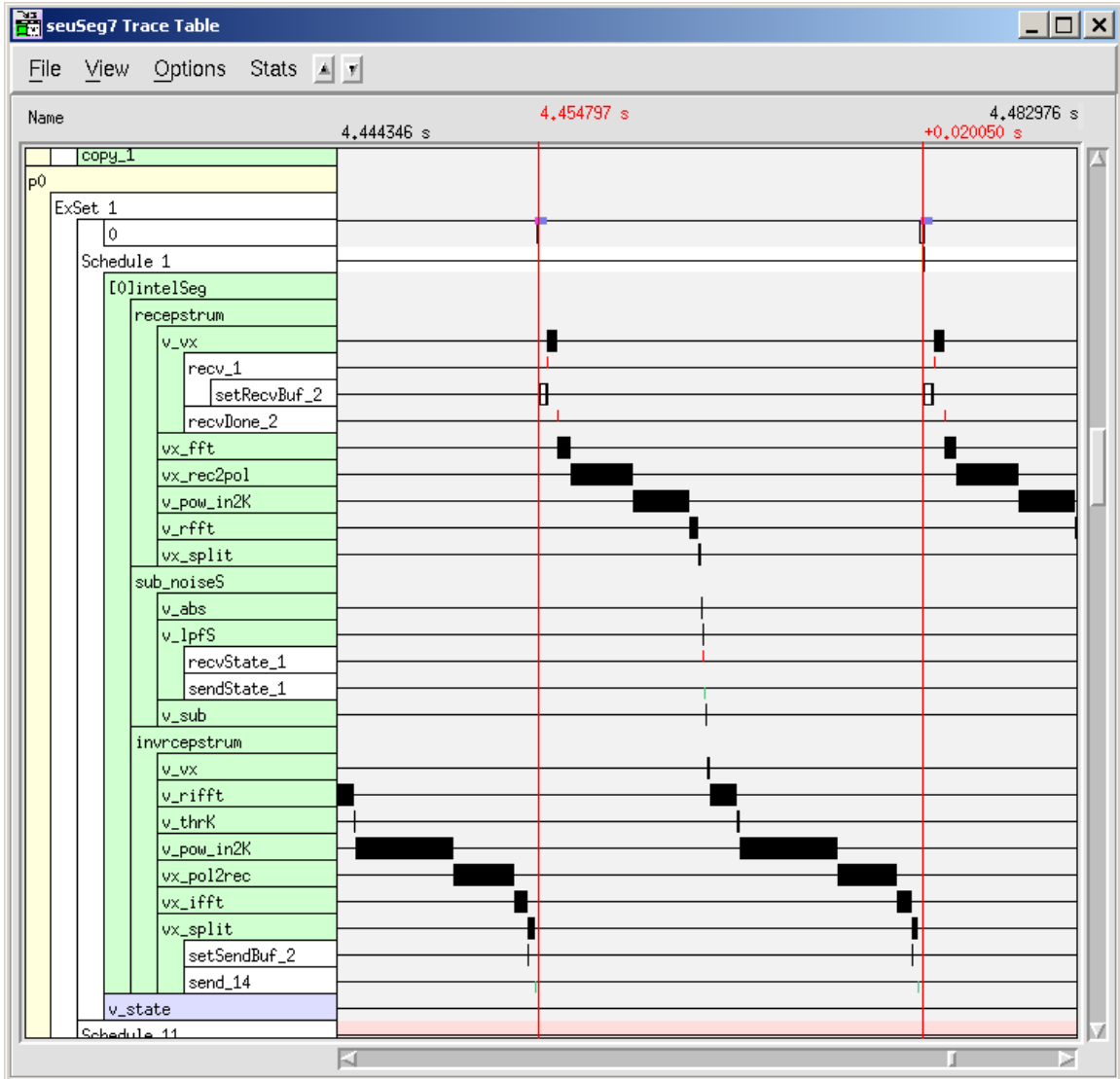
Running the graph produces the following trace table showing a dense timeline on the eight processors. Black sections in the timeline are primitive function box firings doing the ffts, polar to rectangular conversions and filtering functions of the algorithm. The red sections in the timeline are data being received, and the green sections are data being sent.



It is visually obvious that the processors are highly utilized. The statistics table below for processor 6 verifies this high utilization. This table shows that primitives are being executed 95.6% of the time and communication functions are only requiring about 0.6% of the time. The communication time is low because the DSA mechanism allows the communication to be done in parallel with the execution. The communication times in the table are only the times required to kick off the DMA transfers between processors.

Event Type	Total (sec)	Perc (%)	Event Mean (exec/sec)	Burst Rate (bytes/sec)	Throughput (bytes/sec)
Execution	= 4.27	95.62%	0.00107		
Send	= 0.0229	0.51%	5.19e-005	6.32e+008	3.24e+006
Recv	= 0.00369	0.08%	5.56e-006	3.93e+009	3.25e+006
Static Sched	= 0.0213	0.48%	4.19e-006		
Dynamic Sched	= 0.000789	0.02%	3.59e-005		

A detailed view of the execution on partition p0 (processor 1) is shown below. The primitive function boxes that use the Gedae Dy4av2 BSP e\_functions make efficient use of the AltiVec processors. This efficiency is obtained by implementing the e\_function library as a thin layer on the Dy4 Ixlibs-av2 vector processing library.



The most costly functions above are the ones requiring calls to transcendental functions such as `vx_rec2pol` (uses `atan2`), `v_pow_in2K` (uses `pow`) and `vx_pol2rec` (uses `sin` and `cos`). The ffts, real fft and inverse fft and real inverse ffts are comparatively fast.

## Appendix B: Calling embPause/embResume from ent BSP

The ent BSP serves as an example of how embPause and embResume are called from the BSP functions. On both the host and target sides of the BSP whenever a port is created, the schedule for that port is added to the port data structure. The createPort call for both target and host of the ent BSP is:

```
static Port createPort(char *name, int fd) {
    Port p = calloc(1, sizeof(PortRec));
    p->fd = fd;
    p->name = strdup(name);
    p->s = embGetSchedule();
    p->mutex = PTHREAD_MUTEX_INITIALIZER;
    p->cond = PTHREAD_COND_INITIALIZER;
    return p;
}
```

The mutex and cond variable are used by the thread that wakes up the port after a nonblocking call has failed. The ports to be created contain a TCP/IP socket. For input ports to be notified of data becoming available on the socket a posix thread is spawned as:

```
pthread_create(&p->tid, NULL, resumeRead, p);
```

where resumeRead is:

```
static void *resumeRead(Port p) {
    int fd = p->fd;
    fd_set readfds;
    while (1) {
        pthread_mutex_lock(&p->mutex);
        if (!p->waiting_for_port) {
            pthread_cond_wait(&p->cond, &p->mutex);
        }
        p->waiting_for_port = 0;
        pthread_mutex_unlock(&p->mutex);
        FD_ZERO(&readfds);
        FD_SET(fd, &readfds);
        select(fd+1, &readfds, 0, 0, 0);
        p->wakeup = 1;
        embResume(p->thread, p->s);
    }
}
```

Similarly for output ports, a posix thread is spawned that calls the function `resumeWrite`:

```
static void *resumeWrite(Port p) {
    int fd = p->fd;
    fd_set writefds;
    while (1) {
        pthread_mutex_lock(&p->mutex);
        if (!p->waiting_for_port) {
            pthread_cond_wait(&p->cond, &p->mutex);
        }
        p->waiting_for_port = 0;
        pthread_mutex_unlock(&p->mutex);
        FD_ZERO(&writefds);
        FD_SET(fd, &writefds);
        select(fd+1, 0, &writefds, 0, 0);
        p->wakeup = 1;
        embResume(p->thread, p->s);
    }
}
```

Both threads are infinite loops. Each pass in the loop waits for a signal that comes from the nonblocking read or write call when that call has failed. At that point, the thread proceeds to block in a select call until the socket goes to the unblocked state. This unblocking happens when data becomes available on a read port or data is read by the destination of a write port. When the socket becomes unblocked, select returns and the `embResume` function is called. This function puts both the schedule `p->s` in the ready state and calls `embWakeup` on the thread `p->thread`.

The nonblocking read and write functions call `checkSuspend` and pass to it the number of bytes read or written. The function is:

```
static void checkSuspend(Port p, int progressBytes ) {
    if(progressBytes == 0) {
        pthread_mutex_lock(&p->mutex);
        embPause();
        p->waiting_for_port = 1;
        pthread_cond_signal(&p->cond);
        pthread_mutex_unlock(&p->mutex);
    }
}
```

If `progressBytes` are zero, indicating no progress was made by the nonblocking call, then the function calls `embPause` putting the schedule in the paused state and signals the `resumeRead` thread (for a read) or the `resumeWrite` thread (for a write) to wait until the socket becomes ready.

## Appendix C: embWakeup and embGoToSleep Implementations for ent BSP

```
/* WAKEUP is true if call to embWakeup has occurred since the last call to
embGoToSleep has finished. SLEEPING is true when embGoToSleep goes to sleep and
is not set back to 0 until the next call to embWakeup */
```

```
static int WAKEUP = 0;
static int SLEEPING = 0;
```

```
/* used to wake up an embGoToSleep call after timeout time. Note that embWakeup is
used. */
```

```
static void *wakeupHandler(void *thread) {
    Sleep(SLEEP_TIME_MS);
    embWakeup(thread);
    return 0;
}
```

```
/* note how WAKEUP and SLEEPING variables are critical section protected by the
mutex_lock, mutex_unlock and cond_wait calls. */
```

```
void embGoToSleep(int sec, int nsec) {
    pthread_mutex_lock(&GOTOSLEEP_MUTEX);
    if (WAKEUP==0) {
        pthread_t tid = 0;
        if ((sec || nsec) && sec < 1000000) {
            SLEEP_TIME_MS = (DWORD)(1000*sec + 1e-6*nsec);
            pthread_create(&tid, NULL, wakeupHandler, embSelf());
        }
        SLEEPING = 1;
        pthread_cond_wait(&GOTOSLEEP_COND, &GOTOSLEEP_MUTEX);
        if (tid) {
            pthread_cancel(tid);
        }
    } else {
        WAKEUP = 0;
    }
    pthread_mutex_unlock(&GOTOSLEEP_MUTEX);
}
```

```
void embWakeup(void *thread) {
    pthread_mutex_lock(&GOTOSLEEP_MUTEX);
    if (SLEEPING) {
        SLEEPING = 0;
        pthread_cond_signal(&GOTOSLEEP_COND);
    }
}
```

```
    }
    WAKEUP = 1;
    pthread_mutex_unlock(&GOTOSLEEP_MUTEX);
}

/* since embWakeup for this BSP doesn't need a thread id embSelf returns 0 */
void *embSelf(void) {
    return 0;
}

/* this function is called by the internal/monitor Start method to set the control ports
schedule field */
void embSetHostPortSchedule(Schedule s) {
    if (G_HOST_PORT) {
        G_HOST_PORT->s = s;
    }
}
```