PBS Portable Batch System

Internal Design Specification

Albeaus Bayucan
Casimir Lesiak
Bhroam Mann
Robert L. Henderson
Tom Proett
Dave Tweten †

MRJ Technology Solutions
2672 Bayshore Parkway
Suite 810
Mountain View, CA 94043
http://pbs.mrj.com

Release: 2.2
Printed: November 30, 1999

† Numerical Aerospace Simulation Systems Division, NASA Ames Research Center, Moffett Field, CA

PBS IDS

Portable Batch System (PBS) Software License

Acknowledgment: The Portable Batch System Software was originally developed as a joint project between the Numerical Aerospace Simulation (NAS) Systems Division of NASA Ames Research Center and the National Energy Research Supercomputer Center (NERSC) of Lawrence Livermore National Laboratory.

Redistribution of the Portable Batch System Software and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright and acknowledgment notices, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright and acknowledgment notices, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:

  This product includes software developed by NASA Ames Research Center, Lawrence Livermore National Laboratory, and MRJ Technology Solutions.

DISCLAIMER OF WARRANTY

THIS SOFTWARE IS PROVIDED BY MRJ TECHNOLOGY SOLUTIONS ("MRJ") "AS IS" WITHOUT WARRANTY OF ANY KIND, AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT ARE EXPRESSLY DISCLAIMED.

IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW, SHALL MRJ, NASA, NOR THE U.S. GOVERNMENT BE LIABLE FOR ANY DIRECT DAMAGES WHATSOEVER, NOR ANY INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This license will be governed by the laws of the Commonwealth of Virginia, without reference to its choice of law rules.

This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

PBS Revision History

Revision 1.0	June, 1994 — Alpha Test Release
Revision 1.1	March 15, 1995
...	
Revision 1.1.9	December 20, 1996
Revision 1.1.10	July 31, 1997
Revision 1.1.11	December 19, 1997
Revision 1.1.12	July 9, 1998
Revision 2.0	October 14, 1998
Revision 2.1	May 12, 1999
Revision 2.2	November 30, 1999

Table of Contents

[This page is blank.]

1. Introduction

The Portable Batch System is a product designed and developed at the Numerical Aerodynamic Simulation Systems Division at NASA's Ames Research Center, and at the Livermore Computer Center and National Energy Research Supercomputer Center, both at Lawrence Livermore National Labs.

This product provides support for batch processing on POSIX[1] and UNIX[2] systems.

1.1. Purpose

This document is the Portable Batch System Internal Design Specification, also called the **IDS**. The IDS describes the detailed design of the Portable Batch System, **PBS**. Included in this design is detailed information about each package that makes up PBS, the programs that make up each package, the files and functions that make up each program. The level of detail is such to provide an experienced C language and Unix programmer with all the information required to understand, maintain, and expand PBS.

The structure of this document is provided by figure 1−1.

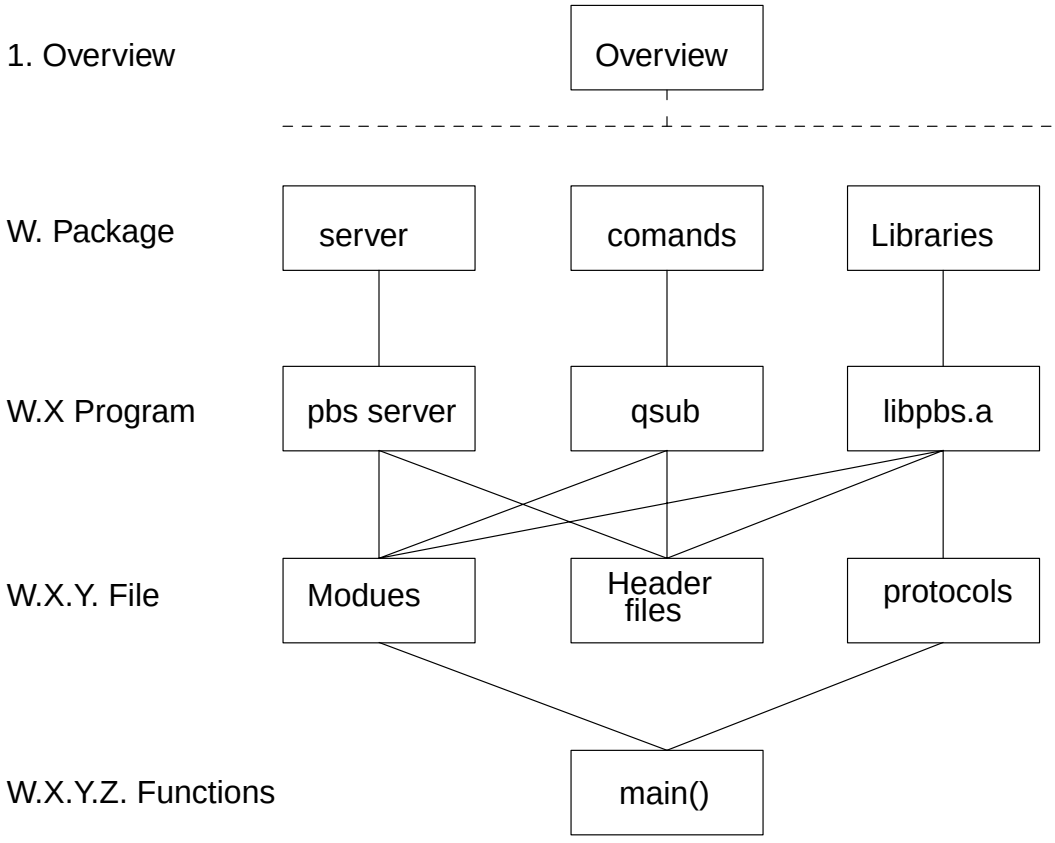


Figure 1−1: IDS Document Structure

[1] Copyright IEEE, see IEEE standards 1003.1, 1003.2, and 1003.2d.

[2] Unix is a trademark of USL.

Two other documents are provided as part of the **PBS** package, the Portable Batch System Requirements Specification, the Requirements Spec, and the Portable Batch System External Reference Specification, the ERS. It is strongly recommended that those two documents be read before attempting to delve into this document.

1.2. Glossary

In order to save trees, the glossary is not reproduced here. Please refer to the glossary section of the PBS ERS.

1.3. System Overview

1.3.1. Batch Pre-history

PBS was developed to provide support for batch processing. As opposed to simply placing a process in the background, batch processing encompasses the scheduling of multiple jobs according to a policy of system resource management. Each job may consist of multiple processes. Jobs may be routed to processing hosts over a network. Resources may be reserved for a job before its execution begins, and limits are established on the consumption of resources by the job. This goes well beyond the traditional process scheduling provided by Unix systems.

Other batch systems have been developed for Unix system. The most well know was NQS, also developed at the Numerical Aerodynamic Simulation Systems Division, NAS. **PBS** was developed to:

- Overcome the problems associated with NQS.
- Extend the features of NQS.
- Be a superset of the POSIX Batch Extensions Standard, P1003.2d.
- Provide a growth platform for batch cluster computing and distributed jobs.

1.3.2. PBS Overview

The PBS batch system consist of the following major sections:

1. The user/operator/administrator client commands which are discussed in chapters 2, 3, and 4.
2. The main server, pbs_server, which is the focal point for all client communication and manages the batch jobs. It is covered in chapter 5.
3. The job scheduler, pbs_sched, which determines which jobs should be executed. There are two supplied version discussed in chapter 6.
4. The job executor and resource monitor, pbs_mom, which manages job execution and monitors system activity and resource usage on a "per host" basis. MOM's resource monitor functons are covered in chapter 7 and job executor functons are described in chapter 8.
5. The user client authentication system is covered in chapter 9.
6. A number of libraries, including the API, are discussed in chapter 10.
8. The network based interprocess communication between the client commands and the server, between the server and MOM, and between the server and the scheduler is covered in chapter 11.

The design concept of PBS follows the "client − server" paradigm. Clients make requests of the server to perform actions on a set of objects. The actions include creation, deletion, modification, and status. There are three classes of objects known to the PBS Server: server, queue, and job. The PBS job executor, MOM, is also aware of the job object, but not the others.

The server owns and manages all batch jobs. All access to jobs are through requests the server. The server performs other services for jobs on a time or event driven basis. These services are known as deferred services. Deferred services include initiation into execution, routing to processing hosts, and resource management.

Queues are collection points for jobs. The term queue is used for historical reasons, it does not imply any ordering of jobs within the queue. A better term might have been “pool”.

Each of the objects discussed above consist of a name and a set of attributes. Attributes are a data name and data value. The objects and their attributes are discussed in great detail at the start of chapter 5.

[This page is blank.]

2. User Commands

2.1. User Commands Overview

This section describes the commands available to the general user. Unless otherwise noted, the command must conform to the POSIX.15 specification of the command as to syntax and functionality.

When more than one operand is specified on the command line, the command processes each operand in turn. An error reply from a server on one operand will be noted in the standard error stream. The command continues processing the other operands. If an error reply was received for any operand, the final exit status for the command will be greater than zero.

2.2. Packaging

The source code for each of these routines consists a single file for each routine and a set of library routines that are shared among some of the other programs. The main file for each command contains a section that parses the execute line options followed by a loop that executes the appropriate command for each operand on the command line. The library has routines for holding attributes created from the options, separating the parts of each operand, and related functions.

2.3. Program: qalter

The **qalter** command modifies the attributes of the job or jobs specified by *job_identifier* on the command line. Only those attributes listed as options on the command will be modified. If any of the specified attributes cannot be modified for a job for any reason, none of that jobs attributes will be modified.

The **qalter** command accomplishes the modifications by sending a *Modify Job* batch request to the batch server which owns each job.

2.3.1. Overview

Parse the options on the execute line and build up an attribute list. For each job given, send the attribute list in the *Modify Job* batch request to the batch server which owns each job.

2.3.2. External Interfaces

Upon successful processing of all the operands presented to the the qalter command, the exit status will be a value of zero.

If the qalter command fails to process any operand, the command exits with a value greater than zero.

2.3.3. File: qalter.c

This file contains the main routine only. All other functions are in the library.

2.3.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[−a date_time] [−A account_string] [−c interval] [−e path] [−h hold_list] [−j join] [−k keep] [−l resource_list] [−m mail_options] [−M user_list] [−N jobname] [−o path] [−p priority] [−r c] [-S path] [−u user_list] [−W dependency_list] job_identifier...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
Build the attribute list for the modify job request
Get appropriate path name where needed
For each remaining operand
Determine the job identifier and server name
cnt:
Connect to the server
Send the modify job request
f Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.4. Program: qdel

The **qdel** command deletes jobs in the order in which their job identifiers are presented to the command. A job is deleted by sending a *Delete Job* batch request to the batch server that owns the job. A job that has been deleted is no longer subject to management by batch services.

2.4.1. Overview

Parse the options on the execute line. For each job on the execute line send a *Delete Job* batch request to the batch server that owns the job.

2.4.2. External Interfaces

Upon successful processing of all the operands presented to the the qdel command, the exit status will be a value of zero.

If the qdel command fails to process any operand, the command exits with a value greater than zero.

2.4.3. File: qdel.c

This file contains the main routine only. All other functions are in the library.

2.4.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[−W delay] job_identifier...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
Build the delay argument for the delete job request
For each remaining operand
Determine the job identifier and server name
cnt:
Connect to the server
Send the delete job request
f Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.5. Program: qhold

The **qhold** command requests that a server place one or more holds on a job. A job that has a hold is not eligible for execution. The **qhold** command sends a *Hold Job* batch request to the server that owns the job.

2.5.1. Overview

Parse the options on the execute line. For each job on the execute line send a *Hold Job* batch request to the batch server that owns the job.

2.5.2. External Interfaces

Upon successful processing of all the operands presented to the the qhold command, the exit status will be a value of zero.

If the qhold command fails to process any operand, the command exits with a value greater than zero.

2.5.3. File: qhold.c

This file contains the main routine only. All other functions are in the library.

2.5.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[−h hold_list] job_identifier ...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:
Zero, if no errors are detected. Positive, otherwise.

Control Flow:
Use getopt to get each option
Set the hold type and expedite flag
For each remaining operand
Determine the job identifier and server name
cnt:
Connect to the server
Send the hold job request
f Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.6. Program: qmove

To move a job is to remove the job from the queue in which it resides and instantiate the job in another queue. The **qmove** command issues a *Move Job* batch request to the batch server that currently owns each job.

2.6.1. Overview

Get the destination from the execute line, and for each remaining argument, move the requested job to the destination queue.

2.6.2. External Interfaces

Upon successful processing of all the operands presented to the the qmove command, the exit status will be a value of zero.

If the qmove command fails to process any operand, the command exits with a value greater than zero.

2.6.3. File: qmove.c

This file contains the main routine only. All other functions are in the library.

2.6.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
destination job_identifier ...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:
Zero, if no errors are detected. Positive, otherwise.

Control Flow:
    The first argument is the destination
    For each remaining operand
        Determine the job identifier and server name
    cnt:
        Connect to the server
        Send the move job request
        f Unknown Job Id and not Located Then
            Locate job
            go to cnt
        Disconnect from the server

2.7. Program: qmsg

The **qmsg** command writes messages into the files of jobs by sending a *Job Message* batch request to the batch server that owns the job.

2.7.1. Overview

Parse the options on the execute line. The argument after any options is the message. Send the message to each job indicated by the remaining operands.

2.7.2. External Interfaces

Upon successful processing of all the operands presented to the the qmsg command, the exit status will be a value of zero.

If the qmsg command fails to process any operand, the command exits with a value greater than zero.

2.7.3. File: qmsg.c

This file contains the main routine only. All other functions are in the library.

2.7.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc	The number of arguments on the command line.
argv	The **argv** array contain the following arguments: [-E] [-O] message_string job_identifier ...
envp	The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:
    Zero, if no errors are detected. Positive, otherwise.

Control Flow:
    Use getopt to get each option
        Set the tofile flag
    The next argument is the message string
    For each remaining operand
        Determine the job identifier and server name

cnt:
Connect to the server
Send the job message request
f Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.8. Program: qrerun

The **qrerun** command directs that the specified jobs are to be rerun if possible.

2.8.1. Overview

For each operand try rerunning the job.

2.8.2. External Interfaces

Upon successful processing of all the operands presented to the the qrerun command, the exit status will be a value of zero.

If the qrerun command fails to process any operand, the command exits with a value greater than zero.

2.8.3. File: qrerun.c

This file contains the main routine only. All other functions are in the library.

2.8.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
job_identifier ...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:
Zero, if no errors are detected. Positive, otherwise.

Control Flow:
For each operand
Determine the job identifier and server name
cnt:
Connect to the server
Send the rerun job request
f Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.9. Program: qrls

The **qrls** command removes or releases holds which exist on batch jobs.

2.9.1. Overview

Parse the execute line to get the hold type. For each remaining operand release the hold on the indicated job.

2.9.2. External Interfaces

Upon successful processing of all the operands presented to the the qrls command, the exit status will be a value of zero.

If the qrls command fails to process any operand, the command exits with a value greater than zero.

2.9.3. File: qrls.c

This file contains the main routine only. All other functions are in the library.

2.9.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[-h hold_list] job_identifier ...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
Set the hold type
For each remaining operand
Determine the job identifier and server name
cnt:
Connect to the server
Send the release job request
f Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.10. Program: qselect

The **qselect** command provides a method to list the job identifier of those jobs which meet a list of selection criteria.

2.10.1. Overview

Parse the options on the execute line and build up an attribute list. For each job given, send the attribute list in the *Select Job* batch request to the batch server which owns each job.

2.10.2. External Interfaces

Upon successful processing of all the operands presented to the the qselect command, the exit status will be a value of zero.

If the qselect command fails to process any operand, the command exits with a value greater than zero.

2.10.3. File: qselect.c

This file contains the main routine and some other functions related to job selection only. All other functions are in the library.

2.10.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

- argc — The number of arguments on the command line.
- argv — The **argv** array contain the following arguments:
  [−a [op]date_time] [−A account_string] [−c [op]interval] [−h hold_list] [−l resource_list] [−N name] [−p [op]priority] [−q destination] [−r c] [-s states] [−u user-name]
- envp — The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
    Build the attribute list for the select job request
If destination is given Then
    Determine the queue and server name
Connect to the server
Send the select job request
Print the job identifiers returned
Disconnect from the server

2.10.3.2.

set_attrop()

```
void set_attrop(struct attropl **list, char *name, char *resource, char *value, enum batch
```

Args:

list The attribute list.

name The name part of the attribute.

resource The resource part of the attribute.

value The value part of the attribute.

op The operation part of the attribute.

Control Flow:

Allocate the memory for an attribute structure
If name is defined Then
    Allocate the memory for the name part
    Copy the name part
If resource is defined Then
    Allocate the memory for the resource part
    Copy the resource part
If value is defined Then
    Allocate the memory for the value part
    Copy the value part
Set the operation part
Add the attribute structure to the beginning of the attribute list

2.10.3.3.

check_op()

```
void check_op(char *opstring, enum batch_op *op, char *value)
```

Args:

opstring The operator and value string from the command line.

op The operator part of the string turned into an enum batch_op. The operator defaults to EQ if none is given.

value The value part of the string.

Control Flow:

Set the operatro to EQ
If opstring contains an operator Then
    Find out which operator was used
Copy the value part

2.10.3.4.

check_res_op()

```
int check_res_op(char *resources, char *name, enum batch_op *op, char *value, char **posit
```

Args:

resources The comma delimited list of resources. The list looks like

    name op value, ...

name The resource name.

op The operator.

value The value.

position The next position in the resource list to parse.

Returns:

Zero, if the resource list is parsed correctly, one otherwise.

Control Flow:

Scan for the resource name
Find out which operator was used
Scan for the resource value
Set the next character position

2.10.3.5.

check_user()

```
int check_user(char *users, enum batch_op *op, char *user, char **position)
```

Args:

users The comma delimited list of users.

op The operator is always EQ.

user The next user name in the list.

position The next character position to parse in the list.

Returns:

Zero, there are no errors.

Control Flow:

Set the operator to EQ Scan for the user name

2.11. Program: qsig

The **qsig** command requests that a signal be sent to executing batch jobs.

2.11.1. Overview

Parse the execute line to get the signal. For each remaining operand send the signal to the indicated job.

2.11.2. External Interfaces

Upon successful processing of all the operands presented to the the qsig command, the exit status will be a value of zero.

If the qsig command fails to process any operand, the command exits with a value greater than zero.

2.11.3. File: qsig.c

This file contains the main routine only. All other functions are in the library.

2.11.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[-s signal] job_identifier ...

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
Set the signal
For each remaining operand
Determine the job identifier and server name
cnt:
Connect to the server
Send the signal job request
If Unknown Job Id and not Located Then
Locate job
go to cnt
Disconnect from the server

2.12. Program: qstat

The **qstat** command is used to request the status of jobs, queues, or a batch server.

2.12.1. Overview

Parse the execute line for options and set the mode of the status request. Depending on the mode set, give the status of the jobs listed on the execute line, the queues listed on the execute line, or the server.

2.12.2. External Interfaces

Upon successful processing of all the operands presented to the the qstat command, the exit status will be a value of zero.

If the qstat command fails to process any operand, the command exits with a value greater than zero.

2.12.3. File: qstat.c

This file contains the main routine and some other functions related to job status only. All other functions are in the library.

2.12.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

[-f] [job_identifier... | destination...]

-Q [-f] [destination...]

-B [-f] [server_name]

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

```
Use getopt to get each option
    Set the full display flag and the mode of the status request
If no operands Then
    Case mode is
    job:
        Set default job id and server name
        go to job_no_args
    queue:
        Set default queue name
        go to queue_no_args
    server:
        Set default server name
        go to server_no_args
For each remaining operand
    Case mode is
    job:
        If operand is a job id Then
            Get full job id and server name
        Else
            Get queue and server name
job_no_args:
        Connect to the server
        Send the status job request
        If Unknown Job Id and not Located Then
            Locate job
            go to job_no_args
        Print the job status returned
        Disconnect from the server
    queue:
        Get queue and server name
queue_no_args:
        Connect to the server
        Send the status queue request
        Print the queue status returned
        Disconnect from the server
```

server:
server_no_args:
Connect to the server
Send the status server request
Print the server status returned
Disconnect from the server

2.12.3.2.

isjobid()

```
int isjobid(char *string)
```

Args:

string Is this string a job identifier? A job identifier begins with a number.

Returns:

True, if the string is a job identifier, false otherwise.

Control Flow:

Is the first non-blank character a digit?

2.12.3.3.

istrue()

```
int istrue(char *string)
```

Args:

string Is this string some textual form of TRUE?

Returns:

True, if the strings represents true, false otherwise.

Control Flow:

Does the string match TRUE
Does the string match True
Does the string match true
Does the string match 1

2.12.3.4.

states()

```
void states(char *string, char *q, char *r, char *h, char *w, char *t, char *e, int len)
```

Args:

string The string that holds the count of jobs in each state from the server.

q The number of queued jobs.

r The number of running jobs.

h The number of held jobs.

w The number of waiting jobs.

t The number of jobs in transit.

e The number of exiting jobs.

Control Flow:

While the string is not empty Do
Scan for the next word
If it is Queued Then set the output pointer to q
If it is Running Then set the output pointer to r
If it is Held Then set the output pointer to h
If it is Waiting Then set the output pointer to w
If it is Transit Then set the output pointer to t
If it is Exiting Then set the output pointer to e
Copy the next word to where the output pointer is pointing

2.12.3.5.

display_statjob()

```
void display_statjob(struct batch_status *status, int header, int full)
```

Args:

A list of information about each job returned by the server.

header True, if the header is to be printed, false otherwise.

full True, if a full display is requested, false for a normal display.

Control Flow:

If not full and header Then
Print the header
While there is an item in the status list Do
If full Then
Print a full job display of all the attributes
Else
Print a normal display of the attributes listed in the ERS
Get the next item in the list

2.12.3.6.

display_statque()

```
void display_statque(struct batch_status *status, int header, int full)
```

Args:

status A list of information about each queue returned by the server.

header True, if the header is to be printed, false otherwise.

full True, if a full display is requested, false for a normal display.

Control Flow:

```
If not full and header Then
    Print the header
While there is an item in the status list Do
    If full Then
        Print a full queue display of all the attributes
    Else
        Print a normal display of the attributes listed in the ERS
    Get the next item in the list
```

2.12.3.7.

display_statserver()

```
void display_statserver(struct batch_status *status, int header, int full)
```

Args:

status A list of information about the server returned by the server.

header True, if the header is to be printed, false otherwise.

full True, if a full display is requested, false for a normal display.

Control Flow:

```
If not full and header Then
    Print the header
While there is an item in the status list Do
    If full Then
        Print a full server display of all the attributes
    Else
        Print a normal display of the attributes listed in the ERS
    Get the next item in the list
```

2.13. Program: qsub

To create a job is to submit an executable script to a batch server. Typically, the script is a shell script which will be executed by a command shell such as sh or csh.

2.13.1. Overview

Parse the execute line and build an attribute list. Get the script and check for embedded operands. Send the script to the server, and, if successful, display the retunred job identifier.

If the job is an interactive job, qsub binds a socket to a port and passes the port number as the interactive attribute. After submitting the job, qsub waits to accept a connection from MOM on that socket. Data from standard in is written to the socket and data from the socket is written to standard out.

2.13.2. External Interfaces

Upon successful processing of all the operands presented to the the qsub command, the exit status will be a value of zero.

If the qsub command fails to process any operand, the command exits with a value greater than zero.

2.13.3. File: qsub.c

This file contains the main routine and some other functions related to job submission only. All other functions are in the library.

2.13.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[−a date_time] [−A account_string] [−c interval] [−C directive_prefix] [−e pathname] [−h] [−j join] [−k keep] [−l resource_list] [−m mail_options] [−M user_list] [−N name] [−o pathname] [−p priority] [−q destination_id] [−r c] [−S pathname] [−u user_list] [−W dependency_list] [−v variable_list] [−V] [−z] [script]

envp The **envp** array contains environment variables for this process. The variables HOME, LOGNAME, PATH, MAIL, SHELL and TZ are sent along with the job.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
Build an attribute list to go in the submit job request
Get the script and any options embedded in the script
Get the queue and server name
Connect to the server
Build the list of environment variables to send
Send the submit job request
Print the job identifier returned
If the job is an interactive job,
Install handler for SIGTSTP so ^Z will not suspend qsub and hand up MOM.
Call interactive to wait for a connection from the job.
Disconnect from the server

2.13.3.2.

set_dir_prefix()

```
char *set_dir_prefix(char *prefix)
```

Args:

prefix The directive prefix supplied by the user, if given.

Returns:

The directive prefix.

Control Flow:
If prefix has something in it Then
Use prefix
Else If the environment variable PBS_DPREFIX is defined Then
Use PBS_DPREFIX
Else
Use the default PBS_DPREFIX_DEFAULT

2.13.3.3.

isexecutable()

```
int isexecutable(char *line)
```

Args:
line A line of the script file.
Returns:
True, if the line is not a comment, false otherwise.
Control Flow:
Is the first non-blank character a #?

2.13.3.4.

ispbsdir()

```
int ispbsdir(char *line)
```

Args:
line A line from the script file.
Returns:
True, if the line is a PBS directive, false otherwise.
Control Flow:
Does the first part of the line match the PBS directive prefix?

2.13.3.5.

get_script()

```
int get_script(FILE *file, char *script, char *prefix)
```

Args:
file The file descriptor of the script.
script The name of the copy that is made of the script.
prefix The PBS directive prefix.

Returns:
Zero, if the script was copied okay, non-zero otherwise.

Control Flow:

```
Create a temporary file
While there is a line left in the script file Do
    If first line of the file Then
        Check for : or #!
    If no executable statements yet and this is a PBS directive Then
        Continuation is TRUE
        While Continuation D0
            Check if this line is continued
            Parse the PBS directives
            If the line is continued Then
                Get the next line in the script
    Else If no executable statements and this an executable statement Then
        Now there are executable statements
    Write the line to the temporary file
```

2.13.3.6.

make_argv()

```
void make_argv(int *argc, char *argv[], char *line)
```

Args:

argc	The number of PBS directives found in the line.
argv	The individual PBS directives.
line	The PBS directives line from the script.

Control Flow:

```
Set argv[0] to qsub
While the line is not empty Do
    If the next character is a quote Then
        Find the matching quote
        Make it a blank
    Scan for the next blank
    Allocate memory for the word
    Copy the word
    Put the word's address into the argv array
    Increment the number of things in argv
```

2.13.3.7.

do_dir

```
int do_dir(char *line)
```

Args:

line A PBS directives line from the script.

Returns:

The value returned from processing the directives (see process_opts).

Control Flow:

If the first time through Then
Clear out the array that will hold the words of the line
Parse the line into words
Process the word list

2.13.3.8.

set_job_env()

```
int set_job_env(char **environment)
```

Args:

environment
The environment variables known to this process.

Returns:

True, if the environment was made correctly, false otherwise.

Control Flow:

Calculate how big to make the environment
Allocate the memory for the environment
Put the required variables in the environment
Parse the environment variables from the command line and add them
If -V was given Then
Add all the environment variable known to this process
Add the environment to the attribute list

2.13.3.9. Interactive job support routines

If interactive job support is compiled in...

interactive_port()

Returns:

a numeric character string representing the port number obtained.

This routine is called if the interactive attribute is specified either by the -I option or directly via -W.

A socket is opened an bound to a port. The port number assigned it obtained and encoded into a numeric string which is returned. If errors occur, the program exits.

settermraw()

```
void settermraw(struct termios *ptio)
```

Args:

ptio Pointer to the saved terminal characteristics.

The saved terminal charactersistics are duplicated and the copy is altered to place the terminal into "raw" mode. *tcsetattr()* is called with {TCSANOW} to set the altered characters.

stopme()

```
void stopme(pid)
```

Args:

pid the pid of the process to suppend, zero (0) for all processes in the group

tcsetattr() is called with {TCSANOW} and the original terminal characteristics to reset the terminal. The SIGTSTP signal is sent to the supplied process (or group if 0). When the process resumes, *settermraw()* is called to return the terminal to raw mode.

reader()

```
int reader(int socket)
```

Args:

socketconnection to the job.

Returns:

Zero (0) on EOF, -1 on error.

This routine reads data from the job over the network and writes to the local terminal (standard output) in raw mode. It loops until either the connection is closed, EOF received, or an error occurs. See the figure 8−1 for a picture of the communication flow between qsub, pbs_mom's children, and the job.

writer()

```
int writer(int socket)
```

Args:

socketconnection to the job.

Returns:

Zero (0) on EOF, -1 on error.

This routine reads from the local terminal (standard input) in raw mode and writes data to the job over the network. It loops until either the connection is closed, EOF received, or an error occurs.

getwinsize()

```
int getwinsize(struct winsize *size)
```

Args:

size pointer to the window size structure, see sys/tty.h or termios.h or some such header.

Returns:

zero (0) if ok, -1 on error.

Gets the current window size by calling *ioctl()* {TIOCGWINSZ}

send_winsize()

```
void send_winsize(int socket)
```

Args:

socketconnection to the job.

Encodes the window size information obtained in a prior call to *getwinsize()* into a string and writes it to the job.

send_term()

```
void send_term(int socket)
```

Args:

socketconnection to the job.

Gets the TERM environment variable and writes it as a string `TERM=`type to the job. Also writes the {PBS_TERM_CCA} (number of) terminal control characters obtained earlier to the job.

catchchild()

```
void catchchild()
```

Signal handler for SIGCHLD. Invoked by death of the reader process. Resets the terminal to normal (cooked) mode.

catchint()

```
void catchint()
```

Signal handler for SIGINT and SIGTERM while qsub is waiting for the job to start. The function asked the user if it should terminate or not. If any string starting with 'y' is received, *pbs_deljob()* is called to delete the job and qsub exits.

interactive()

```
void interactive()
```

This routine waits for the job to connect with it over the socket set up earlier, see *interactive_port()*. The routine *catchint()* is installed as the signal handler for SIGINT and SIGTERM. The current terminal settings are saved by calling *tcgetattr()*. The window size is obtained by calling *getwinsize()*. *select()* is called in a loop with a 30 second time out. After each time out, *locate_job()* is called to see if the job still exists. If the job is gone, qsub exits.

When a connection request is received, the function reads in what should be the job id from MOM. If the string does not match the job id as submitted, qsub aborts. If the job id is correct, we send the termal type, *send_term()* and window size *send_winsize()*. We print that the job is ready and set SIGINT and SIGTERM handler to the default. A child process is forked to become the *reader()* process. The parent (qsub) becomes the *writer()* process. Both processes exit when EOF or an error is received. The writer process will make sure the reader child is killed and resets the terminal.

2.13.3.10.

process_opts()

```
int process_opts(int argc, char **argv, int pass)
```

Args:

argc	The number of arguments in argv.
argv	The command line or PBS directives line arguments.
pass	Zero, if a command line argument list, positive if a PBS directive argument list.

Control Flow:
If pass is greater than zero Then
    Start at the beginning of the argument list
While getopt Do
    The appropriate thing for each option

Note that the following rules are enforced:

1. Option argument values supplied on the command line take precedence over values for the same option supplied in script directivies.
2. If an option is repeated on the command line (or in the script, subject to rules 1), the argument value for the last occurrence:
   - replaces the prior value if the option is singled valued (integer or string).
   - is appended to the prior value(s) if the option is list valued (comma separated elements).

set_opt_defaults()

```
void set_opt_defaults()
```

This function is called after all command line options and script directives have been parsed. According to POSIX, certain job attributes must be set to default values if not set by the user. This is where that happens for: checkpoint, hold, join, keep, mail-points, priority, and rerunable.

2.14. Libcmds

The Libcmds library has supporting routines for the PBS utilities. These mostly consist of parsing job identifiers and destinations, building attribute lists, and some miscellaneous routines.

2.14.1. File: ck_job_name.c

This file has one routine to validate the job name specified vi the -N option.

2.14.1.1.

ck_job_name()

```
int ck_job_name.c(char *name, int alpha)
```

Args:

nameof job specified on -N option.

alphaIf set to one (1), the first character of the name must start with a alphabetic character. Set to zero (0), this check is not made.

Returns:

Zero (0) if name is valid, -1 if invalid.

The name must be less than or equal to 15 characters in length. The first character must be alphabetic if *alpha* is set to 1. PBS allows the remaining characters to be any printable character. The POSIX Batch standard calls for only alphanumeric, but then conflicts with itself to default to the script base-name which may have non-alphanumeric characters. Since the users like to use under_score and dot, we allow it.

2.14.2. File: cvtdate.c

This file has one routine to convert POSIX touch date/time to seconds since epoch time.

2.14.2.1.

cvtdate()

```
time_t cvtdate(char *datestr)
```

Args:

datestr The string **datestr** is a date/time string in the form [[CC]YY]MMDDhhmm[.SS] as defined by POSIX, where

CC = century, ie 19 or 20
YY = year, if CC is not provided and YY is < 69,
then CC is assumed to be 20, else CC is 19.
MM = Month, [1,12]

DD = Day of month, [1,31];
hh = hour, [00, 23]
mm = minute, [00, 59]
SS = seconds, [00, 59]

Returns:

Seconds since epoch, or -1 if an error occurred (see man mktime).

Control Flow:

```
If datestr contains a '.' Then
        Set the seconds from SS
        Take the .SS off datestr
else
        Set seconds to zero
Make sure the rest of the datestr contains all digits
Get the current year
Case length of datestr is
12:     /* CCYYMMDDhhmm */
        Get the century
        Fall through
10:     /* YYMMDDhhmm */
        Get the year
        If century is not set Then
                Set century according to the year
        Combine the century and year together
        Set the year
        Fall Through
8:      /* MMDDhhmm */
        Set the month from the next two digits
        Set the day from the next two digits
        Set the hour from the next two digits
        Set the minutes from the next two digits
default:
        Return -1
Return the result of mktime using the above data
```

2.14.3. File: get_server.c

This file has one routine to parse the job identifier from the command line into a full job identifier and a server name. The complete syntax for a command line job identifier is defined in parse_jobid.c. The routine implements the procedure outlined in Section 5.1.2 of the ERS for setting the name of the server.

2.14.3.1.

get_server()

```
int get_server(char *job_id_in, char *job_id_out, char *server_out)
```

Args:

job_id_inThe job identifier from the command line.

job_id_out
The sequence number and an appropriate server name.

server_out
The name of the server to send the request to.

Returns:
Zero, if the job identifier parse correctly, one otherwise.

Control Flow:
Parse the job_id_in into sequence number, parent server, and current server
If current server is defined Then
Set server_out to the current server (5.1.2 Step 1)
Elseif parent server is defined Then
Set the server_out to the parent server (5.1.2 Step 2)
Else
Set the server_out to NULL (5.1.2 Step 4)
If parent server is defined Then
Set the job_id_out to the sequence number and the fully qualified parent server
Else
Set the parent server to the environment variable PBS_DEFAULT value
If no value exist Then
Set the parent server to the name in the PBS DEFAULT FILE
Set the job_id_out to the sequence number and the fully qualified parent server
Note that the routine *get_fullhostname()* in libnet.a is used to obtain the fully qualified hostname.

2.14.4. File: locate_job.c

This file has one routine that connects to the server the job was submitted to and sends a Locate Job request. The result should be the server the job is at.

2.14.4.1.

locate_job()

```
int locate_job(char *job_id, char *parent_server, char *located_server)
```

Args:
job_id The full job identifier.
parent_server
The name of the server to send the request to.
located_server
The name of the server the job is current at.

Returns:
True, if the job is located, false otherwise.

Control Flow:
Connect to the parent server
Send the Locate Job request
Disconnect from the server

2.14.5. File: parse_destid.c

This file has one routine that parses the destination from the command line. The destination can have the following forms:

queue_name[@server_name[:port_number]]
@server_name[:port_number]

2.14.5.1.

parse_destination_id()

```
int parse_destination_id(char *destination, char **queue, char **server)
```

Args:

destination
: The destination to be parsed.

queue
: The queue part of the destination.

located_server
: The server part of the destination.

Returns:

Zero, if the destination was parsed correctly, one otherwise.

Control Flow:

Initialize the queue and server names to NULL
Get the queue name if it is given
Get the server name if it is given

2.14.6. File: parse_equal.c

This file has one routine that parses set of comma delimited name = value_list into separate name = value_list. On the first call, the first name = value_list is returned. On subsequent calls, the next name = value_list is returned until there are no more.

2.14.6.1.

parse_equal_string()

```
int parse_equal_string(char *start, char **name, char **value)
```

Args:

start
: Where to start parsing.

queue
: The name part.

value
: The value_list part.

Returns:

One, if a name = value_list is found, zero if nothing is left, and minus one if there is a parsing error.

Control Flow:
If there is nothing left to parse Then
return 0
Find the beginning of the name
Find the end of the name
Make sure it is followed by an '='
If value starts with a quote Then
Find the matching quote
Scan for the next equal sign
If at end of input string Then
return 1
Back up to the first comma
If the comma is after the start of the value string Then
Strip off trailing blanks

2.14.7. File: parse_jobid.c

This file has one routine that parses the complete job identification from the command line into it's various parts. The complete syntax is

seq_number[.parent_server[:port]][@current_server[:port]]

The routine returns the sequence number, parent server and current server as separate values. The port is returned as part of the server name.

2.14.7.1.

parse_jobid()

```
int parse_jobid(char *jobid, char **s_number, char **p_server, char **c_server)
```

Args:
jobid The job identification from the command line.
s_number
The sequence number part.
p_server The parent server part.
c_server The current server part.

Returns:
Zero, if parse was okay, non-zero on a parsing error.

Control Flow:
Initialize the sequence number, parent server and current server names to NULL
Scan for the sequence number
If the next character is '.' Then
Scan for the parent server
If the next character is '@' Then
Scan for the current server
If we are at the end of the jobid Then
Return the separate values

2.14.8. File: prepare_path.c

This file has one routine that path given on the command line and turns it into a full path name if needed. The syntax of the path name is

host:path

2.14.8.1.

prepare_path()

```
int prepare_path(char *path_in, char *path_out)
```

Args:

path_in The path to turn into an absolute path name.

path_outThe absolute path name.

Returns:

Zero, if path was converted okay, non-zero not.

Control Flow:

Initialize the host and path to NULL
Scan for the host name
Scan for the path
Get fully qualified host name
Put the fully qualified host name in path_out
Append a ':'
If the path name is relative Then
    Get the current working directory
    Append it to path_out
Append the path to path_out

Note, if the the path is relative and the current directory is in an NFS Automounted path, the currently location may not be mounted when the output is returned. Hence it is neccessary to make the path NFS Automounter "friendly". This entails finding the path name used to cause the automounter to mount. This might be obtained from the shell environment variable PWD if it exists. Otherwise we fall back to using getcwd() to expand the relative path name.

2.14.9. File: prt_job_err.c

This file has one routine that prints a standard error message if a request that involves a job identification fails.

2.14.9.1.

prt_job_err()

```
void prt_job_err(char *cmd, int connection, char *jobid)
```

Args:

cmd The PBS utility that is calling this routine.

connection
The connection identifier to the server.

jobid The job identifier of the failed request.

Control Flow:
If error message returned by server Then
Print server error message
Else
Print generic error message

2.14.10. File: set_attr.c

This file has one routine that builds an attribute list.

2.14.10.1.

set_attr()

```
void set_attr(struct attrl **attrib, char *name, char *value)
```

Args:

attrib The attribute list.

connection
The name part.

jobid The value part.

Control Flow:
Allocate the space needed for an attribute structure
Allocate the space needed for the name field and copy the name into it
Set the resource field to NULL
Allocate the space needed for the value field and copy the value into it
Append the attribute structure to the end of the attribute list

2.14.11. File: set_resources.c

This file has one routine that parses the resource list and makes an attribute structure from this that it appends to the attribute list. The syntax of a resource list is

resource = value, ...

2.14.11.1.

set_resources()

```
void set_resources(struct attrl **attrib, char *resources, int add)
```

Args:

attrib The attribute list.

resourcesThe resource list.

add Force the append or only add if the resource is not already on the attribute list.

Control Flow:

```
While the resource list is not empty Do
    Get the resource
    If followed by an '=' Then
        Get the value
    Allocate memory for the attribute structure
    Allocate memory for the name ATTR_l and copy it to the name
    Allocate memory for the resource name and copy the resource to it
    If value is defined Then
        Allocate memory for the value name and copy the value to it
    Else
        Set the value name to NULL
    If the attribute list is empty Then
        Put the attribute structe on it
    Else
        Search the attribute list to see if the resource is there
        If add is true or not found Then
            Append the attribute structure to the list
```

3. Operator Commands

A batch operator is a user of the system who is granted privilege to perform actions beyond those available to the normal user. Typical of those actions are starting and stopping queues and the server, and modification or deleting of jobs of other users.

In addition to the general user commands, the operator has access to the **qdisable, qenable, qinit, qrun, qstart, qstop,** and **qterm** commands.

3.1. File: qdisable.c

The **qdisable** command directs that a destination should no longer accept batch jobs.

3.1.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

destination ...

A *destination* is in one of the following forms:

queue
@server
queue@server

Returns:

None

Control Flow:

for each argument
parse(in:argument, out:queue, server)
execute(in:queue, server)

3.1.1.1.

parse()

```
int parse(char *destination, char *queue, char *server)
```

Args:

destination The destination queue in the form queue, @server, or queue@server.

queue The queue part of the destination.

server The server part of the destination.

Returns:

int Zero, if no errors occurred. One, if a syntax error occurred.

Control Flow:

Anything before the @ is the queue name

Anything after the @ is the server name
If (null argument) Then set error return

3.1.1.2.

execute()

```
int execute(char *queue, char *server)
```

Args:

queue	The queue name.
server	The server name.

Returns:
None

Control Flow:
Connect to the server
Disable the queue
Disconnect from the server

3.2. File: qenable.c

The **qenable** command directs that a destination should accept batch jobs.

3.2.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

destination ...

A *destination* is in one of the following forms:

queue
@server
queue@server

Returns:
None

Control Flow:
for each argument
parse(in:argument, out:queue, server)
execute(in:queue, server)

3.2.1.1.

parse()

```
int parse(char *destination, char *queue, char *server)
```

Args:

destination	The destination queue in the form queue, @server, or queue@server.
queue	The queue part of the destination.
server	The server part of the destination.

Returns:

int	Zero, if no errors occurred. One, if a syntax error occurred.

Control Flow:

Anything before the @ is the queue name
Anything after the @ is the server name
If (null argument) Then set error return

3.2.1.2.

execute()

```
int execute(char *queue, char *server)
```

Args:

queue	The queue name.
server	The server name.

Returns:

None

Control Flow:

Connect to the server
Enable the queue
Disconnect from the server

3.3. File: qinit.c

The **qinit** command starts the operation of the server.

3.3.1.

main()

```
main(int argc, char **argv)
```

Args:

argc	The number of arguments on the command line.
argv	The **argv** array contain the following arguments:

-t type
The type of initialization of the server: hot, warm, cold, clean, create.

-d config_path
The directory path which is the home of the configuration files.

server_path
The path name of the server to execute.

Returns:
None

Control Flow:
get the arguments with getopt
get the server path
fork
exec the server

3.4. File: qrun.c

The **qrun** command forces the server to execute a batch job.

3.4.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

job_identifier ...
The list of jobs to have the server run of the form:
sequence_number[.server_name][@server]

Returns:
None

Control Flow:
for each argument
parse(in:argument, out:job, server, location)
execute(in:job, server, location)

3.4.1.1.

parse()

```
int parse(char *identifier, char *job, char *server, char *location)
```

Args:

identifier The identifier in the form sequence_number[.server_name][@server].

job The sequence_number part of the identifier.

server The server part of the identifier that owns the job now.

location The location part of the identifier to the server which will run the job.

Returns:

int Zero, if no errors occurred. One, if a syntax error occurred.

Control Flow:

Anything before the . is the job
Anything after the . and before the @ is the server name
Anything after the @ is the location
If (no job number) Then set error return

3.4.1.2.

execute()

```
int execute(char *job, char *server, char *location)
```

Args:

job The job number.

server The server name.

location The location to run the job at.

Returns:

None

Control Flow:

Connect to the server
Run the job
Disconnect from the server

3.5. File: qstart.c

The **qstart** command directs that a destination should process batch jobs.

3.5.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

destination ...

A *destination* is in one of the following forms:

queue
@server
queue@server

Returns:

None

Control Flow:
for each argument
parse(in:argument, out:queue, server)
execute(in:queue, server)

3.5.1.1.

parse()

```
int parse(char *destination, char *queue, char *server)
```

Args:

destination	The destination queue in the form queue, @server, or queue@server.
queue	The queue part of the destination.
server	The server part of the destination.

Returns:

int	Zero, if no errors occurred. One, if a syntax error occurred.

Control Flow:
Anything before the @ is the queue name
Anything after the @ is the server name
If (null argument) Then set error return

3.5.1.2.

execute()

```
int execute(char *queue, char *server)
```

Args:

queue	The queue name.
server	The server name.

Returns:
None

Control Flow:
Connect to the server
Start the queue
Disconnect from the server

3.6. File: qstop.c

The **qstop** command directs that a destination should stop processing batch jobs.

3.6.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

destination ...

A *destination* is in one of the following forms:

queue
@server
queue@server

Returns:

None

Control Flow:

for each argument
parse(in:argument, out:queue, server)
execute(in:queue, server)

3.6.1.1.

parse()

```
int parse(char *destination, char *queue, char *server)
```

Args:

destination	The destination queue in the form queue, @server, or queue@server.
queue	The queue part of the destination.
server	The server part of the destination.

Returns:

int Zero, if no errors occurred. One, if a syntax error occurred.

Control Flow:

Anything before the @ is the queue name
Anything after the @ is the server name
If (null argument) Then set error return

3.6.1.2.

execute()

```
int execute(char *queue, char *server)
```

Args:

queue	The queue name.
server	The server name.

Returns:

None

Control Flow:
Connect to the server
Stop the queue
Disconnect from the server

3.7. File: qterm.c

The **qterm** command terminates the server.

3.7.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:

-t type
The type of termination of the server: immediater, delay, or quick.

server ...
The list of servers to terminate.

Returns:
None

Control Flow:
get the arguments with getopt
get the server names
for each server
execute(in:type, server)

3.7.1.1.

execute()

```
int execute(int type, char *server)
```

Args:

type The type of termination.

server The server name.

Returns:
None

Control Flow:
Connect to the server
Stop the server
Disconnect from the server

4. Administrator Commands

A batch administrator is a user of the system who is granted the highest level of privilege in the batch system. The batch administrator is able to perform all operator functions and additionally modify queue and server configurations.

In addition to the general user commands and operator commands, the administrator has access to the **qmgr** command.

4.1. File: qmgr.c

The **qmgr** command provides an administrator interface to the batch system. The command reads directives from standard input or from the command line. The syntax of each directive is checked and the appropriate request is sent to the one or more batch servers.

4.1.1.

main()

```
main(int argc, char **argv)
```

Args:

argc The number of options on the command line.

argv The **argv** array contains the following options:

-a Abort **qmgr** on any syntax errors or any requests rejected by a server.

-c command
Execute a single command and exit.

-e Echo all commands to standard output.

-n No commands are executed, syntax checking only is performed.

-z No errors are written to standard error.

server ...
The list of servers to manage.

Returns:
Nothing

Main is controls the flow for the entire program. It first parses the arguments with help from the getops(3) call. If there were servers passed in on the command line, the rest of the command line arguments are passed through *strings2objname()* to convert then into objname structures. If no servers are on the command line *default_server_name()* is called to get the default server objname struct. The objname struct is passed into *connect_servers()* to connect to the servers and set the servers global variable. These servers are set to be the active servers. If there was an error and the "-a" flag was given, qmgr exits. At this point is the important part of the main function. The following can be done in a loop depending on if the "-c" flag was given. First, if the "-e" flag was set. If it was, print the command. Next, the *parse()* function is called to parse the command. If the there was an error and the "-a" flag was set qmgr exits. Finally if "-n" was not set and parse did not return an error *execute()* is called to package up the command and send it to the server. If the "-c" flag is not given, it will loop on the *get_request()* function until an EOF is returned. Lastly it disconnects from all the servers and exits.

4.1.1.1.

get_request()

```
int get_request(char *request)
```

Args:

request	OUT: buffer for the string passed by reference

Returns:

int	Zero, if a request was found. EOF, otherwise.

There are two while loops are do most of the work in this function.

The first takes care of getting line from standard input. It will remove whitespace and new-lines. It will also ignore comments, and concatinte continuation lines ending with(. Note, this loop could be skipped if a command seperator was used the last time through. There is still more to be executed.

The second while loop copies the string into the request buffer passed in by the caller. It copies the string character by character handling special cases when they arise. It will end the command if it sees a command seperator(;), start of a comment(#), or null byte. If it encounters a quote (" or ') it will copy the string until it finds another quote of the same type.

The function will make one last check: Is there any more on the line? If the command has ended in a commanx seperator(;), and there is more on the line then just white space, it will set the empty flag to false. If the line ended with a comment or a null byte, the empty flag is set to true. Lastly, the rest (could be all) of the static buffer is filled with null bytes.

4.1.1.2.

parse()

```
int parse(char *request, int *operation, int *type, char **names, struct attropl *attribut
```

Args:

request	One entire qmgr request.
oper	OUT: parsed operator (active / create / delete / set / unset / list)
type	OUT: parsed type of object (server / queue / node)
names	OUT: parsed names of object
attributes	OUT: parsed list of attributes

Returns:

int	Zero, if there are no syntax errors. Non-zero, otherwise.

The parse function is what takes the input line and parses out the necessary things to do the command. A call to *parse_request()* is made to parse out the command, object, and

possible name. The command is used to set the *oper* variable. The object is used to set the *type* variable. If there is a name it is passed to the function *is_attr()* to check and see if it is attribute. If it is, then back up what is returned by *parse_request()* and clear the IND_NAME field of the *req* req array. If it is not an attribute, it is passed into the function *check_list()* to check for errors in a comma seperated list of names. Next, the rest of the request is passed through the function *attributes()* which converts the attribute value pairs to attrl structures. Lastly, a little error checking is done.

4.1.1.2.1.

attributes()

```
int attributes( char *attrs, struct attropl **attrlist, int doper )
```

Args:

attrs	The text of the attributes to parse
attrlist	OUT: the attropl structure to return the parsed attributes in
doper	The operation being done (active/create/delete/set/unset/list)

Returns:

0: no syntax errors

index:the index into the attrs var of where the error occurred otherwise.

The form of the attributes is either
attr OP value
or attr.res OP value
where OP is either = += or -=

freeattropl() is called right off to free the space of previous structures. A forever loop starts here. The first thing that happens in the find the name of the attribute. A attropl struct is created, and space for the attribute name is also allocated and assigned. If the attrs string is currently at a period(.), then the attribute is a resource. Space is allocated and the text of the resource is saved in it. The operator is found and set. If the operator is a comma goto the end of the loop to check for more attriburtes. The value is the last thing to find. If the value is quoted, find the other side of the quote and allocate and assign the string. If it is not quoted, look forward to find a comma or the end of the line and allocate and assign the string.

The last thing is the look if there are any more attributes to to parse. (Remember that goto a second ago... this is where it went)

4.1.1.3.

execute()

```
int execute(int aopt, int oper, int type, char *names, struct attropl *attribs)
```

Args:

aopt	If the -a option was on the command line. Abort on an error.
oper	The operation part of the request. (list/set/unset/create/delete)
type	The type of object from the request. (server/queue/node)
names	The names of the objects.
attribs	The list of attributes of the object.

Returns:

0 success

non-zero
on error

The first thing which is done is th convert the comma seperated list of names into a objname linked list. This is done by a call to *commalist2objname()* if the operation is to set active, call *set_actiive()* and return. Otherwise we need to loop through all the objects doing the right thing. If if the list of names was not passed it, the active objects are used. This starts the two main loops of the function. The outer loop is of the objects, and the inner loop is the servers. If the object has specified a server, the request will only be sent to that server. If it doesnt specify, it will be sent to all the active servers. The outer loop doesnt too anything but the inner loop. The first thing that happens in the inner loop is that it will check if it needs to connect to a server. This is where the meat of the function happens. If the operation is list or print, a pbs_stat* (server/queue/node) call is sent to the server and the result sent to display. Everything else is packaged up into a *pbs_manager()* and sent to the server. Lastly a little error checking is doen so make sure there wasn't an error and then the error, if any is returned.

4.1.1.3.1.

commalist2objname

```
int commalist2objname(char *names, int type)
```

Args:

names	A comma seperated list of names or NULL
type	The type of the object

Returns:

pointer to array of names

This function will take a list of comma seperated names and turn it into a linked list of objname strutures. There is one main loop in this function. We shoot forward in the string looking for either a comma or an at sign. We allocate an objname structure. If we found an at sign, we look for a comma we copy whats before the at sign into the object name, and after the comma into the server name. If we ended on a comma, we just need to copy the object name. If the type is a server, then assign the object name into the server name. After the loop we check for an error. Ifthere was one, clean up and return. If not, return the objname list.

4.1.1.3.2.
`struct attrl *attropl2attrl(struct attropl *from)`

Args:

from — an attropl list to be transformed into an attrl list

Returns:

Pointer
attrl list

This whole function revolves around a loop which goes through the entire attropl list in from. Mainly two things are done in this loop. The first is to allocate and initalize a new attrl struct in the new attrl list. The second is to copy the name, resource, and value fields from the current attropl struct into the new attrl struct. This ends the loop. Lastly, the new list will be returned.

4.1.1.3.3.

freeattrl()

`int freeattrl(struct attrl *attr)`

Args:

attr — The list of attrl structures to be freed

Returns:
Nothing

March through the attrl list freeing all the inner variables, and finally the structure its self.

4.1.1.3.4.

display()

`int display(int otype, char *oname, struct batch_status *status, int format)`

Args:

otype — The type of the object
oname — The name of the object
status — The list of status structures to display.
format — True, if the output is to be formatted as input.

Returns:
Nothing

The output of this function will depend on the format variable. If it is true, the output of the function could be used as input into the qmgr. If it is false, the output will be easier to read output for information. Throughout the function there are checks to print either method depending on the format variable. There is a series of while loops since the batch_status structure is a linked list of objects which contain a linked list of attributes. The outer while loop is to go through the batch_status structs. The inner one is for attributes.

Everything is the same for the two methods of output until you get to printing attributes whose values have multiple values (i.e. managers=root,bob,susan). If format is true, the multiple values will be seperated into multiple lines while using the addition operator(i.e. managers = root ; managers += bob...). If format is false, the multiple values are line wrapped. Sample output:
Set server attributes. # set server scheduling = True set server max_running = 3 set server managers = root set server managers += bob set server managers += susan set server resources_max.mem = 128mb set server resources_available.mem = 100mb set server scheduler_iteration = 120

4.1.1.4.

clean_up_and_exit()

```
int clean_up_and_exit(int extflg)
```

Args:

extflg the exit value

Returns:

This function never returns, its exits.

Free the active object lists. Then call *pbs_disconnect()* on each of the open servers. Finally exit with exit_val return code.

make_connection()

```
int make_connection( char *name, struct server *svr )
```

Args:

name

the name of the server to connect

Returns:

Pointer to newly connected server

A connection attempt is made through the *cnt2server()* library call. If it is successful, a new server struct is allocated and the the server fields s_name and s_connect are assigned. If it fails print an error.

connect_servers()

```
int connect_servers( char **server_names, int numservers )
```

Args:

server_names
: array of server names to make connections

numservers
: the number of servers to connect to on the list

Returns

TRUE
: on error

FALSE
: NOTE: This function modifies the *servers* global variable.

First of all, nonreferenced servers are closed. If the ammount of open servers is still under the max ammount of servers that can be open, then run through a forloop from 1 .. numervers and call *make_connection()* to each one. The objname svr field is set.

blanks()

```
void blanks( int number )
```

Args:

number
: the number of spaces to print

Returns

Nothing

print number spaces to standard error by filling a buffer with spaces and printing them.

check_list()

```
int check_list( char *list )
```

Args:

list
: A comma delemited list

Returns

0
If the syntax is correct

>
If the index into the string where the error occured

This function checks for validity of a comma seperated list. There is one main loop which checks for eronious conditions and returns the index into the string where it happened.

Case 1:
First char is not not alpha or an '@' Ex Good: "ueue" Ex Fail: "!queue"

Case 2:
error situation with an "@" Ex Good: "name@svr" Ex Fail: "name@," "name@"

Case 3:
After a name with an "@" if doesnt end with a "," or EOL Ex Good: "name@svr,name" Ex Fail: "name@svr@name"

Case 4:
a comma at the end of the line. Ex Good: "name@svr" Ex Fail: "name,"

freeattropl()

```
void freeattropl(struct attropl *attr)
```

Args:

attr
A pointer to a linked list of attropl structs to free

Returns
Nothing

Loop through the attropl list freeing the members and finally freeing the structure itsself.

is_attr()

```
int is_attr( int object, char *name, int attr_type )
```

Args:

object
The type of object

name
The name of the attribute

attr_type
The type of the attribute: Public or Readonly

Returns

TRUE
if the attribute is public

FALSE
if not

There are six attribute arrays which are are initialized with the contents of header files. They are used to see if the name is an attribute or not. The object type and attribute type is checked to see what arrays to use. The check is doen by iterating through the arrays.

pstderr()

```
static void pstderr( char * )
```

Args:

string
the string to print to stderr

Returns
Nothing

If the global variable zopt is false, then print to standard error string. else do nothing

pstderr1()

```
void pstderr( char *string, char *arg )
```

Args:

string
The format string

Arg
Argument

Returns
Nothing

This function is like *pstderr()* but instead of just printing a string, it will print a string and one string argument. It uses fprintf().

show_help()

```
void show_help( char *str )
```

Args:

str
Possible subject to get help on

Returns:
Nothing

if the string is NULL or a null byte, print basic help. If there is a string, print more specific help.

find_server()

```
struct server * find_server( char *name )
```

Args:

name
Then name of the server to find

Returns:
The server structure if found or NULL of not

Basically loop though all of the servers in the global variable servers. If the server is found, return it. If not return NULL;

new_server()

```
struct server *new_server()
```

Args:None

Returns:
Newly allocated server structure

Allocates a new server structure and initializes it.

new_objname()

```
struct objname *new_objname()
```

Args:None

Returns:
newly allocated objname structure

Allocates a new objname structure and initializes it.

strings2objname()

```
struct objname *strings2objname( char **str, int num, int type )
```

Args:

str
the array of strings

num
The number of strings in the array

type
The type of objects

Returns:
objname linked list

Loop through all the elements in the array and create a linked list of objnames. If the type is a server set the svr_name to the obj_name field.

default_server_name()

```
struct objname * default_server_name()
```

Returns
objname structure with default server information

If *pbs_connect()* is passed a null string(""), it will open a connection to the server specified in the pbs default file. This function will create an objname structure and fill it with the correct information for the default server.

temp_objname()

```
struct objname *temp_objname( char *obj_name, char *svr_name,
                        struct server *svr)
```

Args
obj_name
The name for the temp obj
svr_name
The server name for the temp obj
svr
The server for the temp obj

Returns
The temporary object

This function has a static struct objname which it will use. It clears all data from the objname, and then assigns in the new data. It will adjust the reference counts on the new and old servers as necessary.

parse_request()

```
int parse_request(char *request, char req[][] )
```

Args
request
the request line to parse
req
OUT: The array to assign

Returns
length of the request line parsed or zero on error

Parse out the first three words of the request. The first word should be the command, the second the object, and the third will be a name. There are five symbolic constants used to help with this array. IND_FIRST is the first index of the array. IND_LAST is

the last index. We then have IND_CMD, IND_OBJ, and IND_NAME. These are for command, object and name indicies of the array.

free_objname_list

```
void free_objname_list( struct objname *list );
```

Args:

list
the objname list to free

Returns:
Nothing

Free all the objnames by looping through the linked list calling *free_objname()*

free_server()

```
void free_server( struct server *svr )
```

Args

svr
The server to free

Returns
Nothing

This function will first attempt to remove the server from the servers list. If it can find the server, it will unlink it from the servers list. Reguardless, it will free the memory used by the structure.

free_objname()

```
void free_objname( struct objname *obj )
```

Args:

list
the objname to free

Returns
Nothing

Free the memory used by the objname and decrement the server which it referenced.

close_non_ref_servers()

```
void close_non_ref_servers()
```

Returns
Nothing

This function goes through the server linked list and will close connections to servers with a zero reference count. This is done by calling *disconnect_from_server().*

set_active()

```
int set_active( int obj_type, struct objname *obj_names )
```

Args

obj_type
The type of object we are setting the active list

obj_names
the objname linked list to set active to

Returns
0 on success / non-zero on failure

This function will set the active servers, or print the active object depending on whether obj_names is NULL. If it is not NULL then the active object will be set. If its a call to set the active servers, then each server is connected to if needed. Otherwise a call to *is_valid_object()* is made to see if the object is exists and is valid.

is_valid_object()

```
int is_valid_object( struct objname *obj, int type )
```

Args

obj
the object to check

type
The type of object

Returns
1 if the object is valid, 0 if not

If the type is queue, a *pbs_statque()* is done to check for the existence of the queue. It acually queries queue type, but nothing is done with this value. This is because something has to be queried, or everything is returned. Simularly with nodes the state of the node is querried for the same reason. If the call is successful, the object exists and is valid. The other case of a valid object is if obj is NULL. A null object means any requests will be sent to all active servers.

disconnect_from_server()

```
void disconnect_from_server( struct server *svr )
```

Args

svr
The server to disconnect from

Returns
Nothing

call *pbs_disconnect()* on the connection descriptor for the server, and make a call to *free_server()*. Lastly

[This page is blank.]

5. The Batch Server

5.1. Server Overview

The batch server is the heart of the batch processing system. There is typically one server per main processing host. Additional servers may exist for testing or special purposes. Also, a single server may be configured to support a cluster of processing nodes.

The batch server has the following responsibilities:

- Own and manage batch jobs.
- Own and manage queues.
- Recover state of jobs and queues upon restart of batch server.
- Perform services on behalf of clients based on batch service requests.
- Perform deferred services on behalf of jobs based on external events (changes in environment, resources, etc.) or time.
- Initiate selection of jobs for execution based on a set of site defined policy rules.
- Establish resource reservations and usage limits for jobs being placed into execution.
- Place a batch job into execution and monitor its progress.
- Perform post job execution processing and clean-up.

5.1.1. Server Objects and Attributes

With apologies to Lewis Carroll...

> 'The time has come,' the Walrus said,
> 'To talk of many things:
> Of Queues - and Jobs - and Attributes -
> Of cabbages - and kings -
> And why the sea is boiling hot -
> And whether PBS has wings.'

To understand the design of the PBS server, it is necessary to understand the concepts behind server objects, like jobs and queue, and the object attributes. Three classes of objects exist within the server: jobs, queues, and the server itself. An instantiation of an object is represented by a structure and the data it contains. There is a separate structure for each object.

5.1.1.1. Job Objects

A job is a set of data about the job and the job script. The job data is maintained in a *job* structure which is defined in *job.h*. This information, along with the script, is also recorded on disk to prevent lose in case of a crash. The job data controls how the server deals with the job, the resources made available to the job during execution, and what happens to the standard output and standard error files of the job when it completes processing.

The job data can be divided in two groups, the fixed data and the attributes. The fixed data is typically private to the server or read-only to the client. This data is fixed in size and maintained in a sub-structure.

The client supplied/modifiable data is in the form of *attributes*. The ERS names these attributes and explains their purposes. The attributes of a job are defined in the file job_attr_def.c as an array of attribute definition structures, `attribute_def job_attr_def[]`. It is critical to maintain the ordering between the definitions in job_attr_def[] and the `enum`

job_atr defined in job.h. The values in this enum are used to index into the job attribute array.

5.1.1.2. Queue Objects

A queue is little more than a collection of jobs. There are attributes associated with a queue which control how the server deals with the queue. As with the job, the queue structure is recorded to disk to preserve it across crashes and shutdowns.

POSIX 1003.2d defined and PBS supports two basic queue types, execution and routing. Jobs remain in execution queues until they are run or aborted. Jobs in routing queues are to be moved to another queue. The destination may be a queue in the same server or in a remote server.

The queue attributes are defined in the file queue_attr_def.c as an array of attribute definition structures attribute_def que_attr_def[]. As with jobs, the ordering between the members of the array que_attr_def[] and the ''enum queueattr'' defined in queue.h must be maintained. The two types of queues have slightly different attributes. The que_attr_def array contains both sets. Only the attributes defined for the type of a queue are used with that queue.

5.1.1.3. The Server Object

The server itself is an object, a structure and a set of attributes which control various aspects of the servers operation. The server attributes are defined in the file svr_attr_def.c as the attribute definition array svr_attr_def[] Again it is critical to maintain the ordering between the attributes and the ''enum srv_atr'' defined in server.h .

5.1.1.4. Just What are Attributes?

The concept of an attribute in PBS provides it with much of flexibility and power of PBS. In one sense, an attribute is just another data item, a element of the parent objects structure. What sets attributes apart is the two tier representation of an attribute and the encapsulation of the data and its associated functions.

An attribute is represented by its name and its value. Exterior to the server, an attribute is seen as a pair of character strings, one for the attribute name and one for its value. Internally, attributes are represented in one of two ways depending on if the meaning of the attribute is known to the program. If the meaning is unknow to the program (specifically a Job Server), the internal representation is very similiar to the external form. This will be discussed in more detail shortly. For those attributes whose meaning is known, i.e. there is code to do something with the value, the attribute is represented by two structures, the *attribute* structure (often referred to as the value), and the *attribute_def* structure. The attribute structure contains the actual attribute value in a machine dependent form. There is one attribute structure for each instance (occurrence) of an attribute.

The attribute_def structure contains the attribute name, flags, and pointers to the functions used to access and manipulate the attribute, see the section **Attribute Manipulation Functions** later in this chapter. There is one and only one attribute_def structure for each named attribute of an object. Attributes of the same data type (integer, character string, ...) may share the access functions. To add an attribute with a new name and new capability, it is only necessary to add the new definition structure and any access function which might be unique.

The attribute definitions exist in an array of attribute_def for each type of parent object. Attributes with the same name but different types and meaning *may* exist in different types of objects such as jobs and queues. However, this is *not* recommended for the confusion factor.

The attribute value is represented in a *attr_value* union within the *attribute* structure. This union contains all possible value data types, see attribute.h. It is assumed that any code needing the attribute value knows what type it is; however, that information is available in

the attribute_def. Some of the attribute value data types, (or more simply *attribute types*) require additional storage to hold the value. In these cases, the additional space is allocated and freed as required.

All possible attributes for the server and queues are known to the server by name. Any reference to an unknown attribute name for those objects is illegal. This is not true for jobs however. Since jobs may be "just passing through" to another server or the name and value may have meaning to the Job Scheduler. The meaning of such attributes are unknown to this specific Job Server. To handle this case, a special attribute, the unknown attribute, is created for jobs. Any unrecognized attribute for a job is maintained under the unknown attribute as a linked list of two strings, name and value, and a control or header structure. The strings and the control structure, which gives their lengths and the total storage required, are placed in a single allocated block of storage. This block can be easily saved to disk without any knowledge of the type. This form is known as the *svrattrl* structure (for server attribute list).

The svrattrl structure also acts to isolate the server from the actual form used for network encoding.

5.1.1.5. What are Resources

Up to this point, there have been a few scattered references to resources. So the time has come to describe what they are. The answer depends on whom you ask. POSIX 1003.2d defined a job attribute named *resource_list*. which has two meanings; and thus it exist in the PBS Server. The resource_list job attribute is actually a set of requirements of system resources needed by the job to execute and a set of limits to place on the usage by the job of thoses resources. For example, a job may need two tape drives to execute. Thus it would have a requirement in the resource_list of "tapes=2". A limit on the cpu usage of a job can be stated by a resource_list entry of "cput=10".

MOM will interpret the resource_list as limits. The scheduler sees the list as a list of job requirements, a slightly different view point. The resource monitor reports the availability of system resources to the scheduler. They have nothing to do with the job resource_list.

Within the Server, resources are treated as a special case of a job attribute. They are special in that they have multiple names and values and are in fact maintained by the server as a linked list headed by the attribute.

The resources for a batch complex managed by a Server are defined within the server. PBS supplies sets of resource definitions in the form of an array of resource definition structure, svr_resc_def[] defined in a series of files resc_def_*.c . There is a file for each target system supported by PBS. Additional resources may be added to the Server by inserting the appropriate definition in the correct file. However, code to process the resource will likely be required in MOM. Be sure to read the section on resource.h.

5.2. Packaging

The PBS Server is a single program which is run with root privilege as a daemon process. The source code for the server consists of the files in the directory *src/server*, many of the header files in *src/include*, and many of the libraries found under *src/lib/**. The descriptions of the server's routines are groups by the object on which they act or the general purpose of the function.

5.3. Program: pbs_server

5.3.1. Overview

The PBS Server is started by the **pbs_server**(8) command. The pbs_server command may be entered by a operator manually or it may be placed in boot time start up file (/etc/rc.local). Once the pbs_server has been started, it will:

1. Validate the server database structure (see pbsd_init).
2. Abort, requeue, restart, or reconnect to executing jobs depending on the initialization mode.
3. Initialize the network (and other interprocess communication) connections.
4. Begin to accept and process batch service requests and to perform deferred services.

The pbs_server will continue to perform services until it is terminated by the receipt of a shutdown request or a SIGTERM (or SIGSHUTDN) signal. The actions taken by pbs_server upon shutdown depend on the type of shutdown, delayed or immediate, see pbs_terminate(3) and qterm(8); but will always include updating the server's database.

5.3.2. External Interfaces

The pbs_server process has the following external interfaces:

- Arguments supplied on the command line.
- The server database which is described in the section ??.??.?? Server Database.
- The batch requests received over the network interface, described in the section 11.3 Protocols.
- The information available from the PBS Scheduler, described in the section 6.1.1 Scheduler/Server Communication and 8.1 MOM's Interpretation of PBS Protocol.

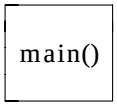
The following modules (source files) are part of the pbs_server.

5.3.3. Server Main Loop

The file pbsd_main.c in directory *src/server* contains the initial entry point for the pbs daemon, the code to interpret the arguments passed to the daemon, the call to initialize the internal data and state, the call to initialize the network interfaces, and the main process loop.

The server's main loop is event driven. The event types are the arrival of a batch service request, the arrival of a reply to a request made to another server or daemon, the arrival of a signal, and the expiration of some timed event. The server runs as a pseudo multi-threaded serial server. Unlike parallel servers which fork a child copy of itself for each service request, the PBS Server runs as a single program which processes all requests. This is to insure consistency of the internal data. There are two situations in which the server will fork, to send mail to a job owner and to send a job to another server. The latter may be time consuming and rather than handle the complexity, the server creates a child which sends the job. The server treats the send operations as atomic, it either successes or fails.

The pseudo multi-threaded comes from the method by which the server handles tasks which might result in a delay. For example when the serve sends a request to another server, rather than block waiting for the reply, the fact that a reply is expected, on which communication connection it is expected, and what function should process the reply is saved as an *event*. The arrival of the reply triggers the event processing. This is know as a "deferred reply" event. There are several other types of events, they are described in the routine *set_task()*.




```
main(int argc, char **argv)
```

Args:The *argv* array may contain the following options:

[-a true|false]
Sets the scheduling attribute.

[-d config_path]
Path of top level, see {PBS_DIR} in figure 5−1.

[-P dis_port]
Specifies the port on which the server listens for DIS encoded requests; must be numberic.

[-t type]
Initialization type

[-A account_file]
Specifies the absolute path to the accounting log file.

[-L log_file]
Specifies the absolute path to the general log file.

[-M port]
Specifies a port on which MOM should be contacted.

[-S port]
Specifies a port on which the scheduler should be contacted.

See **pbs_server**(8) for more detail on the -t and -d options.

Returns:
None.

Control flow:

```
Get local host name and default ports.
uses DIS_tcp_setup() to set tcp routines for DIS encoding
Process arguments, setting flags based on options.
Set log_event_mask and open the log file
Set up to ignore or catch signals.
Perform initialization processing based on type of initialization.
Initialize the network communications.

Begin the main processing loop.
   Process any ready work task event in the various work lists,
   see next_task().
   If the server is in state RUNNING,
      If the recovery type was RECOV_HOT,
         If more than SVR_HOT_CYCLE seconds have passed since last time,
            call start_hot_jobs()
         If more than SVR_HOT_LIMIT seconds have passed since server up,
            reset recovery type to RECOV_WARM to ignore hot jobs.
      If time or event to run scheduler and attribute Scheduling is true,
         call schedule_jobs()
      For each routing queue,
         call queue_route() to route jobs.
   Wait on arrival of batch service request.
   If a request arrived,
      Process request.
   Else if received a signal,
      If signal was death of child,
         Perform sub-server clean up processing.
      Else
         Shutdown the server.
   Continue with main processing loop.
```

Update all server databases.
respond to the shutdown request (if one).
Close network connections.
Log the final shutdown event.
Close the log.
Exit

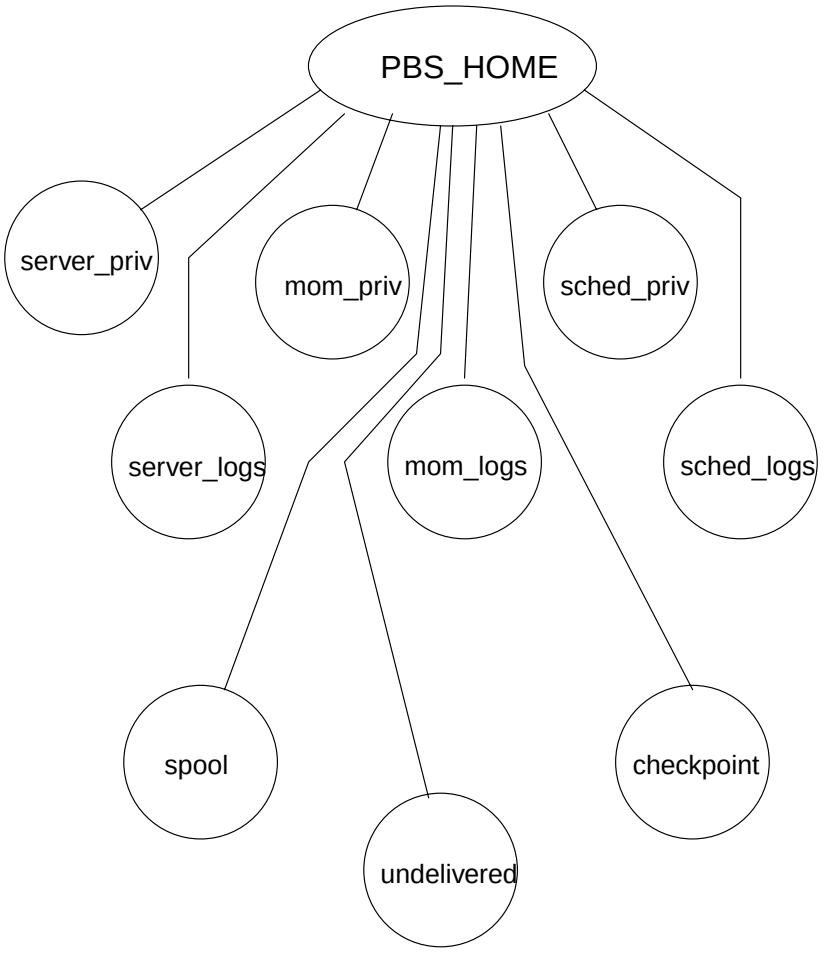


Figure 5−1: PBS Home Directory

next_task()

```
static time_t next_task (void);
```

Returns:

time_ttime to next event

This function scans the various work task lists and for any for which service is now required, calls *dispatch_task()* to invoke the processing routine. The lists are processed in the following order:

1. If the svr_delay_entry global variable is set non-zero, then the external event list, task_list_event, is scanned for events which have been changed to type {Immed}, see

catch_child() for details.

2. Any entry in the immediate list, task_list_immed, is dispatched.
3. If the event time of any entries in the timed list, task_list_timed, has been reached, they are dispatched.

If there is a need to run the job scheduler and scheduling is active, that is done by setting svr_do_schedule to {SCH_SCHEDULE_TIME}.

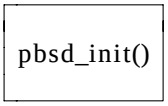
The least of (1) the time to the next timed action (if one), or (2) the time to the next scheduler run.


```
static void start_hot_jobs()
```

This routine is called in the main loop when the server recovery mode is {RECOV_HOT}. Its purpose is to restart jobs which were running when the server last went down. Each job owned by the server which (1) are in state {JOB_SUBSTATE_QUEUED}, and (2) have the {JOB_SVFLG_HOTSTART} flag set in *ji_svrflags* is placed into execution by calling *svr_startjob()*.

5.3.4. Server Initialization

The file pbsd_init.c in directory *src/server* contains the code to initialize the batch server. This code is called once when pbs_server begins execution. The actions performed depend on the type of initiation.




```
int pbsd_init(int type)
```

Args:

type The type of initialization

Returns:

0 If initialization is successful. Note, many internal tables have been loaded and the global server state has been changed.

non-zero
If initialization failed.

The sequence of events for the initialization is:

Catch the following signals: SIGHUP, SIGINT, SIGTERM, and SIGCHLD. Set up path names to various server directories and clear the head of server lists. Set the various default server attribute values, network retry time and force logging of all event types. Set the default log file name to the Julian day of the year.

If this initialization is not of type *create*, load the server attributes from database, see svr_recov().

Initialize server global data items, such as the name of this server, its network address and port, and MOM's address and port number.

Then, if not a *create* initialization, recover the queue attributes from the files in the queue directory. For each queue database file, call *que_recov()*.

If not a *create* or *clean* initization, recover the jobs from the save files in the jobs directory. Change the server's current working directory to the jobs directory. For each file with a name ending in the job suffix, **.JB**, recover the job information, by calling *job_recov()* and process the job to either re-queue or delete, see *pbsd_init_job()*. Report on number of jobs recovered.

If the queue rank number used to order jobs in the queue has gone negative, it is reset to zero and each job has its queue rank updated starting from one. This is to prevent overflow. While this should take a minimum of five years, PBS is such a great product, it is bound to run that long :-).

The job tracking records are recovered from their save file and reloaded into a tracking array. The array is allocated to whole the larger of the number of records in the save file or the minimum number of records {PBS_TRACK_MINSIZE}.

If the initialization type is *Cold* or *CREATE*, set the server attribute Idle to true.

build_path()

```
static char *build_path(parent, name, suffix)
```

Args:

parent
: the name of the parent directory, used as the prefix.

name
: the desired file name.

suffix
: the suffix to append or null.

Returns:

pointer
: to the name string.

The size of the path name is calculated and that amount of space is allocated. The parent directory name is copied into the allocated space. If the parent does not end in a slash, '/', one will be appended. Then the name and any suffix is appended.

pbsd_init_job()

```
static void pbsd_init_job(job *pjob, int type)
```

Args:

pjob
: Pointer to job structure to process.

type
: Initialization type.

This function is called by pbsd_init() for each job file found and recovered. The actions taken depend upon the state (substate) of the job at the time the server went down, and upon the initialization type.

If the initialization type is *clean*, then abort the job by calling *job_abt()*.

Otherwise, act according to the job substate. Unless otherwise noted, the route *pbsd_init_reque()* is called to requeue the job. For job in substate:

TRANSICM
: If the job was created here, the client was a temporary one not a server, then set sub-

state to QUEUED. Otherwise, hold on to the job in the new job list and wait for some server to send a commit.

TRANSOUT
Requeue the job as QUEUED.

TRANSOUTCM
We need to (re)send the "Ready to Commit" and "Commit" messages, however the net connection has not yet been initialized. So, requeue the job as is and establish a work task to finish sending the job.

QUEUED, PRESTAGEIN, STAGEIN, STAGECMP, STAGEFAIL, STAGEGO, HELD, SYNCHOLD, DEPNDOLD, WAITING, RUNNING, or STARTING
Requeue the job as is.

JOB_SUBSTATE_RESOURCE
Requeue the job in state JOB_STATE_QUEUED. It will need to look for its resources again.

JOB_SUBSTATE_SYNCRES
Clear all recorded ready" dependencies and requeue the job.

EXITING or STAGEOUT
Set a task entry to complete job the exit processing.

Any other
Abort the job.

pbsd_init_reque()

```
static void pbsd_init_reque(job *pjob, int change)
```

Args:

pjob Pointer to job structure.

change
flag to change or keep current job state.

This function is called by pbsd_init_job() to perform the enqueue. Messages about the requeuing are placed in the log file.

If the change flag is set to {CHANGE_STATE} (1), *svr_evaljobstate()* is called to determine to what the job state and substate should be set; *svr_setjobstate()* is called to set them. If change is {KEEP_STATE} (0), the job state and substate are unchanged.

Then *svr_enquejob()* is called to add the job to the queue.

catch_child()

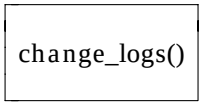
```
void catch_child(int sig)
```

Args:

sig The signal {SIGCHLD} which caused this signal handler to be invoked.

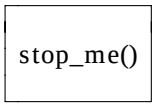
This function is the signal handler for SIGCHLD, death of child. Upon receipt of a SIGCHLD, a *waitpid()* system call is performed to collect the pid and exit status of any terminated child. The event work task list is searched for entry with a type of {Deferred_Child} and

an event id matching the child pid. If found, the exit status is saved in the entry and the entry type is changed to {Immed}. The global flag svr_delay_entry is updated to indicate that the main loop should search the delayed list for entries to be moved to the immediate list. We do this rather than immediately the process the entry to minimize the work performed in the signal handler and to prevent relinking the list when an interrupted function might already have been doing so.




```
static void change_logs()
```

This is the signal handler for SIGHUP. When a hup is received, the handler closes the accounting file calling *acct_close()* and reopens it calling *acct_open()* with *path_acct*. This allows the file to moved to a new name and restarted.




```
static void stop_me();
```

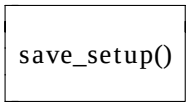
This is the signal handler for all signals which are to terminate the server.

The signal number is saved for a log_event() call which is made outside of the handler, and the server state is set to {SV_STATE_SHUTSIG}.

5.3.4.1. attr_recov.c

The file *src/server/attr_recov.c* contains the functions to write an array of attributes to a file and to restore the attributes from the file. The attributes of an object are saved whenever they are changed and when the server is shut down. This allows the server to recover its state when restarted.

When attributes are being saved, they are encoded into a list of *svrattrl* entries. This list is packed into a buffer by calling *save_struct()*. The buffer is only written whenever it becomes full. This saves I/O calls. On a recovery or restart, performance is not critical.

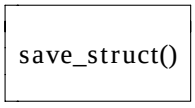



```
void save_setup(int fds)
```

Args:

fds The open file descriptor to which to write.

The file descriptor is squirreled away for calls to other functions in this file. The pointer into the buffer and the amounts of space used and available are initialized.




```
int save_struct(char *pobj, size_t objsize)
```

Args:

pobj A character pointer to a object (structure) to save. The object cannot have data that exists (is pointed to) outside of the object itself.

objsize
The size, in bytes, of the object. This amount of data is saved.

Returns:

0 If object is written to the file successfully.

-1 If an error occurs.

As much data as will currently fit into the "pack buffer" is copied from the object to the buffer. If not all of the object fits, the buffer is written and the pointer into it and the space available/used are reset.

save_flush()

```
int save_flush()
```

Returns:

0 on success

-1 on error

Any data which resides in the "pack buffer" is written to disk. The saved file descriptor is reset to a value to indicate the current save operation is complete.

Note, save_setup() must be called before attempting to pack any additional data. Also, it up to the caller to close the file descriptor if that is appropriate.

save_attr()

```
int save_attr( attribute_def *padef, attribute *pattr, int numattr)
```

Args:

padefPointer to the attribute definition structure, used to obtain the attribute name.

attr Pointer to the first attribute in the array to be saved.

numattr
The number of attributes in the array to save.

Returns:

0 if successful

-1 if error

Each attribute in the array (see below) is in turn encoded into a svrattrl entry using the *at_encode* routine for that attribute. The {ATR_VFLAG_MODIFY} is cleared to indicate the data has been saved. Then the svrattrl entry is packed into the output buffer by calling *save_struct()*. The entry is unlinked from the list and freed.

After the final attribute is encoded into the buffer, a dummy svrattrl entry with a size set to a magic number, {ENDATTRIBUTES}, is appended using save_struct(). This entry will be recognized by recov_attr() as indicating the end of the attributes has been reached.

Note, attributes of type {ATR_TYPE_ACL}, are ignored. These access control list attributes are not saved in the same matter as other attributes, see *save_acl()*.

recov_attr()

```
int recov_attr(int fd, void *parent, attribute_def *padef, attribute *pattr,
               int limit, int unknown)
```

Args:

fd The open file descriptor to read.

parent
Pointer to the parent object (structure) which contains the attributes.

padefThe attribute definition structures for these attributes.

pattr A pointer to the array of attributes which is being restored.

limit The number of attributes in the attribute array and attribute definition array, passed to find_attr().

unknown
If greater than zero, this is the index into the attribute definition array to use when the attribute does not match any known attributes, the attribute to use for "unknown" attributes. This is used only for jobs.

Returns:

0 if successful

-1 if error

Each attribute in turn, is reloaded in two reads, the first read gets the fixed size portion of the *svrattrl* structure itself, this gives the size of the encoded attribute. The second read obtains the variable portion containing the encoded strings. The attribute is identified using the name string and the *find_attr()* function.

If the attribute name does not match any in the definition array, either (1) the job is a transient job (in a routing queue) and has attributes that are not known here; or (2) the server has been rebuilt and the attributes changed. In case one, the attribute is saved in the attribute given by the *unknown* parameter. In case two, *unknown* will be zero, the event is logged, and the attribute ignored.

The attribute value is then passed to the appropriate decode function. If the attribute definition structure contains a non-null pointer to an action function (at_action), the action routine is called. The pointer to the parent structure is passed to the action routine along with the pointer to the attribute and the *action mode*. In this case, the action mode is set to {ATR_ACTION_RECOV}.

The loop is terminated when the total size (al_tsize) specified in the *svrattrl* structure is equal to the magic number {ENDATTRIBUTES}.

5.3.4.2. job_recov.c

The file *src/server/job_recov.c* contains the functions to save and recover (restore) a job structure and its associated sub structures and lists from a job file on disk.

job_save()

```
int job_save (job *pjob, int updatetype)
```

Args:

pjob Pointer to job structure which is to be saved.

updatetype
The type of save, quick, full update or new

Returns:

0 If save was successful.

-1 If error.

The job structure is saved to disk in a file whose name matches the job identifier. The save is one of two types: quick (mode = {SAVEJOB_QUICK}), or full (mode = {SAVEJOB_FULL} or mode = {SAVEJOB_NEW}).

If the *ji_modified* flag in the job structure is set indicating that one or more attributes have been modified, the {JOB_ATR_mtime} attribute is update to the current time. Note, this flag should be set any time a non Read-Only attribute is changed on behalf of a client.

A quick save is performed to record state and other internal data changes. Only the basic fixed length section of the job structure is re-recorded. A rewrite in place is performed. This minimizes the amount of I/O for a common type of save.

A full save is performed for a new job or whenever an attribute changes or an dependency is registered. This update records the basic job structure plus all of the variable length sub-structures and lists. The various pieces of the structure are packed (buffered) and the number of write calls are minimized for performance.
1. The basic structure is written to disk using *save_struct()*.
2. The attributes are encoded and packed into a buffer, using *save_attr()*.

If an error occurs on a write, the whole series is retried once from the start.

Author's Note:
The whole series of save operations uses synchronous writes, the file is opened with {O_SYNC}. For some incomprehensible reason, O_SYNC is not included in POSIX.1 at this time. However, it is felt that the benefit of insuring the completion of the write out-weighs this slight incompatibility.

job_recov()

```
job *job_recov(char *filename)
```

Args:

filename
The name of the job file from which a job is to be reloaded.

Returns:

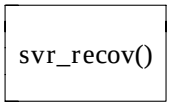
Non-null
job pointer to the newly created job structure on success.

Null job pointer if the recovery failed.

An new job structure is allocated in memory. The job structure, its working attributes, and its dependencies are recovered from disk. This takes place in two steps:
1. The basic job structure is read in.
2. The attributes are restored using *recov_attr()*.

5.3.4.3. svr_recov.c

The file *src/server/svr_recov.c* contains the functions that save and restore the server structure and the server attributes to or from disk.




```
int svr_recov(char *serverdb)
```

Args:

serverdb
name of the server save file.

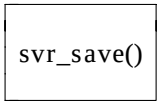
Returns:

0 on success

-1 on error

The server save file is opened. The server structure data is read directly into the structure. Then recov_attr() is called to reload the attributes. The server database file is closed

The server's attributes are searched for one of type {ATR_TYPE_HOSTACL}. When found, *recov_acl()* is called to reload the access control list.




```
int svr_save(server *ps, int mode)
```

Args:

ps Pointer to the server structure.

modeof save, quick or full.

Returns:

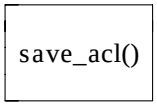
0 on success

-1 on error

If the mode is set for a quick save, {SVR_SAVE_QUICK}, the server database file is opened and the fixed portion, server.sv_qs, of the server structure is written. The file is closed.

Otherwise, a new server database file is opened and save_setup() is called to initialize the save I/O buffer. Then save_struct() and save_attr() are called to save the server structure (serverobj) and the server attributes. Save_flush() is called to finish the I/O and the file is closed. The original save file is unlinked and the new file linked to its name. The new name is unlinked. All this work minimizes the window in which the server database file could be lost if the system crashes.

The server's attributes are searched for one of type {ATR_TYPE_HOSTACL}. When found, *save_acl()* is called to save the access control list.




```
int save_acl(attribute *pattr, attribute_def *pdef, char *path, char *name)
```

Args:

pattr pointer to acl attribute.

pdef pointer to attribute def structure for acl attribute.

path of directory in which acl file lives.

nameof parent object, also the file name.

Returns:

0 if successful.

<0 if error.

Access control list attributes are saved, not with the other attributes of the object, but in their own file. This is done for two reasons. First, the size of the attribute value, the acl entries, might be large. This would slow the updating of the parent object save file and increase the window where a crash might cause loss of data. Second and more important, the separate file allows the administrator to directly edit the access control list. Since the list might be large, its is easier to input directly than through qmgr. Note, any changes will not take effect unless the server is shutdown and reloads the acl.

If the {ATR_VFLAG_MODIFY} is off in the attribute, need do nothing so just return. The file name is created by concatenating the path and the parent object name. The suffix ".new" is appended to the name and this file is created. The attribute is encoded by calling its *at_encode* routine. Note, as the attribute uses the array of strings, arst, encoding, the {ATR_ENCODE_SAVE} flag causes each string entry to be concatenated with a new line separating the sub strings.

Just the value portion of the encoded entry, not the full svrattrl structure is written to disk. This yields an editable file.

The file is closed. The old file name, without the .new suffix is unlinked and then relinked to the new file. The new file name, with the suffix, is unlinked. This ensures the old contents are not lost before the new contents are safe.

recov_acl()

```
void recov_acl(attribute *pattr, attribute_def *pdef, char *path, char *name)
```

Args:

pattr pointer to acl attribute.

pdef pointer to attribute def structure for acl attribute.

path of directory in which acl file lives.

nameof parent object, also the file name.

This function reloads the value of an access control list into the attribute. It is only called when the server is initializing.

The file name is created from the path and parent object name. The file is stat-ed to obtain its size. If the stat fails or the size is zero, the function returns.

The file is opened for read. A buffer large enough to whole the entire file is allocated and the file is read into it. The file is closed.

The data is decoded into the attribute by calling the *at_decode()* routine for the attribute. Then the buffer is freed.

5.3.5. Job Functions

5.3.5.1. job_func.c

The file *src/server/job_func.c* contains general functions to deal with job structures. Functions to allocate and free the job structure, initialize or set the working attributes, abort and restart jobs are included.

job_abt()

```
int job_abt(job *pjob, char *text)
```

Args:

pjob Pointer to job structure for job to be aborted.

text Message to be logged and mailed to owner.

Returns:

0 Job successfully aborted.

-1 Error occurred.

The job state is set to {JOB_STATE_EXITING} and the substate to {JOB_SUBSTATE_ABORT}. A mail message is set to the job owner. A track job batch request is sent to the server which created the job and any defined alternate server.

If the job state was {JOB_STATE_RUNNING} and the server is not initializing, a kill signal is set to the job and the job state is updated to disk.

Else if the job was {JOB_STATE_RUNNING} and the server is initializing (the job was running when the server went down), job exit processing is started to deal with output files.

Otherwise, the job is removed from the system by calling job_purge().

job_alloc()

```
job *job_alloc()
```

Returns:

address
of job structure

NULLif allocation of memory fails.

This function allocates the space for the job structure. The working array of attributes is initialized to "unset" by calling *job_init_wattr()*.

job_free()

```
void job_free(job *pj)
```

Args:

pj Pointer to job structure to be freed.

The various sub-structures of the job structure are freed:
(1) the dependency structures, depend_p and depend_child,
(2) the attribute string set, attrlist, and
(3) the extra space allocated to any of the working attributes.

Finally the job structure itself is freed.

job_init_wattr()

```
void job_init_wattr(job *pj)
```

Args:

pj Pointer to job structure in which to initialize the attributes.

This function is called to initialize the working attribute array in a job structure. For each attribute, the attribute type field is set to match that of the corresponding member of the job attribute definition array. The attribute value flag {ATR_VFLAG_SET} is cleared to indicate the attribute has not be set by a client request (is set to a default unset value).

job_purge()

```
void job_purge(job *pjob)
```

Args:

pjob Pointer to job structure of job to be purged from system.

The job structure is dequeued from any queue by calling svr_dequejob(). The job control file and job script file are unlinked (deleted). If the job has output or checkpoint files in the PBS spool area, they are unlinked. The job structure and all associated structures are freed by calling job_free().

find_job()

```
job *find_job(char *jobid)
```

Args:

jobid The job id character string.

Returns:

Pointer
to the job structure if found, otherwise NULL

Each job in the server's list of all jobs is checked until a job structure with the same job id is found or the end of the list is reached.

5.3.5.2. svr_jobfunc.c

The file *src/server/svr_jobfunc.c* contains general job related server functions.

svr_enquejob()

```
void svr_enquejob(job *pjob)
```

Args:

pjob Pointer to the job.

It is linked into the list of all server jobs. The counts of jobs managed by the server and managed by the server per state are incremented. The queue is located from the queue name in the job structure, *find_queuebyname()* is called. The job structure is linked into the list of jobs owned by the queue.

The position of the job in the server list of all jobs and the queue list is determined by the JOB_ATR_qrank, queue_rank, attribute of the job. Starting at the end of the queue, the most likely place for the job to be placed, the list is searched backwards for a job with rank lower than the new job. The new job is inserted after that job.

The the current count of jobs in the queue, *qu_numjobs*, the number of jobs in the given state, *qu_njstat[state]* and *sv_jobstates[state]*, and the number of total jobs in the server *sv_numjobs* are incremented. The current location attribute, {JOB_ATR_current_loc}, is update to the queue and server name.

If the job is changing queue types, routing to execution for example, the queue dependent type fields in the *ji_un* union are set according to the new queue type.

The job attribute JOB_ATR_qtime is set to the current time if it was unset. This notes the first time into the queue. At this time *account_record()* is called with {PBS_ACCT_QUEUE} to make an accounting file entry. Any unset resource which has a queue specific default value is set to the default value.

If the job is being enqueued in an execution queue, several checks are made. If the job attribute JOB_ATR_depend is set, the function *depend_on_que()* is called to process any job dependency actions which might be required. Note, the use of the {ATR_ACTION_NOOP} mode, this is because depend_on_que() is the at_action routine for dependencies and needs to limit what it does when called for enqueued jobs as opposed to jobs actually being modified. Additionally, the scheduling flag svr_do_schedule is set to {SCH_SCHEDULE_NEW}.

If the job is being enqueued in an route (push) queue, the ji_un union in the job structure is set up for {JOB_UNION_TYPE_ROUTE} type. The ji_quetime field is set to the current time to mark the time in the queue and the next retry time, ji_rteretry, is cleared.

svr_dequejob()

```
void svr_dequejob(job *pjob)
```

Args:

pjob Pointer to job structure to remove from a queue.

The job is unlinked from the queue in which it resides. The the current count of jobs in the queue, *qu_numjobs*, the number of jobs in the given state, *qu_njstat[state]* and *sv_jobstates[state]*, and the number of total jobs in the server *sv_numb_jobs* are decremented. Clear any job resource values which are marked as being set to the queue specific default.

svr_setjobstate()

```
int svr_setjobstate(job *pjob, int newstate, int newsubstate)
```

Args:

pjob pointer to job structure.

newstate
the new value for the job state.

newsubstate
the new value for the job substate.

Returns:

0 if successful.

non zero
if save of job structure failed.

Sets the job state and substate to the supplied values and updates the job save file if needed.

If the job is in substate {JOB_SUBSTATE_TRANSICM}, then it is a brand new job and it has never been added into the various server and queue state counts. Therefore these are not updated at this time. When the job is enqueued into a queue, very shortly, then the counts will be incremented to include this job.

Otherwise, if the state is changed, the server and queue state counts are updated. The state and substate are set to the supplied values. If the queue is an execution queue and the new state is {JOB_STATE_QUEUED}, then *svr_do_schedule* is set to {SCH_SCHEDULE_NEW} to kick start the scheduler as the job is eligible to run. For the later accounting entry, the job attribue JOB_ATR_etime is set to the current time. This will be recorded as the "eligible" time.

If the *ji_modified* flag in the job is set, the job attributes have been modified, then the complete job is save by calling *job_save()* with the save mode of {SAVEJOB_FULL}. Or, if only the state or substate changed, and if you change the state you had better change the substate, then *job_save()* is called with {SAVEJOB_QUICK}. The return value from job_save is passed back to the caller.

svr_evaljobstate()

```
void svr_evaljobstate(job *pjob, int *newstate, int *newsub, int force)
```

Args:

pjob pointer to job structure.

newstate
RETURN: pointer to where recommended job state is returned.

newsub
RETURN: pointer to where recommended job substate is returned.

force if true, force the state evaluation.

When evaluating the state, the attributes of the job which might effect the job state are examined and the recommended state and substate are returned. This function should not be used to directly set the job state. That should only be done via *svr_setjobstate()* as it also updates the job attribute JOB_ATR_state and updates the server and queue state counts.

- If force is false and the current job state is {JOB_STATE_TRANSIT} or {JOB_STATE_RUNNING}, the current state and substate are returned as the suggested state. Code was added to svr_evaljobstate() to not change things when the job was in JOB_STATE_TRANSIT, otherwise a job submitted with a past due execution time screwed up by having its state changed to Queued while still being received; the wait event timer was set to the old time and would go off immediately.

  If force is true, the job is evaluated according to the following rules regardless of the current state.
- If any hold is set it takes precedence over waiting and {JOB_STATE_HELD} is returned.
- If the execute time attribute is set and that time has not been reached, {JOB_STATE_WAITING} is set.
- If the job has a stage-in files attribute JOB_ATR_stagein, set, the state will be {JOB_STATE_QUEUED}. If the files have been staged in (flag {JOB_SVFLG_StagedIn} is set in ji_svrflags), the substate is {JOB_SUBSTATE_STAGECMP} (stage in complete), otherwise the substate is {JOB_SUBSTATE_PRESTAGEIN} (pre-stagein).
- Otherwise, {JOB_STATE_QUEUED} is returned.

get_variable()

```
char *get_variable(job *pjob, char *variable)
```

Args:

pjob pointer to job.

variable
: name of an environment variable passed with job.

Returns:

A pointer to the value part of the name=value environment string if found, null otherwise.

This function finds the environment variable name=value string passed with a job and returns a pointer to the value. It is most often used to find the variable PBS_O_HOST to determine the name of the host from which the job was submitted.

chk_svr_resc_limit()

```
static void chk_svr_resc_limit(attribute *jobatr, attribute *queatr,
attribute *svratr)
```

Args:

jobatrpointer to the job's resource list attribute.

queatr
: pointer to the specific queue's resource limit (max) attribute.

svratrpointer to the server's resource limit (max) attribute.

Returns:

The global variables comp_resc_gt and comp_resc_lt are set according to the comparisions.

For each resource limit (requirement) specified for the job that is not an inherited default value, the limit is compared with:

a. The corresponding queue's limit if one is set for that resource, or

b. The server's limit if one is set for that limit.

The job's resource request (limit) is compared with the the queue or server limit. If the request exceeds the limit, the global variable comp_resc_gt or comp_resc_lt is incremented depending on the relationship of the request to the limit. If neither a queue nor a server limit is set, neither of the global variables is changed.

chk_resc_limits()

```
int chk_resc_limits(attribute *pattr, pbs_queue *pque)
```

Args:

pattr pointer to job's Resource_List attribute.

pque pointer to queue in which the job resides.

Returns:

zero if job's limits are within queue/server bounds, PBSE_EXCQRESC if not.

Each set resource limit (requirement) of the job is checked against the queue's minimum limit specified in the attribute QA_ATR_ResourceMin. If the queue has a maximum limit attribute, QA_ATR_ResourceMax, the job's requirements are checked against it or if there is not a queue max limit, the job is checked against the server's maximum limit SRV_ATR_ResourceMax by calling *chk_svr_resc_limit()*.

svr_chkque()

```
int svr_chkque(job *pjob, queue *pque, char *host, int move_type)
```

Args:

pjob pointer to job structure.

pque pointer to queue structure for queue to check.

host name of host submitting job.

move_type
: type of move, MOVE_TYPE_* as defined in server_limits.h

Returns:

0 if job can be enqueued.

nonzero
: if error, return is an PBSE_ error number.

The *move_type*argument as a result of:

MOVE_TYPE_Move
: new submission or qmove by non-privileged user.

MOVE_TYPE_Route
: routing from a routing queue.

MOVE_TYPE_MgrMv
qmove by privileged user (manager).

MOVE_TYPE_Order
qorder request.

The following checks are made to see if the job can be enqueued into the queue:

1. If the queue is an execution queue, then check the following:
   a. Can the execution uid and gid be established? This is checked first because a return of [PBSE_BADUSER] or [PBSE_BADGRP] is fatal event to a request by a manager to move a job.
   b. Does the job have an "unknown" resource, [PBSE_UNKRESC]? Also fatal to a manager.
   c. Does the job have an "unknown" attribute, [PBSE_NOATTR]? Also fatal to a manager.
   d. If the queue's group ACL is enabled, is the execution group allowed, [PBSE_PERM]? This is not fatal if requested by a manager.
2. The queue is enabled, [PBSE_QUNOENB], and the queue job limit, max_queuable (QA_ATR_MaxJobs) is not exceeded, [PBSE_MAXQUED]. This is not fatal if requested by a manager. This check is skipped for a queue order request on the basis that two jobs are being swapped so the queue limits are not affected.
3. If the queue is marked as accepting jobs only from a routing queue, QA_ATR_FromRouteOnly is true, [PBSE_QACESS]. This is not fatal if either a manger request or the job is from a routing queue. It is not checked for a queue order.
4. If the queue has an enabled host ACL, then the submitting host must be able to access the queue, [PBSE_BADHOST]. This is not fatal if requested by a manager.
5. If the queue has an enabled user ACL, then the job owner must be able to access the queue, [PBSE_PERM]. This is not fatal if requested by a manager.
6. The resources of the job must be with in the range specified by the minimum and maximum resources allowed in the queue, [PBSE_EXCQRESC]. This is not fatal if requested by a manager.

If any check fails, the appropriate error number is returned. If all checks pass, then zero is returned.

job_set_wait()

```
int job_set_wait(attribute *pattr, void *pobject, int actmode)
```

Args:

pattr pointer to the execute-time, {JOB_ATR_exectime}, attribute of a job.

pobject
pointer to a job structure, cast as a void * to match prototype.

actmode
the attribute set mode, see attribute.h.

Returns:

0 if ok.

-1-zero
if error.

This routine is called at the *at_action()* function whenever the execute-time attribute of a job is set.

A search is made for an existing work task on the job's list pointing to *job_wait_over()*. If one is found, the event time is updated to the value of the job wait (execution) time. If one is not found, and if the execution time is later than the current time, an work task entry is created for the wait time and set to invoke *job_wait_over()*.

job_wait_over()

```
static void job_wait_over(struct work_task *pwt)
```

Args:

pwt pointer to a work task entry.

This function is invoked off the server's work task list. The entry was set up with the event time of a job's execution wait time and member wt_parm1 as a pointer to the job. All we need to do is re-evaluate the job's state by calling *svr_evaljobstate()* and *svr_setjobstate()*.

default_std()

```
static void default_std(job *pjob, char key, char *to)
```

Args:

pjob pointer to job.

key to which file, the single character 'o' for output or 'e' for error.

to pointer to buffer in which the file name is placed.

The default name for either the standard output or standard error stream of a job is generated. The name is of the form `job_name".[e|o]job_sequence_number`, where 'e' is used for error or the 'o' for output. The job_name is from the JOB_ATR_jobname attribute. The buffer in which the name is placed must be sufficiently large to hold the name.

prefix_std_file()

```
char *prefix_std_file(job *pjob, char key)
```

Args:

pjob pointer to job.

key to which stream, output or error.

Returns:

pointer
to malloc-ed space holding the generated full path name.

This function builds the fully specified (absolute) default path name for the either the standard output or standard error of a job. The result is of the form:
`qsub_host:$PBS_O_WORKDIR/job_name.[e|o]job_sequence_number`
where `qsub_host` is the name of the host on which the qsub command ran when the job was

submitted, $PBS_O_WORKDIR is replaced by the value of the **PBS_O_WORKDIR** environment variable associated with the job, i.e. the current working directory of the qsub command. The remainder of the path name, the default name, is built by calling *default_std()* described above.

get_jobowner()

```
void get_jobowner(char *from, char *to)
```

Args:

from string from which the owner is obtained.

to buffer to which the owner name is returned.

This function returns the owner name (or any first part of a string) stripping off the "@host" portion (or any part following and including a '@' character). The destination buffer must be large enough to hold the resulting string, for a user name this is {PBS_MAXUSER}+1 characters.

set_deflt_resc()

```
static void set_deflt_resc(attribute *ja, attribute *default)
```

Args:

ja ponter to the job resource attribute (typically Resource_List).

default
pointer to the queue/server attribute to use as a default.

For each resource listed in the default attribute, if the corresponding resource is unset in the job resource_list, set it to the value in the default. Also set {ATR_VFLAG_DEFLT} to indicate it is a default value so it will not be passed if the job is moved to a new queue or server.

set_resc_deflt()

```
void set_resc_deflt(job *pjob)
```

Args:

pjob pointer to job.

This public routine is used to set any default Resource_List values for a job. The function *set_deflt_resc()* (very close in name isn't it) is called in turn with: the queue's resource_default, the server's resource_default, the queue's resource_max, and the server's resource_max attribute.

set_statechar()

```
void set_statechar(job *pjob)
```

Args:

pjob pointer to job.

The job_state attribute, JOB_ATR_state, is set to T, Q, H, W, R, or E depending on the job state in *ji_substate*. A special case − if job state is {JOB_STATE_RUNNING} and the flag {JOB_SUBSTATE_SUSPEND} is set in *ji_svrflags*, the state character is set to S. This is found only for jobs running under Unicos, see *post_signal_req()*.

eval_chkpnt()

```
static void eval_chkpnt(attribute *jobckp, attribute *queckp)
```

Args:

jobckppointer to a job checkpoint attribute.

queckp
pointer to a queue checkpoint attribute.

This function is called when a job is enqueued in an execution queue. It is to insure the that if the job's checkpoint attribute JOB_ATR_chkpnt, is of the form "c=dddd", then the interval value, dddd, is not more than the value of the queue's checkpoint_min attribute, QE_ATR_ChkptMin.

5.3.6. Request and Reply Functions

This section covers the functions related to receiving requests and to issuing requests and replies. Much of the design and implementation was mandated by the use of ISODE.

5.3.6.1. process_request.c

The file *src/server/process_request.c* contains the top level routine invoked to process a batch request from a client program as well as some supporting functions.

process_request()

```
void process_request(socket)
```

Args:

socket
is the socket descriptor from which the request is to be read.

This function is invoked when *accept_conn()* determines that input is available on a socket connected to a client. The purpose of process_request() is to read in the request and dispatch it to the appropriate function for processing.

The server only accepts DIS requests and calls dis_request_read() to read and decode the request. If any connection comes in marked as FromClientASN will cause the server to abort. Note, that MOM only accepts DIS and so only calls dis_request_read().

If the return from dis_request_read() routine indicates end-of-file, The connection is closed by calling a local function *close_netconn()*, If there was a new job being received over the connection, close_netconn is directed to consider enqueuing it.

If the return from the read (isode_request_read()) routine indicates that a read or system error occurred, the connection is just terminated on the assumption that a reply would not get

through either.

If the return from the read routine indicates that the request did not decode correctly, a reject reply is sent to the client.

The host from which the request is being sent is determined by calling *get_connecthost()*. The client host is authorized against the server's host ACL by calling *acl_check()*.

If the client connected to the server on a "reserved" port, the standard socket authorization scheme, we take it as meaning that the client is another server with full privileges. Otherwise, the user making the request is authenticated by calling *authenticate_user()* and the privileges are established by calling >I svr_get_privilege() . If any authentication or authorization fails, the request is rejected with the appropriate error code.

If the server's state is anything other than {SV_STATE_RUN}, then certain requests will be rejected. These ususally entail the running of new jobs or the enqueing of new jobs.

Next, the request is dispatched, via *dispatch_request()*, to the appropriate service function based upon request type. Each service function is required to reply to the request and deallocate the batch_request structure when processing of the request is completed.

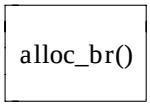

```
void dispatch_request(sock, request)
```

Args:

sock the socket over which the request arrived.

request
a pointer to the batch_request structure.

The request is dispatched to the appropriate routine for processing. Any unrecognized request is rejected.

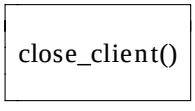



```
struct batch_request *alloc_br()
```

Returns:

pointer
to an allocated batch_request structure.

A batch_request structure is allocated and cleared. The socket descriptor, rq_conn, is set to -1 to indicate there is no connection, This is filled in by the calling routine. The allocated request structure is linked into the list of request structures headed in the global variable svr_requests. The structure should be freed by calling *free_br()*.




```
static void close_client(int socket)
```

Args:

socketthe connection to close.

First, the connection is closed by calling *close_conn()*. The list of active request structures, headed by the global variable svr_requests, is searched for any with the fields rq_conn and rq_orgconn equal to the socket parameter. If found, the field is set to **-1** to indicate the connection has been closed and no reply should be returned.

free_br()

```
void free_br(struct batch_request *request)
```

Args:

request
pointer to the batch_request structure (allocated by process_request).

The batch request structure is unlinked for the list headed by svr_requests. The structure and any allocated sub-structures, including the reply structure, are freed. This is a place where code will have to be added if new types of requests are added.

There are a few routines named *freebr_*()* that are local to this file. They are called by free_br() depending on the type of request.

close_quejob()

```
static void close_quejob(int socket)
```

Args:

socketthe socket descriptor of a closed connection.

When invoked, this function searches the list of incoming jobs headed by sv_newjobs in the server structure. This list is comprised of jobs for which a Queue Job request has been received, but no Commit request.

When the connection to the sending agent is lost one of the following actions is taken.

- If a Ready to Commit has not be received for the job, the job still belongs to the sending agent. The local structure is discarded.
- If a Ready to Commit has been received, the substate is {JOB_SUBSTATE_TRANSICM}, and the job is marked as being created here for the first time, {JOB_SVFLG_HERE} is set in ji_svrflags in the job structure, then the client is a user **qsub** command. In this case all the information is at hand and the client is transitory, so we accept ownership of the job and enqueue it.
- If the substate is {JOB_SUBSTATE_TRANSICM} but {JOB_SVFLG_HERE} is not set, then the job is being transferred from another server. That server retains ownership until it send a Commit. The defined recovery process calls for to just wait for the Commit. Therefore, we leave the job as is.

5.3.6.2. dis_read.c

The file *src/server/dis_read.c* contains the high level functions to read and decode *Data Is Strings* or DIS encoded requests and replies. The lower level routines that perform the actual decode are found as *decode_*()* routines in libpbs.a and *disr*()* routines in libdis.a. An

advantage of the DIS routines is that the data may be decoded directly into the server's batch_request structure eliminating several data copy operations.

dis_request_read()

```
int dis_request_read(int socket, struct batch_request *request)
```

Args:

socketthe socket on which a request has been received.

request
pointer to an allocated batch request structure which will be filled in.

Returns:

0 A request was received and decoded correctly.

-1 EOF received, the client has closed the connection.

positive
PBS error number.

The function *DIS_tcp_reset()* is called to reset the read buffer for the DIS I/O over TCP/IP support routines before the data is read. This would only be required once for the server as it only uses TCP/IP. However MOM uses both TCP/IP and RPP intermixed, so the routines must be reset each time.

The request is in three pieces, (1) the header which contains the requestor's name and the request type, (2) the request body which varies with each type of request, and (3) the request extension. *decode_DIS_ReqHdr()* is called to decode the header. If it fails or if the protocol type and verison in the header are not recognized, [PBSE_DISPROTO] is returned. If decode_DIS_ReqHdr() returns EOF, we also return it (-1).

Based on the request type contained in the header, a large switch statement results in calling the decode_*() routine corresponding to the request type. If an error is returned, it is logged and passed upwards.

The request extension is decoded by *decode_DIS_ReqExtend()*.

DIS_reply_read()

```
int DIS_reply_read(int socket, struct batch_reply *reply)
```

Args:

socketon which to write the reply.

reply pointer to a batch reply structure (contained within a batch_request structure).

Returns:

0 on success, non-zero if error.

This function simply calls *DIS_tcp_reset()* to reset the DIS I/O buffer for TCP/IP and then invokes *decode_DIS_replySvr()* to perform the real work. Any error returned by decode_DIS_replySvr() is just passed on.

5.3.6.3. reply_send.c

The file *src/server/reply_send.c* contains the functions to form an error (or reject) reply and to send a reply back to the requesting client.

set_err_reply()

```
static void set_err_reply(int code, char *msg, struct batch_request *preq)
```

Args:

code The error code to return to the client.

msg pointer to a character buffer in which a message is built.

request
pointer to the batch request.
This routine fills in the basic reply structure within a batch_request. If the current reply union is other than {BATCH_REPLY_CHOICE_NULL}, the structure is freed by calling *reply_free()*.

If the error *code* is [PBSE_SYSTEM], then the value of *errno* is checked for non-zero and having an associated error message, see perror(3). If it exists, the message is appended to the text of *msg_system* for return to the client. If the value of *code* is any other PBS error or if *code* is less than the base number of PBS errors, {PBSE_}, it is assumed to be a local system error number, the routine sees if that error has an associated message. If there is one, that message is placed into *msg*.

reply_send()

```
int reply_send(struct batch_request *request)
```

Args:

request
A pointer to the protocol independent batch request structure which also contains the reply structure.

Returns:

0 If ok

-1 If error

The connection socket descriptor is obtained from the request structure. If the socket descriptor, *sfds*, has the value of {PBS_LOCAL_CONNECTION}, then the request being replied to was from this server. A work task of type {Deferred_Reply_Local} and the event equal to the address of the request structure is located and dispatched by moving the work task entry from the event list to the immediate list. [Note, originally dispatch_task() was called directly to provide immediate processing of the event task. This resulted in a problem of what to do when register dependency request was rejected. The desired end result is to abort the requesting job, however that cannot be done by the routine processing the reply if it is called directly because the higher level routines assume the job will still be around. By moving the work task entry to the immediate list and having it dispatched out of the main loop, all higher level routines have completed their work and we have generalized the case to match that of the request having going off host over the net.]

If the socket descriptor has a positive value, the request came from a different server. The reply is encode by calling *dis_reply_write()*.

Note, if the socket descriptor is negative, but not {PBS_LOCAL_CONNECTION}, then this indicates that the connection was closed on End of File back in *process_request()*. In this case, no reply is sent and no error is returned.

Following either success or failure in sending the reply, the original batch request/reply structure is freed by calling *free_br()*. On an error, a PBS error number is returned.

reply_ack()

```
void reply_ack(batch_request *request)
```

Args:

request
pointer to the batch request.

This routine returns a success reply to a client. The reply structure with in the request structure is filled in with the choice set to {BATCH_REPLY_CHOICE_None}, the code to [PBSE_NONE], and the auxcode to 0. The request and reply are then passed to *reply_send()*.

req_reject()

```
void req_reject(int code, int aux, struct batch_request *request)
```

Args:

code The error code to return to the client.

aux The auxiliary error core.

request
pointer to the batch request.

A batch reply structure with in the request is filled in by calling *set_err_reply()*. The auxcode in the reply is set to the value of *aux*. Then *reply_send()* is called to complete the reply and send it.

reply_badattr()

```
void reply_badattr(int code, int aux, struct svrattrl *pal,
                   struct batch_request *request)
```

Args:

code The error code to return to the client.

aux The auxiliary error core.

pal pointer to the client supplied attributes, in the form of a list of svrattrl.

request
pointer to the batch request.

This routine forms a error reply for a request which is being rejected for an invalid attribute/resource name or value. The basic reply structure is filled in by calling *set_err_reply()*. It is identical to *req_reject()* except that *aux* is used as an index into the *pal* attribute list. The name of that attribute, and resource name if one, is appended to the error message. The main purpose is to identify the offending attribute/resource to the user.

reply_text()

```
void reply_text(struct batch_request *request, int code, char *text)
```

Args:

request
pointer to the batch request structure.

code The error code to return to the client.

text The text string to send to the client.

Set the code to the supplied value, the auxcode to 0, the type to text, and copy in whatever of the text parameter that will fit. Then call *reply_send()*.

reply_jobid()

```
int reply_jobid(struct batch_request *request, char *jobid, int which)
```

Args:

request
pointer to the batch request structure.

jobid the job id string.

whichreply type, the choice discriminator.

Returns:

0 No error

error value if error.

This is used to generate and send a reply containing the job id. It is used to repond to the following requests: Queue Job, Ready to Commit, and Commit.

5.3.6.4. req_getcred.c

The file *src/server/req_getcred.c* contains functions relating authentication of a client making batch requests.

req_getcred()

This function is retained until version 1.1.6 to provide compatibility with 1.1.4 and earlier clients. In 1.1.6, only the non-credential pbs_iff method of authentication will be supported in order to remove encryption and allow export of PBS.

req_connect()

```
void req_connect(struct batch_request *preq)
```

Args:

preq pointer to a Connection Batch Request.

With the removal of encrypted credentials in 1.1.5, the credential type is {int_BATCH_credentialtype_credential__none} and this routine serves mainly to insure the connection from *pbs_connect()* to the server has been made before **pbs_iff** is called to authenticate it.

req_authenuser()

```
void req_authenuser(struct batch_request *preq)
```

Args:

preq pointer to the Authenticated User batch request.

This routine forms the server side of the authentication method introduced in version 1.1.5. The program **pbs_iff** will send over a privileged port the port number of the client. If this connection is found by the server and it is not already authenticated, the connection *svr_conn[socket]* is marked with {PBS_NET_CONN_AUTHENTICATED} and the current time (for historical reasons), and the user and hostname from the request are saved as the credential in *conn_credent[socket].*

5.3.7. Issuing Requests to Other Servers

When the server must issue a request to another server, the Scheduler, or MOM, the server cannot wait on the reply; the issuance of the request and the reception of the reply must be asynchronous events. This is accomplished through the use of a work task order. For each request issued, there is a work task order created that specifies the function to be called when the reply is received. The work task is of type {Deferred_Reply}, and it is connected to the reply by having the event set to the socket number on which the reply will be read.

Another factor which complicates the process of issuing requests is that the request may actually be for the local server itself. For example, a Register Dependency Request may need to be sent to a different server or to the local server depending on the location of the parent job. In order to remove the decision process about location from the request itself, this decision is moved into three common functions:

svr_connect()
: will return a special value, {PBS_LOCAL_CONNECTION}, for the connection handle if the address is local.

issue_request()
: will either connect to a remote server and send it the request, or the function will dispatch the request locally. The decision is based on the value of the connection handle pass to issue_request().

reply_send()
: compliments the issue_request() function by either transmitting the reply to a request to a remote client-server or by directly dispatching the reply if the request was from the local server.

Since the ASN.1 data encoding has been removed, only *issue_Drequest()* is used to issue requests now. Requests will be only use *process_Dreply()* to reply with. All channels should be

marked with {ToServerDIS}.

5.3.7.1. issue_request.c

The file I src/server/issue_request.c contains the function *issue_request()* described in "Issuing Requests to Other Servers".

issue_Drequest()

```
int issue_Drequest(int handle, struct batch_request *request,
                   void (*func)(struct work_task *));
```

Args:

handle
: the connection handle for the connection (real or imaginary) to the server. This is not the socket, but the return from *svr_connect()*.

request
: the batch request structure.

func the function to deal with the reply, it inserted in the work task.

Returns:

0 if request sent ok.

Non-zero
: if could not deliver the request.

If the value of the connection handle is the special value {PBS_LOCAL_CONNECTION}, then the request is for the local server itself. The special value is saved in the request. A work task structure is set up with the the passed function, the type {Deferred_Reply_Local}, and the event being the address of the request structure. Then *dispatch_request()* is called to pass the request to the correct local processing routine. The socket number is set to {PBS_LOCAL_CONNECTION} to indicate this is a request to the local server. (When the reply is returned through *reply_send()*, the work task will be dispatched.)

If the host is a remote host, the work task is set up with the passed function, the type {Deferred_Reply}, and the event equal to the socket number extracted from the connection handle. *DIS_tcp_reset()* is called to reset the write buffer used by the DIS I/O routines. The request is then passed to the appropriate routine to be encoded and written on the network. (Some of these routines reside in the API library, libpbs.a, others are particular to the server. These are handled by calling *encode_DIS_ReqHdr()*, some variant of *encode_DIS_*()* depending on the request, *encode_DIS_ReqExtend()* and *DIS_tcp_wflush()* to complete and write out the request.

issue_Arequest()

```
int issue_Arequest(int handle, struct batch_request *request,
                   void (*func)(struct work_task *));
```

Args:

handle
: the connection handle for the connection (real or imaginary) to the server. This is

not the socket, but the return from *svr_connect()*.

request
the batch request structure.

func the function to deal with the reply, it inserted in the work task.

Returns:

0 if request sent ok.

Non-zero
if could not deliver the request.

If the value of the connection handle is the special value {PBS_LOCAL_CONNECTION}, then the request is for the local server itself. The special value is saved in the request. A work task structure is set up with the the passed function, the type {Deferred_Reply_Local}, and the event being the address of the request structure. Then *dispatch_request()* is called to pass the request to the correct local processing routine. The socket number is set to {PBS_LOCAL_CONNECTION} to indicate this is a request to the local server. (When the reply is returned through *reply_send()*, the work task will be dispatched.)

If the host is a remote host, the work task is set up with the passed function, the type {Deferred_Reply}, and the event equal to the socket number extracted from the connection handle. The request is then passed to the appropriate routine to be encoded and written on the network. (Some of these routines reside in the API library, libpbs.a, others are particular to the server.

process_reply()

```
void process_reply(int sock)
```

Args:

sock The socket file descriptor from which the reply was read.

This function is called by *wait_request()* when a reply to a request is ready to be read, the call to *svr_connect()* was typically established process_reply() as the call back function.

A work task entry on the task_list_event list is located with the event matching the socket. A pointer to the original request is in the work task field wt_parm1. The request address, along with the socket, is passed to *isode_reply_read* which will decode the reply and insert it into the request. The work task is then dispatched.

relay_to_mom()

```
int relay_to_mom(pbs_net_t mom, struct batch_request *request,
                 void (*function)(struct work_task *))
```

Args:

mom The network address of MOM.

request
pointer to the request which is to be sent to MOM.

function
to be invoked when the reply from MOM is received.

Returns:

zero on success

non-zero
if error, see issue_request().

This is a short cut function for transferring an existing or new request to the Machine Oriented Mini-server, MOM. A connection is established to the MOM specified by mom and the request is sent by calling *issue_request()*.

This may be used to relay a request received from a client to MOM. issue_request() will insert the MOM connection socket into the request in rq_conn over-writing the socket to the client which will be needed to reply. Thus, the original socket is saved in the request in rq_orgconn. **Warning:** this value must be restore to rq_conn by whatever routine processes the reply data.

reissue_to_svr()

```
static void reissue_to_svr(struct work_task *task)
```

Args:

task work task pointer created by issue_to_svr().

This routine is called via a time delayed work task entry created by *issue_to_svr()*. It attempts to retry sending a request to a remote server via issue_to_svr(). If the retry time limit is exceeded or the new attempt to connect the remote server fails with no retry possibility, the work task entry will be forwarded to the post processing routine specified by the function which made the request. The *wt_aux* field of the work task is set to -1 to indicate an error. Since, all the post processing routines expect a connection handle in *wt_event*, and this event is a time, wt_event is also set to -1.

If the call to issue_to_svr() was not rejected, this function just returns and lets the dispatch_request() function free the work task entry. Note, that if issue_to_svr() chooses to retry, then a new work task entry is created by it.

issue_to_svr()

```
int issue_to_svr(char *server_name, struct batch_request *preq,
                 void (*reply_function)(struct work_task *))
```

Args:

server_name
of server where request is to be sent.

preq pointer to request to send.

reply_function
is the function to be invoked when the request reply is received.

Returns:

0 on success.

-1 if hard error.

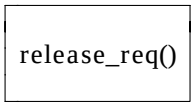
This request is used to send or forward a request to a server. The server may be remote or it may be our self. It is not typically used to send requests to MOM because different error pro-

cessing is required.

The destination server name is copied into the request and the *rq_fromsvr* flag is set to indicate it comes from a server incase the destination server is our self and we use the same structure. Likewise, permissions are set to manager read/write. The server name turned into an address via calls to *parse_servername()* and *get_hostaddr()*. If get_hostaddr() returns *busy, retry*, we will retry later. Any other error is fatal.

svr_connect() is called to obtain a connection to the destination server. If svr_connect() return {PBS_NET_RC_RETRY}, we do so. The handle, request, and post processing function, *reply_function*, are included in a call to *issue_request()*.

If retry is indicated, a work task entry is created by calling *set_task()*. This is a timed entry with a delay of {PBS_NET_RETRY_TIME} seconds.




```
void release_req(struct work_task task)
```

Args:

task setup by issue_request() and used to dispatch this function.

This routine is used as "reply processor routine" when there is no interest in the content of the reply. It frees, *free_br()*, the request structure and disconnects, *svr_disconnect()*, from the other server. It must not be used when the request originated from an outside client, or the client will not receive the answer.

5.3.7.2. svr_connect.c

The file *src/server/svr_connect.c* contains three functions. The function *svr_connect()* is the server's equivalent to the API routine *pbs_connect()*. This function is used by the server to establish a connection to a peer server. The calling server assumes the role of a client to the peer server. The function *svr_disconnect()* is the server's equivalent to the API routine *pbs_disconnect()*.

These two functions brings together the requirements of both the server and its *net_server* system of waiting on I/O together with the *connection_handle* used by the API routines such as *_pbs_queuejob()*. This allows the server to asynchronously wait on the reply from the peer server and use the _pbs_*.c routines of the API. The connection_handle array is much larger than for the typical client.

The function *parse_servername()* will return the host name section of a server name and the optional service port section.


```
int svr_connect(pbs_net_t hostaddr, int port, void (*function)(int socket),
enum conn_type type)
```

Args:

hostaddr

is a pbs_net_t (unsigned long) containing the Internet address in network byte order.

port is the port to which to connect, in network byte order.

function
to be invoked by *wait_request()* when data (a reply) is ready to be read on the connection. The argument to the function is the socket. This function is typically *process_reply()*.

type of data encoding for the connection: {ToServerDIS}.

Returns:

>= 0 is a connection handle for a connection to a remote server.

PBS_LOCAL_CONNECTION
a special value if the destination server is this server.

-1 if an error occurred.

If the host address and port number match that of this server, then {PBS_LOCAL_CONNECTION} is returned. No physical connection is made, see issue_request().

Otherwise, the libnet.a routine *client_to_svr()* is called to open the connection with the specified host address and port number. The socket is added to the svr_conn array by calling *add_conn()*. The entries type is {General} and the call-back function for data ready to read is *func*. If func is not null, meaning that a reply will be read, *add_conn()* is called to make func the call back function. For releases 1.1.9 and 1.1.10, PBS marks the connection with the *type* passed in the call.

The connection handle array used by the API routines has an entry added and the the index into the array is the return value. An ISODE Presentation Stream is allocated for use by the API routines.

svr_disconnect()

```
void svr_disconnect(int handle)
```

Args:

handle
the connection handle returned by svr_connect().

If the handle is valid, the ISODE presentation stream is freed, the connect_handle array member is released, and the socket is closed by calling *net_close()*. Note, a handle of {PBS_LOCAL_CONNECTION} is greater that the maximum allow handle index and a handle of -1 indicates the connection is not open.

socket_to_handle()

```
int socket_to_handle(int socket)
```

Args:

socketnumber of the socket.

Returns:

The number of a “connection handle” set up for the socket; -1 if error.

An unused entry in the connection table, *connection[]*, is located and assigned to the socket. ISODE streams are allocated for it.

parse_servername()

```
char *parse_servername(char *name, int *service)
```

Args:

name the server's name in the form `hostname[:port]`.

service
RETURN: the port number, if specified in name, is returned. If there is not a :port in the name argument, *service is unchanged.

Returns:

A pointer to the host name, up to but not includeding any :port returned. The host name is in static storage and will be overwritten on the next call to parse_servername().

The hostname[:port] passed in name is parsed.

5.3.8. Queue Functions

5.3.8.1. queue_func.c

The file *src/server/queue_func.c* contains general functions for queue structure management.

que_alloc()

```
queue *que_alloc()
```

Returns:

Pointer
to queue structure created

Null if unable to create queue.

This function is called to create a queue structure in memory. The space is allocated and cleared. The structure is linked into the server list of queues headed in sv_queues. The number of queues, sv_numque, is incremented. Each attribute array entry is set to"unset". For the attributes in the array of those common to all queues, the attribute type flag is set. In the union of attributes that are queue type dependent, the type flag is not set.

The queue is marked as modified, qu_modified is set to one, but the structure is not written to disk by this routine.

que_free()

```
void que_free(queue *pq)
```

Args:

pq Pointer to the queue structure to be freed.

Any space allocated to the attributes is freed. The server count of queues, sv_numque, is decremented and the queue structure is unlinked from the server list. Then the queue structure itself is freed.

que_purge()

```
int que_purge(queue *pq)
```

Args:

pq Pointer to the queue to be removed from the system.

Returns:

0 If successful

-1 If error.

An error is returned if the queue to be purged owns any jobs.

The queue save file is unlinked and the queue structure is released by calling *que_free()*.

find_queuebyname()

```
queue *find_queuebyname(char *qname)
```

Args:

qname
The name of the desired queue.

Returns:

Pointer
to the queue structure if found

Null If no queue found

Search linked list of server's queues for one with given name. Any `@server` suffix on the queue name is ignored.

get_dfltque()

```
queue *get_dfltque()
```

Returns:

pointer
to the default queue if defined, or NULL.

If the server attribute default_queue is set, and if there is a queue by that name, a pointer to it is returned. Otherwise, a null pointer is returned.

5.3.8.2. queue_recov.c

The file *src/server/queue_recover.c* contains the functions to save and restore a queue structure and its associate attributes.

que_save()

```
int que_save(queue *pque)
```

Args:

pque Pointer to queue structure which is to be saved.

Returns:

0 if success

-1 if error

If the queue is marked as modified, it is saved to disk. If not, it isn't.

The queue file name is based on the queue name, which is obtained from the queue structure. This file is opened. save_setup() is called to initialize the save buffer. The queue structure is written using save_struct().

The queue attributes are saved by calling *save_attr()*.

The save buffer is flushed, save_flush(), and the file is closed. The queue is marked an not modified.

The queue's attributes are searched for any of type {ATR_TYPE_HOSTACL}, {ATR_TYPE_USERACL}, or {ATR_TYPE_GRPACL}. When found, *save_acl()* is called to save the contents of the access control list to its own file.

que_recov()

```
que *que_recov(char *filename)
```

Args:

filename
The name of the queue save file.

Returns:

Non null
queue pointer to the new queue structure upon success.

Null pointer on failure.

The queue structure is allocated and initialize via *que_alloc()*. The file specified is opened. The basic queue data is read into the queue structure pointed to by pque. The attributes are reloaded by calling *recov_attr()*.

The queue's attributes are searched for any of type {ATR_TYPE_HOSTACL}, {ATR_TYPE_USERACL}, or {ATR_TYPE_GRPACL}. When found, *recov_acl()* is called to reload the contents of the access control list from its own file.

The queue is marked as not modified to prevent an unnecessary rewrite to disk.

5.3.9. Server Functions

This section of the IDS covers a collection of modules which contain general bookkeeping functions for the server. If they did not fit else where, they are probably here.

5.3.9.1. run_sched.c

The file *src/server/run_sched.c* contains functions used by the Server to contact and command the job Scheduler. The connection to the Scheduler is a two faced connection, or maybe I should say it turns on you. The Server contacts the Scheduler to open the connection and sends it a schedule command. This makes the Server a client to the Scheduler. But the

scheduler needs to send requests to the Scheduler as a client. Thus after sending the command the Server adds the connection to those from which it accepts requests and the Scheduler sets up the connection to look like it was created via a call to pbs_connect().

The schedule command sent from the Server and the Scheduler is a simple 4 byte integer, in network order. The integer has the value of: {SCH_SCHEDULE_NEW}(1), {SCH_SCHEDULE_TERM}(2), {SCH_SCHEDULE_TIME}(3), {SCH_SCHEDULE_RECYC}(4), or {SCH_SCHEDULE_CMD}(5). Additional commands are planned but not currently supported.

schedule_jobs()

```
int schedule_jobs()
```

Returns:

- -1 Error occurred, could not contact the Scheduler.
- 0 Scheduler was sent the schedule command.
- +1 An unresponded schedule command is already outstanding to the Scheduler, only one at a time is allowed.

This routine is called from the main scheduler loop in *pbsd_main()*. If this is the first time the function has been called, the scheduler command {SCH_SCHEDULE_FIRST} will be sent to the scheduler regardless of the reason it was called. If scheduler_sock is minus one (otherwise it is the socket of the existing connection to the Scheduler), *contact_sched()* is called to send the command, listed above to the scheduler. The command is found in the external variable svr_do_schedule.

contact_sched()

```
static int contact_sched(int command)
```

Args:

command
is the integer command to be sent to the scheduler.

Returns:

socketof the connection to the scheduler or -1 if error.

The function *client_to_svr()* is called to open a connection to the Scheduler at address pbs_scheduler_addr and port pbs_scheduler_port. Then *add_conn()* is called to add the connection to the set to which the server will listen for requests, and *net_add_close_func()* to register the locat function *scheduler_close()* as the function to be called when the connection closes. Next *put_4byte()* is called to output the command.

put_4byte()

```
static int put_4byte(int socket, unsigned int command)
```

Args:

socketconnection to the Scheduler.

command
to be sent.

Returns:

0 for success, or -1 if error.

This function takes the least significant four bytes of the command, places them in network order and writes them on the connection. It will work for any architecture where the size of an unsigned int is at least 4 bytes.

The corresponding routine, get_4byte(), is found in *src/scheduler.rules/get_4byte.c*.

The return value is -1 if 4 bytes could not be written on the socket.

scheduler_close()

```
static void scheduler_close(int socket)
```

Args:

socketconnection which was closed, unused.

The variable scheduler_sock is set to -1 to indicate to *schedule_jobs()* that the Scheduler connection is terminated.

If only one job was "run" by the scheduler during the cycle, as shown by scheduler_jobct being set to one, then the external (see pbsd_main.c) svr_do_schedule is set to {SCH_SCHEDULE_RECYC} to recall the scheduler. A scheduler script may be written to run only one job per cycle to ensure its newly taken resources are considered by the scheduler before selecting another job. In that case, rather than wait a full cycle before scheduling the next job, we check that one (and only one) job was run by the scheduler. If true, then we recycle the scheduler (a committee decision).

5.3.9.2. geteusernam.c

The file *src/server/geteusernam.c* contains functions to obtain the login name and group under which the job should be executed and set the corresponding uid, gid in the job structure.

geteusernam()

```
static char *geteusernam(job *pjob, attribute *pattr)
```

Args:

pjob pointer to the job structure.

pattr pointer to the User_List attribute, either the job's or the newly modified (qalter).

Returns:

pointer
to the user name.

The name is located by trying the following steps in the order listed until a name is found.

1. A username@host in the attribute User-List with a host name matching the local host name.

2. A username in the attribute User-List with no host name specified, this is the wild card username.
3. The username from the job attribute owner-name. This name is mapped to a local name by calling *site_map_user()*. (Remember, the PBS supplied version of site_map_user() just returns the name given as input.)

The User-List attribute is of type {ATTR_TYPE_ARST}, array of strings. Each string in the array is of the form `username[@host]`.

The selected name is saved in a static buffer and stripped of any host name.

getegroup()

```
static char *getegroup(job *pjob, attribute *pattr)
```

Args:

pjob pointer to the job.

pattr pointer to the group_list attribute, either from the job structure or a newly modified one (qalter).

Returns:

pointer
to a string containing the the group name, null if one is not specified.

This function returns the name of the group under which the job should execute if one was specified. The passed attribute, JOB_ATR_grouplst, is searched for

1. A name with a host name matching the server host, or
2. No host name (the wild card host).

If neither is found, a null pointer is returned.

set_jobexid()

```
int set_jobexid(job *pjob, attribute *attr_array)
```

Args:

pjob pointer to job structure.

attr_array
pointer to array of job attributes, either the actual job's, or if they are being modified, the newly modified array, see modify_job_attr().

Returns:

0 if successful.

non-zero
error number, if error.

The execution uid and gid fields in the job, ji_euid, ji_egid, are set. The name under which the job should be executed is obtained by calling *geteusernam()*. It is called with either the User_List attribute from the passed in attribute array; or it is unset, the actual job's working attribute ji_wattr[JOB_ATR_userlst].

The password entry for returned name is retrieved. If there is not an entry [PBSE_BADUSER] is returned. If {PBS_ROOT_JOBS} is defined non-zero, an UID of zero is allowed if and only if the job owner is root*@this host*. If {PBS_ROOT_JOBS} is defined to zero, then an UID of zero is not allowed at all and is returned.

The job structure, which contains the job owner name and submitting host name, and the local user name are passed to *site_check_user_map()* to see if the user is authorized to execute a job as the selected user. The user name is placed into the job attribute JOB_ATR_euser.

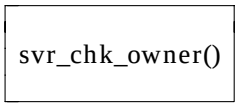
For Cray Unicos system, an addition check is performed. These systems have a User Data Base (UDB) which contains permission bits. Two are of interest at this point, if either {PERMBITS_NOBATCH} or {PERMBITS_RESTRICTED} is set for the user, he is denied access to the system for batch jobs (or at all). The job is aborted with [PBSE_QACESS]. Also for the Cray, if the job account attribute, JOB_ATR_account, is not set, the default account id, ACID, is obtained from the UDB entry.

The routine *getegroup()* is called with either the group_listmember ji_wattr[JOB_ATR_grouplst] of the job; the function determines if a group was specified for the execution. If a group name was specified and the group is not the user's primary group and the user name is not listed as a member of the specified group, [PBSE_BADGRP] is returned. If a group was specified, and it was the user's login group, then that is allowed. If a group was not specied for this host, then the user's login group is taken as the default. The job attribute JOB_ATR_egroup is set to the group name; or in the case of defaulting to the login group and getgrnam() return null (no such group), the numerical value (gid) is converted into a string for JOB_ATR_egroup.

Also, if the group is the primary group from the password file, the attribute default value flag, {ATR_VFLAG_DEFLT} is added to the attribute. This has special meaning to MOM, see *start_exec()*, and *setup_cpyfiles()* in server/req_jobobit.c.

5.3.9.3. svr_chk_owner.c

The file *src/server/svr_chk_owner.c* contains functions supporting authorization and authentication checking of batch requests.




```
int svr_chk_owner(preq, pjob)
```

Args:

preq pointer to the batch request structure.

pjob pointer to the job structure.

Returns:

0 if requesting user is the job owner.

non
if not job owner.

The user name and host name from the request are mapped to a local name by *site_map_user()*. The owner of the job is obtained from the job owner attribute JOB_ATR_job_owner. The host name from which the job was submitted is obtained from the job by calling *get_orighost()*. It along with the job owner's name is mapped via site_map_user(). If the two resulting local names are equal, zero (0) is returned; else non-zero is returned.

svr_authorize_jobreq()

```
int svr_authorize_jobreq(struct batch_request *request, job *pjob);
```

Args:

request
pointer to the batch request.

pjob pointer to the job structure.

Returns:

0 if the client is authorized to act on the job.

non-zero
if not authorized.

The requester or client is autorized to act on a job if the requester is the job owner, see *svr_chk_owner()*, or has been granted Operator or Manager privileges.

svr_get_privilege()

```
int svr_get_privilege(char *user, char *host)
```

Args:

user the name of the user (client).

host the host from which the request is being made.

Returns:

(integer)
which is the read/write privilege granted.

The function *svr_get_privilege()* returns the access privilege granted to the named user. There are three levels of privilege defined:

User has no special level of privilege. A user has the ability to create, alter, status and delete his/her own jobs. A user can also status queues and the server.

Operator
has one level of special privilege. An operator can alter, status, and delete any user's jobs, status and alter queues, and status the server.

Administrator
has the highest level of privilege. An administrator has all the capabilities of an operator plus the privilege to create and delete queues and alter the server.

Any client user is automatically granted "user" privilege. Administrator and operator privilege is granted on a name at host basis. If the user name associated with the host (or wild card) appears in the server's administrators or operators attribute, then that user is granted the corresponding additional privilege.

The return value from svr_get_priv() is the bitwise "and" of the following values which are defined in *attribute.h*:

user	ATR_DFLAG_USRD & ATR_DFLAG_USWR
operator	ATR_DFLAG_OPRD & ATR_DFLAG_OPWR
administrator	ATR_DFLAG_MGRD & ATR_DFLAG_MGWR

authenticate_user()

```
int authenticate_user(struct batch_request *request)
```

Args:

request
pointer to the server network independent batch_request structure.

Returns:

0 if user is authenticated

<0 if authenticate fails.

In the basic provided system, the user is authenticated if the user name and host name provided in the credential matches the user name in the request and the host name determined from the network interface. The time stamp must be current, not less than {CREDENTIAL_TIME_DELTA}seconds {CREDENTIAL_LIFETIME} seconds more than the local system time.

chk_job_request()

```
job *chk_job_request(char *jobid, struct batch_request *preq, int sock)
```

Args:

jobid the job identifier of the job to which the request applies.

preq pointer to the batch request.

sock the socket over which any reply is sent.

Returns:

pointer
to the job, null if error.

This function provides the common checks for batch service requests that apply to existing jobs. First the job is located; if not found [PBSE_UNKJOBID] is returned to the client.

If the client is not authorized to make the request against the job, see *svr_authorize_jobreq()*, [PBSE_PERM] is returned to the client.

Finally, if the job is in the exiting state, [PBSE_BADSTATE] is returned.

On any error, a reply is sent to the client via *req_reject()* and the function returns a null job pointer. The caller should just return up the line.

5.3.9.4. svr_func.c

the file *src/server/svr_func.c* contains various server support functions.

encode_svrstate()

```
int encode_svrstate(attribute *pattr, list_head *head, char *name, char *rescn,
                    int mode)
```

Args:

pattr pointer to the server state attribute.

head head of list of encoded attributes, svratrlst, to which to append the attribute.

namename of the server state attribute.

rescnresource name, null.

modethe encode mode.

Returns:

zero if successful, non-zero if error.

This is a special "at_encode" routine for the server state attribute. It turns the numeric state into the corresponding textual name: Idle, Active, Scheduling, Terminating, or Terminating Delayed.

The choice between Idle and Active is made based on the setting of the scheduling attribute. If there is a call outstanding to the scheduler, its socket is not -1, then the state is mapped into Scheduling.

set_resc_assigned()

```
void set_resc_assigned(job *pjob, enum batch_op op)
```

Args:

pjob pointer to job which is being taken into running or exiting state.

op operator, {Incr} or {Decr}.

When a job is being placed into run state or taken out of run state, this routine is called to update the server attribute SRV_ATR_resource_assn, resources used. This attribute is the sum of certain resource requirements of jobs in the running state. The attribute may be useful in scheduling scripts.

If the job is not in state {JOB_STATE_RUNNING}, this function just returns. (Might be called twice if MOM is restarted after the job terminates). For each resource list member which is marked in the resource definition with {ATR_DFLAG_RASSN}, that resource limit value is added/subtracted to/from the corresponding resource member. of SRV_ATR_resource_assn.

ck_chkpnt()

```
int ck_chkpnt(attribute *pattr, void *pobject, int mode)
```

Args:

pattr pointer to job checkpoint attribute.

pobject
not used here

modenot used here

Returns:
a PBS error number or 0 if ok.

This is the "at_action" routine for the job's checkpoint, JOB_ATR_chkpnt, attribute. Ck_chkpnt is called whenever the checkpoint attribute value is set or changed. The routine makes sure

the value is proper, equal to "n", "s", "u", "c", or "c=dddd", where dddd is a number.

5.3.9.5. svr_mail.c

The file *src/server/svr_mail.c* contains the function to send mail to a job's mail list.

svr_mailowner()

```
void svr_mailowner(job *pjob, char mailpoint, int force, char * text)
```

Args:

pjob Pointer to the job about which mail is to be sent.

mailpoint
The single character indicating the mail point.

force flag to force sending the mail.

text The character string of the message to mail.

The *mailpoint* parameter is a single character which identifies the point at which mail is being sent:

a for abort,

b for beginning of execution,

e for exit, and

s for file staging (in) error.

If the force flag is true, the mail message is to be sent. Otherwise the job attribute JOB_ATR_mailpnts is checked to see if the user requested mail at this point. If not, the function just returns.

If mail is to be sent, the function forks with out setting up a work task on the pid as there is nothing to do when the child exits. The parent returns to the caller.

The child process builds the sendmail command is built up in a buffer. It includes the -f option to specifiy the "sender's name" which is obtained from the server attribute SRV_ATR_mailfrom, "mail_from". Also included is the mail destination, if the job has a specified JOB_ATR_mailuser attribute, that list is used instead of the job owner as the recipient of the mail. The command line is passed to *popen()* and the mail headers and body message are written on the pipe. The headers includes a subject phrase based on the mail point. The child process then exits.

The server will reap the child and clean up the child_task entry.

5.3.9.6. svr_messages.c

The file *src/server/svr_messages.c* has been replaced by *src/lib/Liblog/pbs_messages.c* because of a change to log_err() to print messages associated with PBS error numbers.

5.3.9.7. svr_resccost.c

The file *src/server/svr_resccost.c* contains functions associated with the resources_cost attribute and calculating the resource cost of a job. This attribute and these functions support the synchronous job starting functions found in *req_register.c*.

It was the original intent to have the resource cost be an integer recorded in the resource_definition structure itself. It seemed logical, one value per definition, why not. But "the old atomic set" destroys that idea. Is is necessary to be able to have temporary attributes with their own values, hence it came down to another linked-list of values. Each en-

try contains the cost value and a pointer to the resource definition structure to tie the cost to that resource.

add_cost_entry()

```
static struct resource_cost *add_cost_entry(attribute *pattr,
                                            resource_def *pdef);
```

Args:

pattr pointer to the resources_cost attribute.

pdef pointer to the resource definition structure for the specific resource.

Returns:

pointer
to the newly created resource cost entry, NULL if error.

A new entry is allocated and initialized to zero.

decode_rcost()

```
int decode_rcost(struct attribute *pattr, char *name, char *rescn, char *val)
```

Args:

pattr pointer to the resources_cost attribute.

nameof the attribute.

resc the resource name.

val The cost of the resource (the value).

Returns:

zero on success.

non-zero
on error.

The resource cost entry for the specified resource is found in the list headed in the attribute. If not found, a new one is created by calling *add_cost_entry()*. The value string is converted to an integer and inserted in the structure.

encode_rcost()

```
int encode_rcost(attribute *pattr, list_head *phead, char *atname,
                 char *rsname, int mode)
```

Args:All arguments are standard for an at_encode() routine.

Return:

Greater than zero on success, zero if attribute was unset, negative if error.

For each entry in the resource cost attribute list, a svrattrlst entry is created by calling *attrlist_create*. The al_value field is set to the resource cost value and the entry is linked on

the list headed by phead.

set_rcost()

```
int set_rcost(attribute *old, attribute *new, enum batch_op op)
```

Args:

old the attribute whose value is to be modified.

new the attribute whose value is the modifier.

op SET, INCR, or DECR operation.

Return:

zero on success, non-zero on error.

For each entry in the new attribute, the corresponding value in the old attribute is modified according to the operation.

free_rcost()

```
void free_rcost(attribute *pattr)
```

Args:

pattr pointer to the resource cost attribute which is to be freed.

All entries in the list of resource_cost structures headed in the attribute are deleted from the list and freed.

calc_job_cost()

```
long calc_job_cost(job *pjob)
```

Args:

pjob pointer to the job for which the resource cost is to be calculated.

Returns:

The resource cost of the job.

The resource cost of the job is the sum of the "per system cost," SVR_ATR_sys_cost, and the products of the specified resource costs and their respective amounts of resources. To the the produce for becoming too large, for those resources measured in "size", the size is converted to *megabytes* before multiplying by the cost, i.e. the cost is in terms of megabytes, not bytes.

5.3.9.8. svr_task.c

The file *src/server/svr_task.c* contains the server functions for maintaining the list of deferred services such as route retry, job waiting, and completion of batch requests that depend on communication with other processes, e.g. MOM.

The tasks fit into one of three major types:

Immediate Tasks which the server should act upon immediately. Many entries are placed into this list by *pbsd_init()* during server recovery.

Time — Tasks which are deferred to a specific time (in the future). Jobs in the Wait state have a task entry of this type.

Event — Tasks which are deferred to the occurrence of a specific (external) event. All child processes are recorded by this type of task entry as are batch requests which depend on the response of another process.

The deferred tasks are recorded in a work task structure. All tasks of the same type are linked together in lists headed in the global variables *task_list_immed*, *task_List_time*, or *task_list_event*. An entry is created and added to the appropriate list by *set_task()*. An entry is removed by calling *delete_task()*.

Note that set_task() returns a pointer to the work task entry. This is often used to add the entry to a list headed in the structure referenced by *wt_parm1*. Wt_parm1 is often a pointer to a structure, such as a job structure. This pointer is typically used by the function invoked by the task dispatcher. If it is at all possible that the "pointed to" structure could be freed before the work task is acted on, the list of work tasks in the structure is used to delete the work task along with the "pointed to" structure. The caller of *set_task()* **MUST** add the work task entry to the structure's list of work tasks.

The tasks on the immediate and timed list are processed in the main server loop. Events on the event list are either processed when the event is detected or shortly there after by moving the event to the immediate list.

> WARNING:
> You should never move an entry from one list to another in a signal handler as you cannot be sure of the state of the links.

set_task()

```
struct work_task *set_task(enum work_type type, long event_id,
                           void (*func)(struct work_task *),
               void *parm1)
```

Args:

type The type of task.

event_id
: An identifier to relate this task with a specific event.

func The function to perform the task.

parm1
: The parameter to be saved in wt_parm1 in the work task entry.

Returns:

pointer
: to the allocated work task entry.

Null if an error occurred and no entry was allocated.

A work task entry is allocated and initialized with the data passed as arguments to this function. The entry is added to one of the three lists maintained by the server depending on the event type: immediate, timed, or external event. If the additional parameter entries in the work task entry *wt_parm2* and *wt_aux* are meaningful to the invoked function, *set_task* **the caller of must initialize them**.

The function assigned to process the task, *func()*, must take one argument, a pointer to the work task entry.

dispatch_task()

```
void dispatch_task(struct work_task *task)
```

Args:

task pointer to a work task entry to dispatch.

The work task entry is unlinked from both the main server list and the optional (job) structure list. If specified, the function in the task entry is called and passed a pointer to the work task itself. When the function returns, the work task entry is freed.

delete_task()

```
void delete_task(struct work_task *ptask)
```

Args:

ptaskpointer to the task entry to clear.

The task entry is unlinked from its list(s) and freed.

5.3.9.9. list_link.c

The file *src/server/list_link.c* contains routines for maintenance of a doubly linked list. The list is linked through a structure *list_link* in each entry. The list is headed by a *list_head* structure (nothing more than another list_link), Each link is contained in a *list_link* structure. In addition to forward and backward pointers, the list_link structure contains a pointer to the parent structure which contains the link. This allows a structure to have multiple list_link structures and to reside in multiple lists. In the head entry, this pointer to the parent structure is a NULL pointer. This allows the end and head of the list to be recognized. NEVER, NEVER, allow the parent structure pointer in a list member TO BE NULL; or the parent structure pointer in the head structure TO BE NOT NULL; or the next and prior pointers in the head to be NULL!

The definition of the link structures are contained in the file *include/list_link.h*. Also defined in the header file are the following macros:

CLEAR_HEAD()
: which clears a list head structure including the parent structure pointer.

CLEAR_LINK()
: which clears the next and prior members of a list link structure.

GET_NEXT()
: which returns the address of the parent structure of the next item in the list. A NULL pointer is returned if the end of the list is reached.

GET_PRIOR()
: which returns the address of the parent structure of the previous item in the list. A NULL pointer is returned if the head of the list is reached.

insert_link()

```
void insert_link(struct list_link *old, struct list_link *new,
                 void *pnewobj, int position)
```

Args:

old Pointer to an list_link entry already in the list or the head structure.

new Pointer to the list_link sub-structure in the new entry.

pnewobj
Pointer to the parent structure, holding the new list_link sub-structure.

position
If 0, then the new entry is added before the old, else it is added afterwards.

The new entry is added to the list either before or after the old entry depending on the setting of position. Note, if the old entry is the list head, inserting "after" makes the new entry the first in the list; inserting "before" makes the new entry the last in the list.

The seemingly extra parameter, pnewobj, is a pointer to the parent structure of the list_link sub-structure, If the list_link could always be the first member of the parent structure, this would not be needed. However, to allow for the structure to be in multiple lists, this extra parameter is required. The links always point to the top of the parent structure, allowing other members to be addressed.

append_link()

```
void append_link(struct list_head *head, struct list_link *new,
                 void *newpobj)
```

Args:

head Pointer to (address of) the list_head structure.

new Pointer to the list_link sub-structure in the new entry.

pnewobj
Pointer to the parent structure containing the new list_link structure.

The new entry is appended to the end of the list.

delete_link()

```
void delete_link(struct list_link *old)
```

Args:

old Pointer to the entry to be deleted from the list.

The entry is removed from the list. The forward and back link pointers in the old entry are set to point to itself. Otherwise the old entry is not disturbed.

swap_link()

```
void swap_link(list_link *one, list_link *two)
```

Args:

one pointer to one entry in a list.

two pointer to another entry in the same list.

This routine swaps the positions in a list of two members of the list. If the two members are adjacent, one is moved after the other. Otherwise, each entry is unlinked and relinked after the entry ahead of the other.

is_linked()

```
int is_linked(list_head *head, list_link *entry)
```

Args:

head Pointer to head of list.

entryPointer to list_link structure in question.

Returns:

1 if the entry is in the list headed by head.

0 if the entry is not in the list.

This function walks the list until it encounters the entry in question or reaches the end of the list.

list_move()

```
void list_move(list_head *from, list_head *to)
```

Args:

from pointer to a list_head.

to pointer to a list_head.

The list headed by *from* is moved to be headed by *to* instead. The list head *from* is cleared. The whole thing is just insuring that the pointer in the head and tail list elements point to the correct list_head structure.

5.3.9.10. accounting.c

The file *src/server/accounting.c* contains routines for the creation of the server accounting file.

acct_job()

```
static void acct_job(job *pjob, char *buf)
```

Args:

pjob pointer to job for which the accounting record is to be written.

buf pointer to a buffer in which the record is built. It must be big enough.

Returns:

pointer to next available byte in buffer.

This private routine is used by *account_jobstr()* and *account_jobend()* to add the following information to the accounting record being built in buffer: user, group, account, job name, session id, job creation time, job queued time, time when the job became eligible for execution, the time the job started execution, and the job resource requirements.

acct_open()

```
int acct_open(char *filename)
```

Args:

filename
of the accounting file to be opened.

Returns:

zero on success, -1 if error.

Calling acct_open() with a null pointer request that the default account file, based on the current day be opened. The file will be switch each day with the first record after midnight, see *account_record()*.

Calling it will a pointer to a null string, from a -A "", is direction to not to open a file. This in effect, turns off account recording. Calling acct_open() with a full path name turns off switching to a new file each day.

void

```
void acct_close()
```

Closes the accounting file if open.

account_record()

```
void account_record(char type, job *pjob, char *text)
```

Args:

type of record

pjob pointer to job

text to append to record.

This function formats and records the basic record. The supplied text is appended to the date time stamp, type character, and job id.

If automatic file switching is on (using default file name) and the current day is not the same day as the the day the file was opened, then the file is closed, *acct_close()*, and opened anew, *acct_open()*.

account_jobstr()

```
void account_jobstr(job *pjob)
```

Args:

pjob pointer to job

This function builds the text part of a job start (of execution) record. The function *acct_job()* is used to list the basic information about the job. Then *account_record()* is called.

account_jobend()

```
void account_jobend(job *pjob, char *used)
```

Args:

pjob pointer to job

used text about the resources which were used by the job.

This function builds the text part of a job end (of execution) record. The function *acct_job()* is used to list the basic information about the job. Then the information from *req_jobobit()* about resource usage is appended. Last, *account_record()* is called.

5.3.10. Node Functions

The functions in this section deal with **Node** resources. The functions include allocating, reserving, and freeing.

5.3.10.1. node_manager.c

The file *src/server/node_manager.c* contains functions that

(1) Deal with nodes as resources: allocating, reserving, and freeing.

(2) Server to Mom communication used to tract state of the nodes.

write_node_state()

```
void write_node_state()
```

This routines writes the node state file {NODE_STATUS} which is PBS_HOME/server_priv/node_status. If the file is not already open, it is opened. If already opened, the file is truncated to zero length.

The file is written as the node name and the state as an integer. Only those nodes which are marked *off-line* are recorded in this file. If a node is allocated to a job, that is determined by the recovered job attributes. If a job is down, that is discovered when the server cannot communicat with the node.

free_prop()

```
static void free_prop(struct prop *proplist)
```

Args:

proplist

A pointer to a linked list of properities of a node.

A properity is just a string, which may be descriptive of some property of the node, assigned by the Batch Administrator. Zero or more may be assigned via the node description file, see *setup_nodes*().

This routine frees the structures used to hold the property strings.

node_unreserve()

```
void node_unreserve(resource_t handle)
```

Args:

handle

A resource handle used to identify a set of reserved resources.

This function releases the reservation on a set of nodes. The reservation is identified by *handle*. If *handle* is the special value {RESOURCE_T_ALL}, then all reserved resources are released.

hasprop()

```
static int hasprop(struct pbsnode *node, struct prop *props)
```

Args:

node pointer to a single pbsnode structure

propsA list of properities, some or all of which are marked as "needed"

Returns:

One if the node has the "needed" properities, zero if not.

For each "needed" property in the *props* list, check the property list of the specified *node*. If all needed properites are in the nodes property list return 1, else return 0.

mark() nodes

```
static void mark(struct pbsnode *node, struct prop *props)
```

Args:

node pointer to a single pbsnode structure

propsa list of properities

For each property in *props*, mark that property in the *node* property list.

search() nodes

```
int search(struct prop *glorf, int skip, int order, int depth)
```

Args:

glorf a properity list

skip a bit mask, if bits match those in the *inuse* field of the node, that node is skipped (ignored)

order the position or order of the needed node in the user's specification

depth the limit of the depth of the recursive search - used to limit the search time

Returns:

One if the nodes are available - the nodes are marked in the flag field with the {thinking} flag. If the nodes are not available, zero is returned.

This function looks for a node which contains the properties given in the list *glorf*. The parameter *check* is a flag to indicate if nodes which are in use should be checked. The parameter *order* is the order of this particular node in the user's specification. First, the node list is searched for one with the given properties. If one is found, it is marked "thinking" and a 1 is returned. If not, the nodes which are marked "thinking" are searched. If one is found with the given properties, mark it "conflict" and call *search()* recursively to find a node with the properties being used by the conflict node. If one is found, return 1. If this second loop finishes without finding a match, return 0. The depth of recursive calls is limited by the parameter *depth*.

number()

```
static int number(char **ptr, int *num)
```

Args:

ptr pointer into a node specification string; it is updated

num RETURN: the integer found is returned in the location pointed to

Returns:

0 A valid integer was found in the node spec location pointed to by ptr

1 No integer found

-1 An integer of value zero was found, not legal in the node spec

The next token in the node spec is checked to see if it is an integer. The pointer to the node spec is updated to point beyond the integer.

property() of node

```
static int property( char **ptr, char **prop)
```

Args:

ptr pointer into a node specification; will be updated

prop RETURN: pointer to the located valid properity name in the node spec

Returns:

Zero if a valid properity name was the next token, 1 if not.

To be a legal properity name, the first character of its name must be alphabetic, the remaining chacters must be alphanumeric or '-' or '.'. The next token in the node spec is checked to see if it is an legal properity name. The pointer to the node spec is updated to point beyond

the name.

proplist()

```
static int proplist(char **str, struct prop **list)
```

Args:

str A pointer into a node specification, updated

prop RETURN: a pointer to a new property list is return

Returns:

Zero on success, 1 else

Starting at *str*, the next element in a supplied node spec is checked by calling *properity()* to see if it is a properity name. If it is, a new element in a generated property is allocated and filled in. If it is not a valid property name, 1 is returned. The processing is stopped at the first invalid property name or at the colon, ":", that ends the node spec section.

listelem()

```
static int listelem(char *str, int order)
```

Args:

str pointer into a node specification, it is updated

order The order of this node spec within the total specification

Returns:

1 if the node spec can be satisfied

0 if the node spec cannot be completely satisified

-1 if the node spec is impossible to satisify ever.

This function handles a singular node specification. It checks for a leading number, *number()*, followed by a sequence of properties *(proplist()*, and creates a list for each one.

The number of nodes in the total pool which have the required set of properities is counted via calling *hasprop()* on each node. If the number of nodes with the properties is less than the requested number, -1 is returned. If sufficient nodes are available, +1 is returned.

If neither of the above cases are true, an addition search is made via *search()*, ignoring none of the nodes (checking allocated/down ones) to see if the request can be satisifed if all were free.

mod_spec() nodes

```
static char *mod_spec(char *spec, char *global)
```

Args:

spec pointer to a node specification

globalpointer to a properity to add

Return:

a pointer to a modified node specification string

The properity given by *global* is appended to each node specification section within the *spec*. I.e. with a global value of `general` and a node spec of `2:propA:propB+3:propC` a new spec of `2:propA:propB:general+3:propC:general` is returned.

nodecmp()

```
int nodecmp(void *aa, void *bb)
```

Args:Both aa and bb are pointers to pbsnode structures

Returns:

The comparison relationship is returned

This routine is passed as the comparison function for the general C lib sort routine. It orders nodes by

- free nodes first if the global variable *exclusive* is set, or
- shared nodes first if exclusive is not set.

When assigned nodes, we want to assign matching free nodes for exclusive use and match nodes already shared for shared use.

node_spec()

```
int node_spec(char *str, int early)
```

Args:

str pointer to node specification string

early flag to quit test early

Returns:

>0 number of nodes required to meet spec, if they are available

0 if cannot currently be satisfied

-1 if cannot ever be satisfied

We assume unless the key word `shared` is found that the node request is for exclusive allocation, so the global variable *exclusive* is set by default. If any *global* properities are specified at the end of the spec, they are checked for the key word `shared`; if found, *exclusive* is cleared.

ctnodes() is used to determine the total number of nodes specified in the spec. It that is greater than the total number of nodes, we bail out with a -1.

The nodes are sorted by free or shared depending on the setting of exclusive. The *flag* field of each node is cleared to {okay}. If the node is free, it is counted in a count of nodes, *svr_numnodes*.

The node spec is checked by calling *listelem()* which also tentively allocates nodes matching the subspec by marking them `thinking`.

If any node is marked for allocation with `thinking`, but is not available to the job (already in use), then *search()* is used to attempt to find a replacement. This may entail given up a node already marked thinking which matches the empty spec and finding a replacement node for the one surrendered. A complex problem.

setup_nodes()

```
int setup_nodes()
```

Returns:

zero on success, -1 otherwise.

Open and read the `(PBS_HOME)/server_priv/nodes` file. Allocate structures for pbsnodes and props as requried. The total number of nodes in the file is maintained in *svr_numnodes*.

Each primary host name is validated by calling gethostbyname(). The IP address for the node is recorded in the node structure.

The state of each node is initialized to {INUSE_UNKNOWN} until the server is able to check with pbs_mom on that node.

set_nodes()

```
int set_nodes(job *pjob, char *spec, char **rtnlist)
```

Args:

pjob pointer to a job to which nodes are to be assigned

spec node specification required by that job

rtnlistRETURN: a list of allocated nodes (if possible) will be returned here.

Returns:

Zero if ok, or a PBS error number if not.

This function allocates nodes to a job. The requirement is given in the node specification *spec*.

The nodes to allocate are chosen by calling *node_spec()*. If the return indicates the request cannot be satisified currently, [PBSE_RESCUNAV] (temporarily unavailable) is returned, if the return from node_spec() indicates the request can never be satisified, [PBSE_BADATVAL] is returned.

If *exclusive* is set, the number of allocated nodes is deducted from the total number available, *svr_numnodes*. Each node selected by node_spec() is marked in the *flag* field with the flag {thinking}, each of those nodes is marked as being allocated to the job either as shared, {INUSE_JOBSHARE}, or exclusively {INUSE_JOB}. A pointer to the job is linked into the node structure. Note, a share node may be allocated to more than one job.

The list of nodes is ordered to match the specification given. This was carried around in the *order* field. The list is a string of the form: `node1+node2+node3+...`

node_avail()

```
int node_avail(char *spec, int *avail, int *alloc, int *reserved, int *down)
```

Args:

spec pointer to a node spec

avail RETURN: pointer to a integer in which the number of available nodes that match the spec is returned.

alloc RETURN: pointer to a integer in which the number of allocated nodes that match the spec is returned.

reserved
RETURN: pointer to a integer in which the number of reserved nodes that match the spec is returned.

downRETURN: pointer to a integer in which the number of down nodes that match the spec is returned.

Returns:

Zero on success or PBS error number.

This is the node specific part of a batch Resource Query request, see *pbs_rescquery()*. The node specification may come in two flavors:

simple
The request is of the form `nodes` or `nodes=` and covers all possible nodes; or the request deals with a single set of properities, `nodes=prop[:prop...]` in which case the numbers returned concern the number of nodes with those properities. All four numbers are valid. The above is determined by calling *hasprop()* against each known node. If the node has the requested properities, the count of available, allocated, ... is incremented depending on the node state.

complex
The request is of the forms: `nodes=number` or with multiple nodes `nodes=prop[:prop]+prop...` In this case, only the *avail* number has meaing and it is kludged. If greater than zero, it is the number of nodes requested by the spec and some set of nodes is currently available which would satisify the spec. If equal zero, the spec is possible, but some node or nodes are currently allocated/reserved/down. If *avail* is -1, the spec could never be satisfied. This is determined by calling *node_spec()* with the spec and setting *avail* to its return value. Note, the number of available nodes, *svr_numnodes* would be reduced by node_spec() and must be reset since the nodes are not actually assigned.

node_reserve()

```
int node_reserve(char *spec, resource_t tag)
```

Args:

spec another node spec

tag A resource reservation handle

Returns:

>0 if the reservation was made

0 if the reservation was not made or was made in part but may be satified later

-1 if the reservation could never be made

This is the node specific piece of the Resource Reserver batch request, see *req_rescreserve*().

If this is a reservation that had been attempted before (was partially satisfied), then *tag*will {RESOURCE_T_NULL} and the nodes currently reserved for that tag are freed by calling

node_unreserve() This allows us to reallocate them (or differents ones as the case may be).

The routine *node_spec()* is called to determine if the nodes requested are available. If they are, the {thinking} nodes are reset to {INUSE_RESERVE}. If the reservation cannot be currently satisfied, those nodes which are {thinking} and {INUSE_FREE} are reserved as above.

free_nodes()

```
void free_nodes(job *pjob)
```

Args:

pjob pointer to a job struture

Any node with the given job in its allocated to job list has that job removed. If and only if the job list becomes null, is the node marked free.

ping_nodes()

```
void ping_nodes(struct work_task *ptask)
```

Args:

ptaskpointer to a work_task structure

This routine is called off of the server's work task list. It is used to *ping* Mom on nodes periodically to see if they are alive.

If the node is down, or in use by a job it is not pinged. When a down node comes up, its Mom should yell at the server. If required, an RPP stream is setup to Mom on the node. *is_compose()* starts a message to the Mom and *rpp_flush()* sends it. If there is a failure, the RPP steam is closed and the node marked {INUSE_DOWN}. Note, there is no reply to this ping message, if the stardard RPP handshaking acknowledges receipt of the message, that tells us it is up.

A new work task is set for 300 seconds later.

set_old_nodes()

```
void set_old_nodes(job *pjob)
```

Args:

pjob pointer to job structure

This routine is called on the server's startup from *pbsd_init*(). It looks at the nodes assigned to running jobs in the attribute JOB_ATR_exec_host and calls *set_one_old()* to mark that node as in use and allocated to this job. The job attribute JOB_ATR_resource is scanned for the resource `neednodes`. If found (and set), a search is made for the global property `shared`. If found, then the nodes allocated to the job are marked as {INUSE_JOBSHARE}, else they are marked {INUSE_JOB}.

set_one_old()

```
static void set_one_old(char *name, job *pjob, int shared)
```

Args:

nameof the node to mark as belong to the job

pjob pointer to the job

shared

either {INUSE_JOB} or {INUSE_JOBSHARE}

This is a helper routine for set_old_nodes(). The list of pbsnode structures is scanned for a node with this name. Note, the node **name** is the last property in the prop list. The node is marked in use with the value of *shared* and the job pointer is added to the list of jobs allocated to the node.

5.3.11. Server Batch Request Functions

The functions in the following sections perform the processing required for batch requests received from clients, including other servers.

The first item of business in processing each job related task is to determine if the requesting user has the authority to make the request. This is done by calling *svr_authorize_jobreq()*. If the request is not a job related request, then that request will use another mechanism.

For job related requests, unless otherwise specified, the request must be rejected if the job is in the {JOB_STATE_EXITING} state.

The last item of business required for each batch request function is the generation and issuance of a reply to the client.

5.3.11.1. req_quejob.c

The file *src/server/req_quejob.c* contains the functions associated with the sequence of batch requests that request a server to create (queue) a job. The job may be a new job, the requesting client is qsub(1)/pbs_submit(3). Or the job may be an existing job, the client is a another server routing the job to this server.

req_queuejob()

```
void req_queuejob(batch_request *request)
```

Args:

request

pointer to the batch_request structure containing the request.

This request is to create a new job or to transfer a job from one server to another. The destination, a queue name, for the job is specified in the request. When a job is being transferred (routed), the job identifier will be specified in the request and the client must be another server. A null user name in the credential indicates the client is another server. If the job is not from another server, it cannot have a job id specified in the request. If the job is from a user client and thus being created here, the next job sequence sv_jobidnumber, number is assigned. Together with the server name, this is the job id.

Both the list of new (inbound) jobs and existing jobs is searched for a job with the same id. If found this is a serious error.

The destination queue is validated. If it does not exist or is not enabled (receiving new jobs), an error reply is returned to the client.

It would be nice to be able to use the job id as the base file name under which the job information is maintained. However, since the job name contains the server (host) name, it can be quite long; longer than the 14 characters guaranteed by POSIX. Hence, we make up a name which is the job name shorted to 11 characters, 14 - 3 for the ".JB" suffix. Unfortunately, this name might collide with one from a different server whoses name starts the same. The made up, or hashed name is opened. If one already exists, the name is changed starting with the eleventh character and working toward the first until a unique name is created. The 11 character basename is recorded in the job structure.

The routine *job_alloc()* function is called to allocate and initialize the job structure. Error replies are returned if the job cannot be created or already exists.

Each attribute in the request is decoded via the appropriate at_decode() function into a local copy of the job attribute array. If any attribute name is unknown to the server, it is maintained in the special unknown attribute list. If any attribute fails to decode correctly, an error is returned. The *auxcode* field in the reply identifies the attribute in error. On any error, the job is purged from the server.

When all supplied attributes, including resources, are successfully decoded, the job attributes are updated by "setting" them to the decoded values.

If the job is being created by this server, the job-owner attribute is set to the client user name, the ctime (create time) attribute is set to the current time, and the hopcount attribute is initialized to one.

Otherwise, if the job is being routed here, not created here, then if the job-owner attribute was not been passed with the request, the request is rejected. The hopcount is incremented and if too big, the request is rejected.

If the destination queue is an execution queue, the job execution uid and gid are set by calling *set_jobexid()*. This has the side effect of checking any queue access control list; the user must have access rights or the request is rejected. For security reasons, no batch job is allowed to be submitted or run with the uid of zero (0); it might allow a user to crack security and submit a job which would cause root-rot.

In addition to the attributes, the following fields in the job structure are set: ji_state, ji_substate, ji_svrflags, ji_numattr, ji_ctime, ji_un_type (to {JOB_UNION_TYPE_NEW}), ji_jobid, ji_quen, ji_euid, ji_egid. The job state is set to {JOB_STATE_TRANSIT} and the substate to {JOB_SUBSTATE_TRANSIN}.

If any error occurs after the job structure has been allocated, the request is rejected and the job structure is freed via *job_purge()*. Job_purge must be used rather than job_free() because the control (save) file has been created.

The job structure is linked into the server's new job list, sv_newjobs. A success reply is returned to the client.

req_jobcredential()

```
void req_jobcredential(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

In the standard PBS release, this routine is a stub which will reject the request. It is provided to allow a site or vendor to add support for Kerberos or AFS (Andrew File System) where access tickets must be passed with the job.

req_jobscript()

```
void req_jobscript(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

The job's script is passed by one or more jobscript requests. The amount of data in each request is limited to just under 8KB. This allows the use of UDP protocol if anyone ever cares to implement PBS on it. Since the Job Script request must follow a Queue Job request, the network connection table has already been set up with a pointer to the job structure. This pointer is used to locate the job for which the script is intended.

The size of the script file is maintained in the job structure. If the size is zero when a jobscript request is received, we assume that the request must be the first and create the script file. Otherwise, we open the file with {O_APPEND}. The script file name is based upon the job control file name with a different suffix, " .SC".

The script data is written to the file, the file is closed, the file size in the job structure is updated, and an reply is returned to the client.

req_rdytocommit()

```
void req_rdytocommit(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

When this request is received, we know the client has completed sending all data for the job. The job is manually marked in state {JOB_STATE_TRANSIT} and substate {JOB_SUBSTATE_TRANSICM}. The state is set manually to prevent the server and queue (which the job is not yet in anyway) from being updated. The job structure is saved in the job file by calling *job_save()*. It will remain in substate {JOB_SUBSTATE_TRANSICM} until the Commit Request is received.

req_commit()

```
void req_commit(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

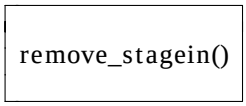
When this request is received, the job should reside in the server's new job list and be in substate {JOB_SUBSTATE_TRANSICM}. This request tells us that the client is giving up control of the job to us. The job state and substate are updated to reflect the setting of certain attributes, see *svr_evaljobstate()*. Typically, the new state will be either {JOB_STATE_QUEUED}, {JOB_STATE_HELD}, or {JOB_STATE_WAITING}.

The JOB_ATR_qrank, queue_rank, attribute is set from the global variable *queue_rank*. This is used to insure the job will be ordered in the queue in the correct place on a restart of the server. The job is placed into its destination queue and the various state counts are updated by calling *svr_enquejob()*. The job file is "quickly" updated by calling *job_save()*. It is now ready for processing depending on the queue type.

If the job was not created here and is not a new job, then the server calls *issue_track()* to notify the tracking server of the job's new location.

5.3.11.2. req_delete.c

The file *src/server/req_delete.c* contains the functions used in processing a delete job batch request.




```
void remove_stagein(job *pjob)
```

Args:

pjob pointer to job which has had files staged in.

When a file has had files staged in but not yet run and the job is to be deleted or moved, the staged files should be removed to restore the system to the state before the job was submitted.

A delete files request is built by calling *cpy_stage()*, see req_jobobit.c, and the request is sent to MOM via *relay_to_mom()*. Note, only one try is made, win or lose. The request structure is freed after the send by *release_req()*.


```
void req_deletejob(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

If the job is in the {JOB_STATE_TRANSIT} state (outbound) and a job routing process has been forked and recorded in a work task entry, a pointer to the delete job batch request structure is recorded in a new work_task entry with a processing function of *post_delete_route()*. The routing process is sent a {SIGTERM} signal. The abort processing is continued when the routing process terminates, the death of child processing locates the work task entries and places them on the immediate work list. The work task entry for the routing child will be processed before the entry with the *post_delete_route()*. If the router process returned an exit status of zero, the job was routed to another server before it could be deleted, and the job is purged from this server. If however, the router exit status was non-zero, then the job is still ours to delete. See what happens in *post_delete_route()*.

If the job is substate {JOB_SUBSTATE_PRERUN}, then we need to wait for MOM to finish receiving the job so we can delete it, otherwise there is a race condition and the runjob command may hang. Therefore, the delete job request is placed in the work task for one second later by calling *set_task()* with a timed event pointing to *post_delete_route()*.

Otherwise, if the requesting client is not the job owner, then send mail to the user to inform him of the delete. If the request extend field, rq_extend is a non null pointer, and the text to which it points does not start with `deldelay=`, then the text is a message (from the Scheduler) which is appended to the mail message.

If the job is in the state of {JOB_STATE_RUNNING} then the function *issue_signal()* is called to send a Signal Job request to the MOM responsible for the execution of the job requesting a {SIGTERM} signal be sent to the job. The address of the client batch_request structure is passed to issue_signal() so that it can be found and completed with MOM responds. Likewise, the function *post_delete_mom1()* is passed to issue_signal() as the reply processor. The real work continues within post_delete_mom1().

If the job has a non-migratable (Cray style) checkpoint image as shown by *ji_svrflags* containing {JOB_SVFLG_CHKPT}, then job exit processing is performed to deliver the output and remove the job files under MOM's control. The job is set to state {JOB_STATE_EXITING} and substate {JOB_SUBSTATE_EXITING}. The variable *ji_momhandle* is set to -1 force *on_job_exit*() to obtain a new connection to MOM and a work task entry is created to invoke on_job_exit() immediately.

If the job has files that have been staged in already, marked by setting {JOB_SVFLG_StagedIn} in the job structure field ji_svrflags, then *remove_stagein()* is called to ask MOM to delete the files and the job is aborted via *job_abt()*.

If the job is in any other state, *job_abt()* is called to dispose of the job immediately and a reply is generated for the request.

post_delete_route()

```
static void post_delete_route(struct work_task *pwt)
```

Args:

pwt pointer to the work_task entry whoses dispatch resulting in calling this function.

All that need be done is to recall the *req_delete()* function. The work_task member, wt_parm1 contains a pointer to the original Delete Job batch request. and it will either (1) find the job has been requeued by the router when it received the signal, or (2) the job was already gone (and now forgotten) in which case [PBSE_UNKJOBID] is returned and the client can look elsewhere.

post_delete_mom1()

```
static void post_delete_mom1(struct work_task *pwt)
```

Args:

pwt pointer to the work_task entry whoses dispatch resulting in calling this function.

Here we continue the work started in req_deletejob() for a job in the running state. The work task pointed to by pwt is the one especially created to send to MOM. It the rq_extra field is a pointer to the original client request. If MOM did not reject the signal request, we can ac-

knowledge the client request. (If we wait till after the job is signaled a second time, coming up, the user may feel the delay is too long.) Note, at this point, the original request is gone. We now build a new work task entry for the time delay, (1) either specified in the original request in the request extension, (2) the queue attribute kill_delay, or (3) 2 seconds (why 2, why not?). This new work task points to the function *post_delete_mom2()*, which will continue the work and points to not the batch request, but directly to the job.

post_delete_mom2()

```
static void post_delete_mom2(struct work_task *pwt)
```

Args:

pwt pointer to the work_task entry whoses dispatch resulting in calling this function.

When we get here, it is time to send the {SIGKILL} signal to the job, if the job still exists in the running state. We will assume that MOM will accept the signal request, so just pass *release_req()* as the post processing function to *issue_signal()*.

Once the job dies, normal job exit processing will occur.

5.3.11.3. req_holdjob.c

The file *src/server/req_holdjob.c* contains the functions to process the Hold Job and Release Job requests.

chk_hold_priv()

```
int chk_hold_priv(long value, int privilege)
```

Args:

valuethe hold value specified.

privilege
of the calling client.

Returns:

0 if ok, PBS error number otherwise.

This routine checks that the client has the required privilege for setting a hold:

HOLD_u
No special privilege is required.

HOLD_o
Operator, {ATR_DFLAG_OPWR}, or manager, {ATR_DFLAG_MGWR}, privilege is required.

HOLD_o
Manager privilege is required.

req_holdjob()

```
void req_holdjob(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

The hold types specified in the request are determined by calling a private routine *get_hold()* which finds the holds to be set, decodes them, and checks the privilege required against the clients. These holds are then added to the job JOB_ATR_hold attribute. (This should be done by calling at_set(), but I cheat and set them directly.)

If the job is in {JOB_STATE_RUNNING} state, and if checkpoint is supported by the server, and if the checkpoint attribute is not n, then the following additional actions are taken:

- a Hold Request is sent to the MOM which is shepherding the job in execution. Upon receipt of the Hold Request, MOM will attempt to checkpoint the job and terminate its execution.
- If MOM returned a reply indicating she was successful in checkpointing the job, the job substate is set to {JOB_SUBSTATE_RERUN} to cause rerun post job processing, and the job is retained in the execution queue. Note, the job is left in run state until MOM aborts the job and notifies us with the Job Obit notice.

  If MOM returned a reply indicating that the checkpoint failed, the error reply is returned to the client requesting the hold. If the reply from MOM indicated that checkpoint was not supported on the execution host, the job is left in execution state, with the hold noted just as if checkpoint was not supported by the server.

If checkpoint is not supported, or if the checkpoint attribute is n, no additional processing is performed, the job is left executing.

req_releasejob()

```
void req_releasejob(batch_request *request)
```

Args:

request
pointer to the batch_request structure containing the request.

The hold types specified in the request are determined by calling a private routine *get_hold()* which finds the holds to be released, decodes them, and checks the privilege required against the clients. Each hold type specified in the request is removed from the job hold-list attribute by calling the *at_decode()* routine and clearing the corresponding bit in JOB_ATR_holdvia *at_set()* routine for the attribute.

get_hold()

```
static int get_hold(list_head *head, char **pstring)
```

Args:

head of list of svrattrl structures containing the attributes from the request.

pstring
RETURN: pointer to a string pointer which will be set to point to the hold characters, n, u, o, and/or s.

Returns:

0 if ok, error otherwise.

The Hold_Types attribute in the supplied list is located, there should be one and only one such attribute, otherwise the hold or release request was ill formed. The character pointer, *pstring*, is set to point to the attribute (external) value. The attribute is decoded into a temporary attribute which is available to the routine routines in this file.

post_hold()

```
static void post_hold(struct work_task *pwt)
```

Args:

pwt Pointer to the work task entry.

This routine is called to when MOM responds to the Hold Job request passed to her from *req_holdjob*() when checkpointing is supported by the server. If MOM returns an error indicating that checkpoint failed (including not supported), it is logged and the error is returned to the client that initiated the Hold Request. The job state is restored to {JOB_SUBSTATE_RUNNING} since the job is still running.

If checkpointing succeeded, *ji_svrflags* is updated with {JOB_SVFLG_HASRUN} and either {JOB_SVFLG_CHKP} (for Cray style non-migratable checkpoint) or {JOB_SVFLG_ChkptMig} (for yet non implemented migratable checkpoint). *account_record()* is called with {PBS_ACCT_CHKPNT} to record the checkpoint suspension in the accounting file.

The Hold Job request is acknowledged.

5.3.11.4. req_jobobit.c

The file *src/server/req_jobobit.c* contains the function to process the Job Obituary batch request and associated post-execution functions. The Job Obituary request is actually a notice from MOM that the referenced job has terminated execution. Generally, if the job terminated, post processing is performed to return output and remove the job, see *on_job_exit()*. If the job is to be rerun, the job is requeued in its current queue, see *on_job_rerun()*.

There are several special cases of job termination which are handled.

- If MOM dies, either on her own or because the system crashed, MOM has lost control of the executing jobs. Either they died also or they became detached from MOM. When MOM recovers, she will attempt to kill all jobs and mark them as exited. She will insert a special exit status code of {JOB_EXEC_INITABT} to be returned to the server as the job exit status. This exit status notes the job died/was killed on recovery. The server will rerun the job if allowed or terminate it if not.
- If MOM aborted the job on recovery and the job had a Cray style non-migratable checkpoint file, mom returns a special job exit code of {JOB_EXEC_INITRST}. The job is marked in *ji_svrflags* with {JOB_SVFLG_HASRUN} and {JOB_SVFLG_CHKPT}. The job state is simply requeued.
- If MOM aborted the job on recovery and the job had a as yet unimplemented migratable checkpoint image, mom returns a job exit status of {JOB_EXEC_INITRMG}. The job is marked in *ji_svrflags* with {JOB_SVFLG_HASRUN} and {JOB_SVFLG_ChkptMig} and its substate is set to {JOB_SUBSTATE_RERUN} to cause rerun processing.
- If MOM is unable to start a job for some reason that is permanent, i.e. the user account was invalid or the job asked for an unknown resource, then MOM will set the job exit code to either {JOB_EXEC_FAIL1} if the error was detected before the standard output of the job was created or if the error was noted after the standard output was set up. In both cases

the server will abort the job; the difference is the message mailed to the user.

- If MOM is unable to start a job for some reason that is believed to be temporary, such as a resource has be gobbled up by an interactive session, then MOM will set the job exit code to {JOB_EXEC_RETRY}. The server will requeue the job; it is treated as a rerun except that the job's output is not saved.

wait_for_send()

```
static void wait_for_send(struct work_task *ptask)
```

Args:

ptaskpointer to the work task entry that called this routine.

This routine just calls back *req_jobobit()*. The work task was set up there as a delay mechanism.

req_jobobit()

```
void req_jobobit(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

This function validates the request, updates the list of resources used, records the job exit status in ji_exitstat, and replies to MOM. The scheduling flag, svr_do_schedule, is set to {SCH_SCHEDULE_TERM}. If the job cannot be found and the server was initiated {RECOV_COLD} or {RECOV_CREATE}, then the jobs were blown away. The server replies to MOM with [PBSE_CLEANEDOUT] to instruct her to trash her files relating to that job. Otherwise if the job cannot be found, the server returns {PBSE_UNKJOBID}.

If the job is already in the {JOB_STATE_EXITING} state, then MOM must be recovering and sending the server a second notice. Return [PBSE_ALRDYEXIT] to MOM which tells her to mark the job as exiting and close the connection. The server will continue to process the job on the thread started by the original notice.

If the job state is not {JOB_STATE_RUNNING}, an obit should never have been issued, so this logged and ignored. Otherwise if the substate is {JOB_SUBSTATE_PRERUN}, then the obit notice "won the race" condition between it and the SIGCHLD from the child of the server that sent the job to MOM to run, see *svr_strtjob2()* in req_runjob.c. We need to wait for the send side to complete so the run job request can be acknowledged. So a work task with a one second delay is created to call *wait_for_send()*. It will just restart req_jobobit().

The information in the request is processed first, saving the status and building the mail/log message before replying to MOM, otherwise the information is lost. The reply is then made to keep MOM from waiting any longer.

A normal exit status from a job can never be negative, since only 8 bits is return. If the exit status of the job is negative, it is a special status from MOM, and is one of the following:

JOB_EXEC_INITABT

The job was aborted by MOM on her recovery. If the job can be rerun, its substate is set to {JOB_SUBSTATE_RERUN} and the {JOB_SVFLG_HASRUN} flag is set in *ji_svrflags*. Rerun processing will take place.

JOB_EXEC_RETRY

MOM was unable to start the job, but it should be retried. If the job has been rerun before and has output files, this case is treated as another rerun. If the job has not be run before, the empty output files are not saved, but other rerun processing is performed. This is accomplished by setting the substate to {JOB_SUBSTATE_RERUN1}, see on_job_rerun().

all other

A special mail message is sent to the owner and normal exit processing takes place.

If the exit value is greater than 10000, then the job ended on a signal. 10000 was added by MOM to the signal number. Different executions hosts may have different size exit masks in wait.h, so the signal value is forced to be uniform. This allows us to issue a different mail message to the user on job end.

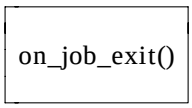
If the job is being terminated, not rerun, then the job state is set to {JOB_STATE_EXITING} and the substate to {JOB_SUBSTATE_EXITING}. If requested, mail is sent to the mail_list by calling *svr_mailowner()*. The new-lines in the resource usage message are replaced with spaces for the log entry. The function *account_jobend()* is called to record the usage to the accounting file. If {PBSEVENT_JOB_USAGE} is sent in the server's log_events attribute, then the same message is recorded in the log, otherwise a short form is recorded.

The function *on_job_exit()* is called with a pointer to a work task of type WORK_Immed.

If the job is being truly rerun, not restarted from checkpoint, then the resources used attribute, JOB_ATR_resc_used, and the execution host attribute, JOB_ATR_exec_host, are cleared. Then the function *on_job_rerun()* is invoked with a work task entry of WORK_Immed.

If the job has a Cray style checkpoint file, {JOB_SVFLG_CHKPT} is set, the job is requeued directly.

As a reminder, both on_job_exit() and on_job_rerun() are invoked via *dispatch_task()* so that the work task structure is deleted when the on_job_* function returns.




```
void on_job_exit(struct work_task * ptask)
```

Args:

ptask pointer to a work task entry which points to the job in exit state.

The steps required for a normally terminating job are:

1. Set the job state to {JOB_STATE_EXITING} and the substate to {JOB_SUBSTATE_EXITING}. Deal with any job termination dependences.
2. Set the substate to {JOB_SUBSTATE_STAGEOUT}. Send a Copy Files request to MOM to move the standard output, standard error, and any "staged out|*U files of the job and the files listed in the *stage out* resource to the final destination.
3. Set the job substate to {JOB_SUBSTATE_STAGEDEL}. Send a Delete Files request to MOM to delete any "staged in" files.
4. Set the job substate to {JOB_SUBSTATE_EXITED}. Send a Delete Job request to MOM to remove remaining traces of job including the job control file, job script, and any checkpoint file.
5. Send the final Track Job request to the original server (if not me) and purge the job from the system.

This function is invoked by *req_jobobit()*, by work task when various copy and delete file requests to MOM complete, and by *pbsd_init()* on recovery. Its purpose to determine where in

the exiting processing the job is and resume with the next step. If the work task type is {WORK_Immed} or {WORK_Timed}, this routine is being called to perform the cycle or stage of processing the job indicated by its substate. Otherwise, the work task type is {WORK_Deferred_Reply}, and this routine is being called back after MOM has been replied to a request. A {WORK_Timed} call back results when on_job_exit() is called from *pbsd_init()*. In this case, as with the first time on_job_exit() is called for a job, there is not a connection to MOM and one must be made by calling *mom_comm()*. If the connection cannot be established, MOM is not (yet) alive, a time delay work task is created to retry MOM after a delay.

Switch on the job substate:

JOB_SUBSTATE_EXITING or JOB_SUBSTATE_ABORT

Process any dependencies by calling *depend_on_term()*. Advance the job substate to JOB_SUBSTATE_STAGEOUT.

JOB_SUBSTATE_STAGEOUT

If the work task type is {WORK_Immed}, then this is the first call into this routine. The first task is to determine which of the standard job files (output and error) are to be moved.

- If the job attribute JOB_ATR_join is set to other than n, then determine which file is listed first (exists and will be moved) and which are joined into that one (does not exist).
- For each file not joined to another file, determined if it is to be kept by checking the job attribute JOB_ATR_keep. If kept, add to the Copy File list with the destination (remote) name set to the default file name.
- For each file not joined and not kept, add the file to the the Copy File request with the the destination name set to the name given in the corresponding path attribute.

If the job has a stage-out resource, then append thoses files to the Copy Files request. Then send the request to MOM.

If the work_task type is not {WORK_Immed},
MOM has replied to the Copy File request. If the reply indicates a failure, generate a mail message to the job owner and set the substate to {JOB_SUBSTATE_EXITING}.

Regardless of the reply from MOM, free the prior batch request, set the substate to {JOB_SUBSTATE_STAGEDEL} and set up a work task of type {WORK_Immed} and pointing back to on_job_exit. On being dispatched, the appropriate next action will be performed.

JOB_SUBSTATE_STAGEDEL

If the work task type is {WORK_Immed}, this is the first time into this section. Build and send MOM a Delete Files request for each file that was “staged in”.

If the work task type is not immediate and If the reply indicates a failure, generate a mail message to the job owner. Free the the batch request, set the job substate to {JOB_SUBSTATE_EXITED} and continue with that action.

JOB_SUBSTATE_EXITED

Send a Delete Job Request to Mom. Send the final Track Job request to the creating server if that is not here. Call *job_purge()* to remove the job.

on_job_rerun()

```
void on_job_rerun(struct work_task * ptask)
```

Args:

ptaskpointer to a work task entry which points to the job in exit state.

This function requeues a job when it stops following a rerun request. The substate of the job has already been set to {JOB_SUBSTATE_RERUN} by *req_rerunjob()*. At the end of processing, the job state is reset to {JOB_STATE_QUEUED}, and the substate to {JOB_SUBSTATE_QUEUED}. The job is left in the current queue. The actions on the job are driven by the substate recorded in the job.

JOB_SUBSTATE_RERUN

On the first entry in on_job_rerun(), the substate will already be set to {JOB_SUBSTATE_RERUN}, and the work task pointer will be of type WORK_Immed. *mom_comm()* is called to obtain a connection to MOM. If the host on which the job was executing is the server host, no file action is required. The job state is set to {JOB_STATE_EXITING} and substate to {JOB_SUBSTATE_RERUN1}. A work task entry is created to pick up at that point in post processing.

If the execution host is not the server host, then the various files must be recovered to the server in case the job is rerun elsewhere. A **Rerun Job** request is sent to MOM. This directs her to return standard output, standard error, and any checkpoint file to the server using a **Job Files** request.

If MOM responds with success, the job state and substate are set to {JOB_STATE_EXITING} and {JOB_SUBSTATE_RERUN1} and the server proceeds with the next step.

JOB_SUBSTATE_RERUN1

If there are file to be staged-out, the server builds a **Copy Files** request, see *cpy_stage()*, and the server sends it to MOM. Note, MOM will delete any files she stages out. Regardless of sucess or failure, the substate is updated to {JOB_SUBSTATE_RERUN2}.

JOB_SUBSTATE_RERUN2

If the job had files staged-in, *cpy_stage()* is called to build a copy files request for those files and the request is converted to a **Delete Files** request which is sent to mom. If there are no staged-in files to delete or after the request is prcessed (success or failure), the job substate is updated for the next phase.

JOB_SUBSTATE_RERUN3

The job is removed from MOM's custody by sending her a Delete Job request.

The socket handle, ji_momhandle and the {JOB_SVFLG_StagedIn} flag in ji_svrflags are cleared. The new job state and substate are determined by calling *svr_evaljobstate()* and set by *svr_setjobstate()*. In effect, this requeues the job.

mom_comm()

```
int mom_comm(job *pjob, void (*function)(struct work_task *))
```

Args:

pjob pointer to job structure.

function

to invoke via a work task if the connection to MOM cannot be established.

Returns:

the connection handle or -1 if no connection was established.

If a handle has already be recorded in ji_momhandle of the job structure (not -1), it is returned. Otherwise, a new connection to MOM is established by calling *svr_connect()* with the host address of MOM found in ji_un.ji_exect.ji_momaddr. If this address is zero (which might be the case if the ji_un union was cleared on by moving the job), the address of the host in the

JOB_ATR_exec_host attribute is obtained prior to calling svr_connect(). If the connection is established, the handle is saved in ji_momhandle and returned to the caller.

If the connection cannot be established, a work task structure is set up with a time delay of {PBS_NET_RETRY_TIME} and a call back function as passed in the parameter function.

setup_from()

```
static char *setup_from(job *pjob, char *suffix)
```

Args:

pjob pointer to job structure

suffixto append to file based on output, error, or checkpoint.

Return:

Pointer to allocated string containing file name.

This function returns a name for a standard file for a job. The suffixes are defined in job.h as: {JOB_STDOUT_SUFFIX} − .OU for standard output, {JOB_STDERR_SUFFIX} − .ER for standard error, and {JOB_CKPT_SUFFIX} − .CK for checkpoint.

setup_cpyfiles()

```
static struct batch_request *setup_cpyfiles( struct batch_request *preq,
job *pjob, char *from, char *to, int direction, int flag)
```

Args:

preq Pointer to copy file request structure to build. If null, the structure will be allocated, otherwise the existing one will be expanded.

pjob pointer to job.

from name of file local to mom.

to name of file remote to mom (destination on stage-out, source on stage-in).

direction
of transfer: {STAGE_DIR_IN} − in to Mom, or {STAGE_DIR_OUT}.

flag indication type of file: {STDJOBFILE} − standard job file (output, error), {JOBCKPFILE} − checkpoint file, or {STAGEFILE} − user specified stage-in/out file.

Return:

Pointer to the copy files batch request structure.

If *preq* is null, then a batch_request structure is allocated and initialized for {PBS_BATCH_CopyFiles} including the job id, owner, effective user and effective group names. Note, if the effective group is the user's login group as indicated by {ATR_VFLAG_DEFL} set in the JOB_ATR_egroup attribute, the group name is set to the null string. This tells mom to use the gid from the password entry. If the *preq* is not null, the existing copy files request structure is used by appending the new file pair to the current list.

A file pair structure, rqfpair, is allocated and initialized with the *from* and *to* names and the file type flag which is an indication to Mom as to where the local file is/should be.

is_joined()

```
static int is_joined(job *pjob, enum job_atr nat)
```

Args:

pjob pointer to job

nat indicates which attribute the file name concerns: {JOB_ATR_outpath} or {JOB_ATR_errpath}.

Returns:

1 if file is joined to another.

0 if not joined.

This routine takes the number, nat, of a job attribute and determines if that file (output or error) is joined to another in the job's JOB_ATR_join attribute. Note in the case of "-j oe" option, the error file is joined to the output file. If nat was JOB_ATR_errpath, the return would be true (1).

cpy_stdfile()

```
static struct batch_request *cpy_stdfile(struct batch_request *preq,
job *pjob, enum job_atr nat)
```

Args:

preq Pointer to copy file request structure to build. If null, the structure will be allocated, otherwise the existing one will be expanded.

pjob pointer to job.

nat identifies attribute specifying output or error path.

Return:

Pointer to the copy files batch request structure.

This function determines if one of the job's standard files (output or error) should be copied. If so, it builds or adds to the copy files request.

If the job is interactive, there is no output to copy. Otherwise we choose the suffix and a default key letter based on the file. The key letter is used to check the keep list, JOB_ATR_keep. *is_joined()* is used to determine if the file was joined to another and doesn't exist separately.

The *to* file name is based on the job attribute value. The *from* name is returned from *setup_from()*.

The function *setup_cpyfiles()* does the rest of the work.

cpy_stage()

```
struct batch_request *cpy_stage(struct batch_request *preq, job *pjob,
enum job_atr nat, int direction)
```

Args:

preq Pointer to copy file request structure to build. If null, the structure will be allocated, otherwise the existing one will be expanded.

pjob pointer to job.

nat identifies attribute specifying path.

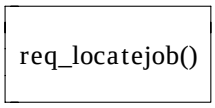
direction
of transfer.

Returns:
pointer to new or expanded copy files batch request.

This is the equivalent to cpy_stdfile() for stage-in/out files. If the attribute specified by nat is set, then for each *local_name@remote_host:remote_name* element in its value, the element is parsed into the *to* path, and the *from* path. *setup_cpyfiles()* does

5.3.11.5. req_locate.c

The file *src/server/req_locate.c* contains the function to process the Locate Job batch request.




```
void req_locatejob (struct batch_request *req)
```

Args:

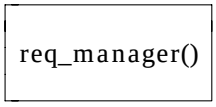
req pointer to the batch_request structure.

The function will attempt to find information about the job in two places. First, the server will search its list of all active jobs by calling *find_job()*. If that fails, the server will search the array of tracking records pointed to by the server structure member *sv_track*.

If found in either place, the current location is reported to the client in the reply. If not found, the server responds with [PBSE_UNKJID].

5.3.11.6. req_manager.c

The file *src/server/req_manage.c* contains the server function for processing the Manager batch request, creating and deleting queues and setting server queue and node attributes.




```
void req_manager(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

As this is not a job related batch request, the authorization is performed differently. The user's privilege is obtained. If the manage command is a *create* or *delete*, the privilege must be at the administrator level. If the manage operation is a *set* or *unset*, the privilege generally can be at either the administrator or operator level. The exception to this statement comes when dealing with *node-attributes,* where certain changes are only available to managers.

A function to perform the requested operation is now called. The function called is chosen based on the *command* and *object type* specified in the Manage request. The *command* values can be {Create}, {Delete}, {Set} and {Unset}. The object type values can be {Server} or {Queue.} It

is not legal to either create or delete a server.

If any error is detected, an error reply is returned to the client.

Each command — object specific function generates and sends a success or error reply to the client.

mgr_server_set()

```
void mgr_server_set(int sfds, struct batch_request *req)
```

Args:

sfds the socket connection to the requesting client.

req pointer to the batch_request structure.

The specified attributes of the server are set by calling *mgr_set_attr()* with the address of the server attribute array, the address of the server attribute definition array, the number of attributes in the array, the list of attributes from the batch request, and the privilege of the requester.

Svr_update() is called to save the server information to disk and *mgr_log_attr()* to log the changes in the log file. An appropriate reply is generated and sent to the client.

mgr_server_unset()

```
void mgr_server_unset(int sfds, struct batch_request *req)
```

Args:

sfds the socket connection to the requesting client.

req pointer to the batch_request structure.

The specified attributes of the server are set by calling *mgr_unset_attr()* with the address of the server attribute array, the address of the server attribute definition array, the number of attributes in the array, the list of attributes from the batch request, and the privilege of the requester.

Svr_upatedb() is called to save the server information to disk. The routine *mgr_log_attr()* is called to log the attributes changes. An appropriate reply is generated and sent to the client.

mgr_queue_create()

```
void mgr_queue_create(int sfds, struct batch_request *req)
```

Args:

sfds the socket connection to the requesting client.

req pointer to the batch_request structure.

Find_queuebyname() is called to insure a queue does not already exist with the specified name. Space for the queue structure is allocated and initialized by calling *que_alloc()*.

At this point the type of the queue is indeterminate. It is established by the first attribute found which is restricted to a certain queue type. The attribute list in the request is scanned for the first attribute whose definition contains a parent type flag of other than {PARENT_TYPE_QUE_ALL}. The queue takes on the queue type indicated by that attribute.

The function *mgr_set_attr()* is called to actually set the queue values. If successful, *svr_save()* and *que_save()* are called to write the queue save file and update the server's save file. A success reply is returned to the users.

If any attribute being set is incompatible with the queue_type as determined by calling *check_que_attr()*, a "warning" message is returned to the client. Any errors in the request will result in the queue structure being freed by *que_free()* and a error reply returned to the user.

mgr_queue_delete()

```
void mgr_queue_delete(int sfds, struct batch_request *req)
```

Args:

sfds the socket connection to the requesting client.

req pointer to the batch_request structure.

The function *que_purge()* is called to remove the queue. If the queue contains any jobs, the request is rejected.

mgr_queue_set()

```
void mgr_queue_set(int sfds, struct batch_request *req)
```

Args:

sfds the socket connection to the requesting client.

req pointer to the batch_request structure.

The queue is located by calling *find_queuebyname()*. Then *mgr_set_attr()* is called to update the queue attributes.

The routine *check_que_attr()* is called to insure the specified attributes are appropriate to the queue type; if there is a problem, a "warning" message is sent. An appropriate reply is returned to the client.

mgr_queue_unset()

```
void mgr_queue_unset(int sfds, struct batch_request *req)
```

Args:

sfds the socket connection to the requesting client.

req pointer to the batch_request structure.

The queue is located by calling *find_queuebyname()*. The specified attributes of the queue are unset (cleared) by calling *mgr_unset_attr()*. An appropriate reply is returned to the

client.

mgr_set_attr()

```
int mgr_set_attr(attribute *patr, attribute_def *padef, int numattr,
                 svrattrl *reqattr, int privilege, int *bad)
```

Args:

patr pointer to the attribute array in the server or queue to be set.

padefpointer to the attribute definition array for the objects attributes.

numattr
integer number of attributes in the parent object attribute array.

reqattr
pointer to the list of attributes in the batch request

privilege
level of the client.

bad RETURN: pointer to integer, if an error occurs, the integer is set to the index of the attribute in error.

Returns:

0 if successful.

>0 error number if not successful.

The setting of the requested attributes is treated as an atomic operation, all are set or none are. This is accomplished by calling *attr_atomic_set()* which duplicates the attribute values and updates the copies with the new values. If any error occurs, the copies are removed by calling *attr_atomic_kill()*.

For each and every modified attribute, the original parent object attribute is cleared and set to the temporary (new) value. If there is an *at_action()* routine associated with the attribute, it is invoked.

When all modification have been completed successfully, the temporary new attributes are removed. Note, the values are not freeded because the real attributes point to the values where malloc-ed storage is involved.

If an specified attribute is not found in the attribute definition array, if the attribute cannot be written with the client privilege, or the attribute is read-only, the integer pointed to by bad is set to the number, starting with 1, of the attributes ordinal position in the request list. An error value is returned.

mgr_unset_attr()

```
int mgr_unset_attr(attribute *patr, attribute_def *padef, int numattr,
                   svrattrl *plist, int privilege, int *bad)
```

Args:

patr pointer to the attribute array in the server or queue to be unset.

padefpointer to the attribute definition array for the objects attributes.

numattr
the integer number of the attributes in the definition array.

plist pointer to the list of svrattrl elements in the batch request.

privilege
level of the client.

bad RETURN: pointer to integer, if an error occurs, the integer is set to the index of the attribute in error.

Returns:

0 if successful.

>0 error number if not successful.

If an named attribute is not found in the attribute definition array or the attribute cannot be written with the client privilege, the integer pointed to by bad is set to the number, starting with 1, of the attributes ordinal position in the request list. An error value is returned.

If the attribute(s) specified in the request are not resources, the appropriate at_free() routine is called for each attribute of the parent object, queue or server, listed in the request. This also results in the flag {ATR_VFLAG_SET} being cleared.

If the attribute(s) are of type resource, {ATTR_TYPE_RESC}, and if a specific resource (member) is not specified, the attribute is freed as above. If however, a specific resource member is given, that member only is freed.

Kludge Warning

The server attribute "resources_cost", {SRV_ATR_resource_cost}, is set as a resource type attribute, i.e. the type field is set to {ATTR_TYPE_RESC}. This is because they relate to the different resource names. However, the structures in the value list are not resource structures, but are *resource_cost* structures. Therefore, when unsetting a single member of this attribute, the at_free() routine associated with *the resource* cannot be used; the value is just unlinked and freed. Rather then set up a new attribute type and have to check it where ever the server checks for ATTR_TYPE_RESC, in this one place in mgr_unset_attr(), we special case resource_cost structures by checking that the attribute parent type is {PARENT_TYPE_SERVER} as only the server has this type of resource and that the index into the attribute definition array is SRV_ATR_resource_cost.

mgr_node_set()

```
void mgr_node_set(struct batch_request *preq)
```

Args:

preq pointer to the batch request structure that holds the specific node request.

If the request is to apply to all nodes at the server, the local flag allnodes is set. Otherwise, the server's array of pbsnode structs is searched for the node specified in the request. If the node is not found in the server's array, the value [PBSE_UNKNODE] is sent back to the requestor.

If the node is found in the server's pbsnode array or the request applies to all nodes, the request is logged with the server and function *mgr_set_node_attr()* is called for each node in the request in an attempt to satisfy it. Assuming the entire request was able to be satisfied, *reply_ack* is called to send back the simple acknowledgement message and function *write_node_state* is called if any changed node-state information needs to be permanently recorded by the server. A return from function follows.

If for some reason the node modification request could not be satisfied, *mgr_set_node_attr* returns with a nonzero return code. The specific return code indicates the type of error encountered.

For the case where an error has occurred and the modifications were intended for a specific node an appropriate reply message is generated and returned to the requestor, along with the error code, either by calling *req_reject* or *reply_badattr*. In the latter case the variable bad will contain the node-attribute (its list position) that created a problem for the request. A return from function follows.

For the case where an error has occurred and the modifications are intended for all nodes, a pointer to the failed node is recorded in an array and processing advances to the next of the server's nodes. After processing all nodes, the array of failed nodes is scanned to construct a reply message listing those nodes that failed to get modified. This generated message is passed to the function *reply_text()*. Following this, memory malloc'd for the temporary recording of failures and message building is freed and, a return from function occurs.

mgr_node_delete()

```
void mgr_node_delete(struct batch_request *preq)
```

Args:

preq
: pointer to batch-request structure holding specific node request.

Top level function for deleting a node (or all nodes) in the server's node list. The pbsnode will be marked as deleted. It will no longer be assigned to any new jobs, will no longer be pinged in the server's main loop and, any current job tasks will continue executing on the node until they terminate or the job aborts, or the job is killed.

A check is made to determine if the node specification in the request is valid. If it is, the node is effectively deleted from the server's internal node list by calling *effective_node_delete*. At this point the function *chk_characteristic* is called to determine if the node is also marked as INUSE_OFFLINE. If it is so marked, an indicator is set to signal the fact that the file *node_status*, which tracks nodes that are offline, must be updated. Likewise, a global indicator, *svr_chngNodesfile*, is set to alert the server that the nodes file needs to be regenerated from the server's internal pbsnode list. Finally, the function *reply_ack* is called to send an acknowledgement of the request, an indication of success.

If the batch request cannot be successfully completed, an appropriate reply is sent back to the requester. During a global modification, a list of those nodes not being able to be modified is sent back as part of that reply.

mgr_node_create()

```
void mgr_node_create(struct batch_request *preq)
```

Args:

preq pointer to batch-request structure holding specific node request.

Top level function for creating a node for the server's internal node list. After the pbsnode is initialized, any properties or state or node type that has also been specified in the batch-request is set on the pbsnode by calling the function *mgr_set_node_attr*.

Assuming that all of this occurs successfully, a global indicator is set which will, at server shutdown, cause the *nodes* file to be regenerated based on the server's current internal pbsnodes list. And, the other thing which transpires on successful pbsnode creation is that each pbsnode which has not been "effectively deleted" from the server's list will have its INUSE_NEEDS_HELLO_PING bit set in the pbsnode's *inuse* field. This causes the *ping_nodes* function, called periodically in the server's main loop, to send a HELLO message to the MOM on the node being pinged. This message ultimately leads to the server sending to the MOM on the node in question all of the IP addresses for all the non-deleted nodes that it has in its list. The MOM can then update its internal set of *okclients,* those nodes from whom a communication is deemed valid. Finally, function *reply_ack* is called sending back to the requester an acknowledgement, all was successful.

If the pbsnode creation does not meet with success, the reason for the failure shows up in an error return code (rc) variable and that is processed to generate an appropriate reply to the requester. The possible error codes are PBSE_NONODES, PBSE_NODEEXIST, PBSE_SYSTEM, PBSE_INTERNAL, PBSE_NOATTR, PBSE_MUTUALEX, PBSE_BADNATVAL.

mgr_node_set_attr()

```
static int mgr_node_set_attr(struct pbsnode *pnode, attribute_def *pdef,
                             int limit, svrattrl *plist, int privil,
                             int *bad, void *parent, int mode)
```

Args:

- pnode
  pointer to pbsnode structure needing modification
- pdef beginning of the definitions array for node attributes
- limit
  length of the node-attribute definitions array
- plist
  pointer to the batch request's list of svrattrl structures
- privil
  requester's privilege level
- bad for a "bad node-attribute" type of error, pass back the offending attribute's position in the request list
- parent
  may go unused in this function
- modepassed to attrib's action func, not currently used by this func

Returns:

- 0 if successful in doing all modifications
- >0 return code if a problem occurs (modifications are rolled back)

This function is called by the top level function *mgr_node_set*. A successful (0) return means all necessary modifications to various data fields belonging to the specified pbsnode have gotten appropriately modified. If a problem occurs in mid-process, any partially completed modifications are abandoned, allocated memory is freed, an error return code indicating the source of the problem is passed back to the caller and, no modification is made to the subject pbsnode.

Processing occurs as follows:
Space for a temporary array of node-attribute structures is acquired on the heap. The num-

ber of node-attribute structures requested is the number that reside in the definitions file, *server/attr_node_def.c*. For each attribute in the array, that attribute's "action" function in the definition is called to give an initialization to the node-attribute. Any error that might occur mid way through halts the process, with function *attr_atomic_kill* being called to facilitate the roll back of the processing that has occurred to this point and an error code is passed back to the caller.

Once the temporary node-attribute array is setup, it is passed to function *attr_atomic_node_set* along with the list of requested node-attribute changes specified in the batch request received by the server. Attr_atomic_node_set calls upon the "decode" and "set" functions for each node-attribute specified in the request. Assuming this process is successful for the entire request, the temporary node-attribute array will have been updated appropriately and those node-attributes of the array that received an update have been marked. Should any problem occur mid way through the process, function *attr_atomic_kill* is called upon to roll back the processing and a non-zero error return code is passed back to the caller, for use in shaping a reply.

Give processing success to this point, a temporary copy (tnode) of the pbsnode is now updated using the data from the node-attributes in the temporary array. With success at this step, the temporary pbsnode gets copied back to the original pbsnode, the temporary node-attribute array is freed and success (0) is returned. Any failure during update of the temporary node gets handled as before.

mgr_log_attr()

```
static void mgr_log_attr(svrattrl *list, int log_class, char *object_name)
```

Args:

list of svrattrl structures containing the attributes from the request.

log_class
: an object class defined in log.h.

object_name
: the name of the parent object, queue or server.

For each attribute modified by a Manager Request, a log entry is formatted as:

```
attributes set: attribute_name =|+|- value
```

and written to the log file.

set_queue_type()

```
int set_queue_type(attribute *pattr, void *pque, int mode)
```

Args:

pattr pointer to the QA_ATR_QType attribute.

pque pointer to the queue being created or modified.

(unused).

This is the *at_action()* routine for the queue type attribute. The new string value of the QA_ATR_QType attribute is checked against the allowable values: Execution and Route. The match is made regardless of case and the string may be shorted to any set of initial charac-

ters. The attribute value string is replaced with a copy of the full string for consistency in the status display.

The internal type representation, qu_type, is also set.

check_que_enable()

```
int check_que_enable(attribute *pattr, void *pque, mode)
```

Args:

pattr pointer to the queue Enabled attribute.

pque pointer to the queue being enabled.

mode(unused).

Returns:

0 if queue completely defined.

Non-zero
error if queue has not be defined sufficiently to determine its type.

This function is the *at_action()* function associated with the queue QA_ATR_Enabled attribute. It is called whenever the attribute is modified. If the queue type has not yet been set, the enable is disallowed and [PBSE_QUENOEN] is returned.

check_que_attr()

```
static char *check_que_attr(queue *pque)
```

Args:

pque pointer to the queue being modified or created.

Returns:

Null if no conflict is found.

pointer
to the name of the attribute if one conflicts with the queue type.

This strangeness requires some explanation. A queue can either of two types: execution, or route. Some queue attributes are common to both types, others are specific to a single type. Rather than have two attribute definition arrays, one is defined with all possible queue attributes included. The tentative type of the queue is defined by "usage," the first type specific attribute specified determines the tentative queue type. Once that has happened, no attribute is allowed that is specific to a different type. Thus the existence of this routine.

For each attribute set in the queue, it is determined if the attribute is appropriate to the queue type. If the queue type has not yet been fixed, it is tentatively (internal to this routine) set to {QTYPE_Unset}, then any attribute is allowed, but the first attribute associated with only one type of queue forces the queue to that type (internal to this routine).

If the attribute does not fit with the real or tentative queue type, a pointer is the name of the attribute is returned.

manager_oper_chk()

```
int manager_oper_chk(attribute *pattr, void *pobject,int actmode)
```

Args:

pattr pointer to server managers or operators attribute.

pobject
pointer to parent object, the server.

actmode
the the type of action affecting the attribute.

Returns:

0 if no error

PBS error
number, if error.

This is the *at_action*() routine for two server attributes, managers, and operators. When the list of those with manager or operator privilege is set, altered, or otherwise modified, this routine is invoked. The routine validates each list entry to insure that the entry is in the form `user@fully.qualified.hostname` or `user@*.wildcard.domain`. This is done to insure that the list is not created with an invalid host name or a name that might be resolved in a different domain than was intended.

The user name must be followed by an '@' sign. If the string after the '@' does not start with a wild card character, '*', the string is used to obtain a fully qualified host name by calling *get_fullhostname()*. It is an error if the returned host name does not match the specified name. If the string does start with '*', no additional checks are possible. On any error on a set or alter actions (resulting from a batch request), [PBSE_BADHOST] is returned. On a recovery action (server initialization), any improper lines are logged, but no error is returned. An error might occur here if the access files were edited by hand.

5.3.11.7. node_func.c

The file *src/server/node_func.c* contains certain functions which are used in support batch requests pertaining to nodes.

find_nodebyname()

```
struct pbsnode    *find_nodebyname (char *nodename)
```

Args:

nodename pointer to the name of the node being sought

Returns:

0 if node name isn't found in the server's node list or the server doesn't have a list of nodes

address pointer to the pbsnode

This function walks the server's node list pbsnlist and returns the address of the pbsnode structure whose name field, last, matches the name pointed to by nodename. Zero is returned for the value of the pointer if no match is found or the list is empty.

save_characteristic()

```
void    save_characteristic(struct pbsnode *pnode)
```

Args:

pnode
: pointer to the pbsnode structure in question

Saves the *characteristics* of the pbsnode along with the address of the pbsnode. These are saved to static variables in the file and are later examined by the function, *chk_characteristic,* whose job it is to report back any changes in the node's characteristics to the caller.

chk_characteristic()

```
int    chk_characteristic(struct pbsnode *pnode, int *need_todo)
```

Args:

pnode
: pointer to the pbsnode structure in question

need_todo
: return various bit flags into this location

Returns:

-1 current pbsnode address doesn't match that stored by *save_characteristic.*

0 check was performed successfully and flag bits in *need_todo* got appropriately set/cleared

This function is the companion to function *save_characteristic(),* which should be invoked prior to the invocation of the current function. If the function is successful, the integer location pointed to by *need_todo* will hold the result of the check. Currently, the results of the check are encoded in a pair of bits in this integer location and are used in determining whether or not file *nodes* should be updated from the internal pbsnode array, and whether the file tracking nodes that are marked as being offline needs an update.

status_nodeattrib()

```
int  status_nodeattrib (svrattrl *pal, attribute_def *padef, struct pbsnode *pnode,
                        int limit, int priv, list_head *phead, int *bad)
```

Args:

pal pointer to an svrattrl structure from the batch request

padef pointer to the array of node-attribute definitions

pnode pointer to the subject pbsnode

limit number of elements in the array pointed to by padef

priv requester's privileges

phead heads a list of svrattrl structs in the reply area of the batch request structure

bad if there is a node-attribute error in processing, record it's list position here

Returns:

0 all requested status information was successfully obtained

!=0 some kind of error occured; if it's a node-attribute error *bad returns the position; an appropriate PBS error code is returned to the caller to shape the reply

This function is invoked when a batch status request regarding a node(s) is received by the server. It adds the status of each requested (or all) node-attribute to the status reply.

initialize_pbsnode()

void initialize_pbsnode (struct pbsnode *pnode, struct prop *pname, ulong *pul, int ntype)

Args:

pnode
pointer to pbsnode being initialized

pname
pointer to a prop struct carring the node's name

pul pointer to an array of unsigned ints. Each entry holds an ipaddrs for this node

ntype
flag indicating whether to set the node to time-shared or cluster

This function is invoked to carry out initialization on any new pbsnode being created via the *qmgr* command. The assumption is that all the input parameters are valid. This initialization parallels that done in function *setup_nodes* where the server reads the file *nodes* as part of its startup process.

effective_node_delete()

void effective_node_delete (struct pbsnode *pnode)

Args:

pnode pointer to pbsnode being effectively deleted

The pbsnode pointed to by *pnode* is effectively deleted from the server's internal pbsnodes list. This is accomplished by setting the INUSE_DELETED bit on the inuse field, removing the prop list that hangs from the pbsnode (including the name prop) and, clearing any INUSE_NEEDS_HELLO_PING bit that might be set in the pbsnode's inuse filed. Depending on the node's *ntype* field, the server's count of time-shared nodes or its count of cluster nodes is decremented by one.

setup_notification()

void setup_notification()

This function is invoked to Set up the mechanism for notifying the other members of the server's node pool that a new node was added manually via qmgr. Actual notification really occurs some time later via the server's invocation of the ping_nodes routine from within the server's main loop. For each node that does not have its INUSE_DELETED bit set in the inuse field, the INUSE_NEEDS_HELLO_PING bit is set. Setting of the bit causes the server to send a *Hello Ping* message to the node during the server's later invocation of the ping_nodes function. The node responds with a HELLO and the server then builds and sends to the node a list of all the IP addresses of all the non-deleted nodes that it has in its list. This message is read by the MOM on the node being pinged and the new IP address-set gets used to update the tree of *okclients* for the MOM on that node.

process_host_name_part()

int process_host_name_part (struct batch_request *preq, ulong **pul,
struct prop **pname, int *ntype)

Args:

preq pointer to a batch request (INPUT)

pul receives location of null terminated array with node's ip addresses (OUTPUT)

pname
receives location of a struct prop with name field that of the node in the batch_request (OUTPUT)

ntype
address of an integer location. Records into this integer whether the node is to be of type time-shared or of type cluster (OUTPUT)

Returns:

0 Success

!=0 An error code (PBSE_UNKNODE,PBSE_SYSTEM)

When invoked this function does the following, processes into a prop structure the hostname portion of a batch request involving a node, gets that host's set of IP addresses into an array and, places a code for the node's specified node-type (cluster/time-shared) into an integer variable. If the object name contained in the batch request is not null and, that name is a valid host name, a prop structure is allocated on the heap to hold the name. The IP addresses for the node are obtained from the system and written to a null terminated array of ints allocated on the heap. The location of these data structures are passed back via the calling parameters as is an indication of whether the request is for a node of type time-shared or cluster.

update_nodes_file()

```
int update_nodes_file()
```

When called, this function will attempt to update the *nodes* file of the server. It walks the server's array of pbsnodes constructing for each entry, which is not marked as deleted, a line for a new nodes file. The lines are written to a temporary file which subsequently, after all node processing is done, replaces the current nodes file. If any system errors happen along the way, the temporary file, if it exists, is closed and removed and the original nodes file is

not modified.

This function gets called by various primary functions in the *req_manager.c* file whenever a node is created/deleted or its properties/ntype modified. Should for some reason the function return error, a global indicator *svr_chngNodesfile* is set signaling that this function ought to be call during the server's shutdown process.

recompute_ntype_cnts()

```
void recompute_ntype_cnts()
```

The action of this function is to walk the server's array of pbsnodes and for each entry that is not marked as deleted notes its ntype value and increments one of the appropriate local counters (time-shared or cluster).

The server's global node counters, *svr_clnodes* and *svr_tsnodes,* are then replaced by the values from these local counters.

5.3.11.8. req_messagejob.c

The file *src/server/req_messagejob.c* contains the server function for processing the Message Job batch request.

req_messagejob()

```
void req_messagejob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

If the job is not in state {JOB_STATE_RUNNING} and substate {JOB_SUBSTATE_RUNNING} the request is rejected with [PBSE_BADSTATE].

The request is forwarded to the MOM responsible for the running job by calling *relay_to_mom()*. The action will be picked up in *post_message_req()* when MOM replies.

post_message_req()

```
static void post_message_req(struct work_task *task)
```

Args:

task pointer to the work task entry.

When MOM replies to a relayed Message Job Request, the delayed child work task entry points to this function. All it does is reply to the client with an acknowledge or reject based on the code in the reply from MOM.

5.3.11.9. req_modify.c

The file *src/server/req_modify.c* contains the server function for processing the Modify Job batch request.

req_modifyjob()

```
void req_modifyjob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

It is critical that the Modify Job request be atomic, either all of the attributes modifications are performed or none are. Therefore the function *attr_atomic_set()* is used to perform the set.

First, certain checks must be made first if the job is in the {JOB_STATE_RUNNING} state. If so, each specified attribute or resource is identified by calling *find_attr()*. If the attribute or resource is not marked as alterable in the run state, {ATR_DFLAG_ALTRUN} set, then the request is rejected with [PBSE_MODATRRUN]. Which resources are alterable when a job is running depends on MOM's ability to update the limit. Polled limits such as walltime can be updated on any host. On systems which use the *setrlimit()* system call, those system enforced limits are not updatable since they can only be set by the process which they control.

The routine *modify_job_attr()* is called to perform the set operation. If an error is detected, the attributes and resources are not updated and the error is returned to the user. The routine *set_resc_deflt()* is called to set to the default values any Resource_List values which may have been unset.

If the job is not currently running, *svr_evaljobstate()* and *svr_setjobstate()* are called to review and update the job state. Svr_setjobstate will also save the job structure and updated attributes to disk.

If a resource limit for a running job is being changed, *relay_to_mom()* is used to forward the request to MOM. When the reply is received, *post_modify_req()* is invoked.

modify_job_attr()

```
int modify_job_attr(job *pjob, svrattrl *list, int permission, int *bad)
```

Args:

pjob pointer to job whose attributes are to be modified.

list pointer to the first member of a list of svrattrl structures containing the new attributes values from the modify request.

permission
of the client from the request.

bad RETURN: pointer to an integer in which the index of the first bad attribute is returned.

Returns:

0 if ok.

non-zero
error number if error.

The function *attr_atomic_set()* is called to decode and set a copy of the job attributes. If the set is unsuccessful, the copies are freed and the error is returned.

If one or more resource limits are being changed, additional checks are made: If the job is running, only a manager or operator is allowed to raise them. The function *comp_resc()* is

used to compare the current and new values. If the job is not running, the limits may be adjusted up or down, but must remain with the queue minimum and maximum as established by QA_ATR_ResourceMax and QA_ATR_ResourceMin.

If there are no errors, each modified attribute value replaces the original. The original attribute value is freed and the new value inserted. It is important to note that the attribute copy value is not freed, it now belongs to the original. It is also should be noted that for those attributes whoses value is represented by linked list, the first and last list elements must be relinked, this is accomplished by calling *list_move()*.

If there is an *at_action()* routine associated with the attribute, it is invoked. If there are any failures, an error reply is returned. If either the User_List or group_list attributes changed, then *set_jobexid()* is called to determine the effective execution user and group names. This is done outside of any at_action routine because it involves two inter-dependent attributes.

Finally, the job modified flag, ji_modified, is set.

post_modify_req()

```
static void post_modify_req(struct work_task task)
```

Args:

task pointer to work task established by *relay_to_mom()*.

This function is invoked to process the return from MOM of a modify job request. The connection to MOM is closed ane the original request is reset to point back to the original client connection. If there was an error, it is logged and the error code returned to the client.

5.3.11.10. req_movejob.c

The file *src/server/req_movejob.c* contains the server function for processing the Move Job batch request.

req_movejob()

```
void req_movejob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

The job must be in one of the following states: {JOB_STATE_QUEUED}, {JOB_STATE_HELD}, or {JOB_STATE_WAITING}, otherwise the request is rejected.

If the destination is another queue on this server, the state of the destination queue and the authorization of the user to access that queue is checked. If the may be moved, the job is dequeued by calling *svr_dequejob()* and queue in the new destination by calling *svr_enquejob()*. A success reply is returned to the client. If the job cannot be moved, the request is rejected.

If the destination is on a different server, the destination specified in the request is saved in the job structure member *ji_destin* and *ji_un_type* is {JOB_UNION_TYPE_ROUTE}.

The function *create_child_entry()* is called to create a child process table entry of type {Child_ROUTER}. The batch request is linked into the list headed in the child table entry field cp_deferred. The job state and substate are set to {JOB_STATE_TRANSIT} and {JOB_SUBSTATE_TRNOUT}. A child process is created by calling *fork()*, the cp_pid field of the child process table entry is updated by the parent. The parent server returns to continue

processing other events and requests.

The child process calls the function *svr_routejob()* to perform the move operation. The child process will generate and create the various subrequests which are part of the Queue Job batch request. If any errors or network time outs occur, an error code is returned as the child process exit status.

req_orderjob()

```
void req_orderjob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

This function provides the batch service in Order Job batch request. This request is to swap the positions of two jobs in a queue. The requestor must have permission to operate on both jobs; be owner of both or be privileged. Neither job can be running, or [PBSE_BADSTATE] is returned.

If the two jobs are in the same queue, the problem is fairly simple. The list_link function, *swap_link* () is called twice, first to swap the position of the two jobs in the server's all job list and the second time to swap positions in the queue list. The JOB_ATR_qrank, "queue_rank", is also exchanged between to the two jobs so they will be correctly ordered if the server is restarted. Both jobs are saved to disk to record the queue rank change.

When the two jobs are in different queues, extra checks must be made to be sure that each job is allowed into the other's queue. The function *svr_chkque()* is called for both of the jobs with the opposite queue header. If the two jobs are allowed in the opposite queue, the rank in JOB_ATR_qrank, is swapped as above. The parent queue name in the job structure, *ji_queue*, is swapped and the two jobs are dequeued from their existing queue and requeued into the other. This insures that the current queue attribute and queue type are updated. A "Q" record is also produced for the accounting log.

5.3.11.11. req_register.c

The file *src/server/req_register.c* contains the functions to deal with the Register Dependent Job batch request as well as additional dependency related functions. This file and the similarly named function are mis-named since the register operation is only one of several dependency related operations. It is just too much bother to go back and change all the references. It is quickly determined by the reader that this set of functions is the strangest and most difficult to understand in all of PBS. A extra credit "A" is here by given to the reader that figures it all out.

req_register()

```
void req_register (struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

This function provides the batch service in response to a Register Dependent batch request. The request may ask for one of four operations.

Three of the operations are used for the non-synchronous dependencies and are fairly straight forward:

REGISTER

This operation registers a dependency relation between parent and child. It results from an after* dependency attribute on the child or a before* dependency attribute on the parent. Included in the request is the full type of dependency and the id of the registering job. When received, the server will set up a mirror image type dependency attribute. This will remind the server to send notification to the child job when the parent reaches the specified state.

Note, if the server is built with {PBS_DEPENDENCY_SECURE=1}, then any Register Dependent batch request must be from the owner of the job affected. This prohibits cross user dependencies. If the server is built with {PBS_DEPENDENCY_SECURE=0}, then Register Dependent batch requests which "register" the "after" types and corresponding "release" operations described below, are accepted even when the requesting user does not own the affected job. This allows cross-user dependencies, however with the check on the ready request described below, the only type of dependency that can be established by another user is one where that user's job runs *after* another users. This prevents a user from delaying or expediting another user's program execution.

RELEASE

This operation is sent from the parent to the child's server. It indicates that the specified after* dependency has been satisfied and can be removed from the dependency list. When all after* dependencies have been removed, the hold is removed from the child and it is free to run.

DELETE

The DELETE operation requests that a server abort a dependent job. It is sent to dependent jobs whoses dependency cannot be satisfied. For example, if Job-B is dependent on Job-A termination normally, exit status of zero, and Job-A terminates abnormally, then the server managing Job-A will send a DELETE operation to the server managing Job-B.

UNREGISTER

The UNREG operation is the reverse of the register. An existing relationship between the job and a child is to be removed. Either *unregister_sync()* or *unregister_dep()* is called depending on type dependency type.

If the dependency type is synchronous, the work is a bit more involved. There are three approached that could be taken here, which oh which ???

1. When the "master" job has received a register operation from all other jobs in the set, the server will send a release operation to each job. This will remove the system hold and allow the job to begin to compete for resources. When each job has its resources, it would notify the master; when all have their resources at the same time, a run request would be sent to each.
2. As each job registers, it sends the "cost" of its required resources. When all jobs have registered, the job with the highest resource cost is released from its hold. When that job is scheduled, the lower cost jobs are "forced" into running as well.
3. As each job registers, it sends the "cost" of its required resources. When all jobs have registered, the job with the lowest resource cost is released from its hold. When that job is scheduled and begins to run, it notifies the master. Then the job with the next lowest resource cost is released. This continues until all jobs are placed into execution.

Approach 1 seems too complex to work. Without a master scheduler, it is unlikely on loaded systems that all jobs would have resources available at the same time. To keep from choking the system, the jobs could not hold onto their slot forever, but would have to time out and release their run window. Approach 2 is a possibility but might lead to threshing when the

lower cost jobs are "forced" to run. But it might still work.

To start with, the PBS team chose to go with approach 3, even though it is the least synchronous of the approaches.

REGISTER

The register operation dependency request is sent to the server managing the "master" job. This establishes the link from the master back to the child and reports the cost of resources for each job. It is also used to update the location of the child/master when that job moves.

Requests must be from the job owner regardless of the setting of {PBS_DEPENDENCY_SECURE}.

RELEASE

When an Register operation has received for each expected dependent child, and when a Ready operation is received from a prior released job, the master server will send a release request to release the hold on the job with the cheapest resource cost which has not yet been released. This allows that job to fight for resources (be scheduled).

READY

When a child job is able to obtain its resources (has be scheduled), a *Ready* operation is sent to the master. When the master scheduler receives a Ready operation from a child, as described above, it will release the next cheapest job until all have been released.

alter_unreg()

```
static void alter_unreg(job *pjob, attribute *old, attribute *new)
```

Args:

pjob pointer to job being altered.

old pointer to job's current (old) dependency attribute.

new pointer to job's new (as altered) dependency attribute.

For any dependency type currently established for the job which are being deleted (are not in the new [altered] attribute), an unregister, {JOB_DEPEND_OP_UNREG}, operation is send to the parner job. This deletes the corresponding dependency listed with that job. This routine is called by *depend_on_que()* when it is acting as the at_action() routine for the dependency attribute.

depend_on_que()

```
int depend_on_que(attribute *pattr, job *pjob, int mode)
```

Args:

pattr pointer to the dependency attribute.

pjob pointer to a job structure.

modeis the at_action mode.

Returns:

Zero if ok, non-zero otherwise.

The function is called on two events, when a job is moved into an execution queue, the mode will be {ATR_ACTION_NOOP},
and when the dependency attribute is altered, the mode is {ATR_ACTION_ALTER}. The alter case happens when this routine is called as the *at_action* routine for the dependency attribute. In either case, we want the actions to only happen if the job is in an execution queue so jobs are not held in routing queues. The other time this routine is called is when a job is moved into an execution queue, so it is called from *svr_enquejob()*.

For the alter case only, existing dependencies could be deleted, so *alter_unreg()* is called to check for that possibility.

If the job has dependencies which required placing a system hold on the job, that is done by calling *set_depend_hold()*.

If the job has SYNCCT dependency, the (master) job's resource cost is calculated by *calc_job_cost()* and a entry in the syncct list is created (as if a Register operation request had been received) by calling *register_sync()*. If all jobs have registered (unlikely in this case as this is the master job), *release_cheapest()* is called to send a release to the cheapest job.

For all other dependency types except JOB_DEPEND_TYPE_ON (all Before and After types), a Register Dependency — Register operation is sent to the parent job (job on which the dependency is based).

post_doq()

```
static void post_doq(struct work_task *pwt)
```

Args:

pwt pointer to work task entry created by issue_request().

This routine is the call back routine when the reply to a Register dependency request is sent from within depend_on_que(). If the request was rejected, then the job for which the request was sent is aborted.

depend_on_exec()

```
void depend_on_exec(job *pjob)
```

Args:

pjob pointer to a job structure.

This routine is called when a job with dependencies goes into execution. If the job has BEFORESTART dependencies, a Register Dependencies — Release message is sent to each job in the set. If the job is a member of a sync set and not the master (has dependency of SYNCWITH), a Register Dependencies — Ready message is sent to the master stating that this job is about to run. If the job is the master of a sync set, (has dependency of SYNCCT), then *release_cheapest()* is called directly to release the next cheapest job.

post_doe()

```
static void post_doe(struct work_task *pwt)
```

Args:

pwt pointer to work task entry created by issue_request().

This routine is the call back routine when the reply to a Register dependency request is sent from within depend_on_exec(). If the request was rejected, then the job for which the request was sent is aborted.

depend_on_term()

```
void depend_on_term(job *pjob)
```

Args:

pjob pointer to a job structure.

This routine is called when a job with dependencies terminates execution. If the job has BEFOREANY dependencies, a Register Dependencies — Release message is sent. If the job has BEFOREOK dependencies and the job terminated "normally", and/or BEFORENOTOK dependencies and the job terminated abnormally, a Register Dependencies — Release message is sent to each job. Otherwise, a Register Dependencies — Delete message is sent to those jobs that will never run because of the dependency on the reverse exit status.

If the job has JOB_DEPEND_TYPE_SYNCCT a special check must be performed. The whole purpose behind the sync set concept is to have jobs run at the same time and communicate with each other. If there is no communication, there is no need to run together. So if a job, especially the master, quits before all jobs have started running, then there must be a problem. Doubly so for the master, not because of any relation internal to the jobs, but because with out it there is no place to register the Release and Ready operations. Therefore, if the master has terminated and not all of the jobs in the sync set have reported Ready (running), then all jobs are aborted.

release_cheapest()

```
static void release_cheapest( job *pjob, struct depend *pdep)
```

Args:

pjob pointer to job for which the resource cost should be calculated.

pdep pointer to the SYNCCT dependency.

For each job in the set (list) headed by the SYNCCT dependency which have not been Released or Readied (running), find the one with the lowest resource cost. If this is the first job of the set to be released, then set the scheduler hint field to {SYNC_SCHED_HINT_FIRST}, otherwise, set it to {SYNC_SCHED_HINT_OTHER}. Call *send_depend_req()* to send a Register Dependency — Release operation message.

The scheduler hint field is recorded in the receiving job's Sched_hint attribute. As explained in the ERS, this is purely a hint to the scheduler to decrease the priority of the first job to prevent cheating and increase priority of the other jobs in the set to improve synchronism.

set_depend_hold()

```
static void set_depend_hold(job *pjob, attribute *depend)
```

Args:

pjob pointer to job structure

depend
pointer to the dependency attribute.

This function examines the dependencies on a job and if required, sets a system hold. Depending on the dependency type, the job state is set to {JOB_STATE_HELD} and the substate to either:

JOB_SUBSTATE_SYNCHOLD
If the job has either {JOB_DEPEND_TYPE_SYNCWITH} or {JOB_DEPEND_TYPE_SYNCCT} dependencies that have not been released.

JOB_SUBSTATE_DEPNHOLD
If the job has any {JOB_DEPEND_TYPE_AFTER*} or {JOB_DEPEND_TYPE_ON} type dependencies.

If the job has none of the above dependencies and was in substate {JOB_DEPEND_TYPE_SYNCWITH} or {JOB_DEPEND_TYPE_SYNCCT}, the system hold is removed and the state is re-evaluated by calling *svr_evaljobstate()*.

depend_clrrdy()

```
void depend_clrrdy(job *pjob)
```

Args:

pjob pointer to a job structure.

This function clears any synchronous dependency ready flags in the job's dependency attribute. It is called from pbsd_init() during recover. The flags are cleared because it is unlikely that the children are still ready. At some point in the future, the children will again notify the parent that they are ready.

find_depend()

```
static depend *find_depend(int type, attribute *pattr)
```

Args:

type of the dependency to find.

pattr pointer to the dependency attribute of the job.

Returns:

pointer
to the depend structure found of the requested type.

This function searchs the dependency attribute of a job for a certain dependency type, a depend structure of the specified type. If it is found, a pointer to it is returned.

make_depend()

```
static depend *make_depend(int type, attribute *pattr)
```

Args:

type of the dependency to be added.

pattr pointer to the dependency attribute of the job.

Returns:

pointer
to the created depend structure.

This function allocates and initializes a depend structure and links it on the list of structures headed in the dependency attribute.

register_sync()

```
static int register_sync(struct depend *depend, char *child, char *host,
                         long cost)
```

Args:

depend
pointer to the {JOB_DEPEND_TYPE_SYNCCT} dependency structure.

child the job id of the child (non-master) job.

host the name of the server (host name) which manages the child job.

cost the resource cost for the child job.

Returns:

0 if successful

error [PBSE_SYSTEM] if failed.

This function is called when a Register Dependency request is received with an operation of {JOB_DEPEND_OP_REGISTER} and the dependency type is {JOB_DEPEND_TYPE_SYNCWITH}. If the client job has already been registered with this, the "master" job, the location of client job is updated. Otherwise, the client (child) job is registered by making adding a **depend_job** structure, see *make_dependjob()*, to the "syncwith" depend structure. The child job's resource cost is recorded and the count of registered job is incremented in dp_numreg. If dp_numreg exceeds the number of expected jobs, dp_numexp, [PBSE_IVALREQ] is returned.

register_after()

```
static int register_dep(attribute *pattr, struct batch_request *request,
                        int type, int *made)
```

Args:

pattr pointer to the dependency attribute of a job.

request
pointer to the Register Dependency batch request.

type of the dependency to set up.

makeRETURN: pointer to integer which is set to 1 if the child dependency is new (was made), 0 if already exists.

Returns:

0 on success.

error number if fails.

This function is called from *req_register()* when a register request is received with the operation of {JOB_DEPEND_OP_REGISTER} and a type of any of the {JOB_DEPEND_TYPE_AFTER*} or {JOB_DEPEND_TYPE_BEFORE*} forms. The purpose is to set up or update a dependency of the opposite form (before_X becomes after_X, after_Y becomes before_Y) to remind the server to release the depend job at the right time. First, find or make a depend structure of the type needed, opposite of that in the request. Then add or update the location of the dependent child job. One is returned in the argument pointed to by made if the dependency is created, zero is returned if just updated.

unregister_dep()

```
static int unregister_dep(attribute *pattr, struct batch_request *preq)
```

Args:

pattr pointer to the job's dependency attribute.

preq pointer to the batch request (dependency register, op of unregister).

Returns:

zero on success, [PBSE_IVALREQ] if the dependency to unregister is not present.

This handles unregistering (deleting) before/after dependencies. The mirror image type dependency (before* <-> after*) pointing to the requesting job is located. It is deleted by calling *del_depend_job()*.

unregister_sync()

```
static int unregister_sync(attribute *pattr, struct batch_request *preq)
```

Args:

pattr pointer to the job's dependency attribute.

preq pointer to the batch request (dependency register, op of unregister).

Returns:

zero on success, [PBSE_IVALREQ] if the dependency to unregister is not present.

This handles unregistering (deleting) syncwith dependencies. The master, {JOB_DEPEND_TYPE_SYNCCT} dependency is located and within it the registration pointing to the requesting job. It is deleted by calling *del_depend_job()*. The number of registered jobs is decremented. Assuming that drops the count below what is required to release the first job, if the master job has been released, is re-held.

find_dependjob()

```
static struct depend_job *find_dependjob(struct depend *depend, char *jobid)
```

Args:

depend
: pointer to the depend structure.

jobid of the job of which the depend_job structure is desired.

Returns:

pointer
: to the depend_job structure if found.

The list of depend_job structures attached to the depend structure is searched for one with the child id matching the supplied job id.

make_dependjob()

```
static struct depend_job *make_dependjob(struct depend *depend,
                                         char *jobid, char *host)
```

Args:

depend
: pointer to the parent depend structure.

jobid of the job to add.

host name (server name) owning the child job.

Returns:

pointer
: to the created depend_job structure.

A depend_job structure is allocated, initialized and appended to the list headed in the parent depend structure.

send_depend_req()

```
static int send_depend_req(job *pjob, struct depend_job *parent, int type,
                           int op, int scheduler_hint,
                           void postfunc(struct work_task *))
```

Args:

pjob pointer to the job which is to be registered with another.

parent
: pointer to the depend_job structure holding the parent job's name.

type the type of dependency of the child (to register).

op the operation, Register, Ready, Delete, ...

scheduler_hint
: the value of the scheduler hint to pass to the other job.

postfunc
: the function to call when the reply to the request is received.

Returns:

zero on success.

non-zero
: error value if an error occurred.

This function forms and issues a Register Dependent Job batch request. The owner of the job, not the server, is inserted into the request as the requester. The job id of the parent job is taken from the job dependency structure pointed to by parent. The dependency type and operation code is set according to the arguments type and op. The destination server is obtained from the job dependency structure.

If the request is of type {JOB_DEPEND_TYPE_SYNCWITH} and the operation is {JOB_DEPEND_OP_REGISTER}, then the child job's resource cost is calculated, *calc_job_cost()* and included in the request. Otherwise it is set to 0.

The function *issue_to_svr()* is called to send the request on its way, with postfunc as the call back routine.

decode_depend()

```
int decode_depend(attribute *pattr, char *name, char *rescn, char *val)
```

The value string passed in parameter *val* is a null string or a comma-separated series of substrings. Each substring is of the form:

```
depend_type=argument[,argument,...][,depend_type=argument[,argument,...]...]
```

where *depend_type* is one of the following:

```
on    after       before      syncwith
      afterok     beforeok    syncct
      afternotok  beforenotok
      afterany    beforeany
```

as described in the ERS. The *argument* portion of the substring depends on the depend_type. For on or syncct, the *argument* is a numeric string which is a count of jobs. Otherwise, *argument* is a job identifier.

If the *val* is the null string, the attribute is being "unset". The attribute is freed by calling *free_depend()* and marked with {ATR_VFLAG_MODIFY} (free_depend() cleared

Otherwise, for each *depend_type* specified in the value string:

1. If a *depend* structure of that form does not already exist, one is created and linked into the list headed in the attribute structure. This structure identifies the base dependency type and the number of jobs listed for this type.
2. An explicit "0n" form will have set its count in the number of expected jobs in the "on" structure. The "on" structure is created if non-existent.
3. For each "after", "before", or "sync" form, a *depend_job* structure is created containing the job identifier, the job location, and the registered/ready flags.

The attribute flags are set with {ATR_VFLAG_MODIFY} and {ATR_VFLAG_SET}.

cpy_jobsvr()

```
static int cpy_jobsvr(char *dest, char *source)
```

Args:

dest pointer to destination string.

sourcepointer to source string.

This little kludge is used in *encode_depend()* to copy a job id of the form `seq.server[:port][@server[:port]]` to `seq.server[:port][@server[:port]]`. It escapes the colons since the colon is also used to separate job ids within the dependency string.

dup_depend()

```
static int dup_depend(attribute *pattr, struct depend *depend)
```

Args:

pattr pointer to job's dependency attribute in which a dependency is to be duplicated.

depend
pointer to the dependency to be duplicated.

Returns:

zero on success, non-zero if error.

This function duplicate (adds) a dependency to attribute. A new dependency sub-structure is allocated by calling *make_depend()* with the attribute and the type of the dependency from the existing one. Various fields are copied into the new and for each child job, the depend_job structure is reproduced.

encode_depend()

```
int encode_depend(attribute *pattr, list_head *phead, char *atname,
                  char *rsname, int mode)
```

The values of the dependencies are encoded into a series of strings and placed into a buffer. The encoding performed is according to the following rules:

1. For each depend structure in the list, a *depend_type* string is placed in the buffer followed by an equal sign, "=", followed by the appropriate argument string.
2. If the the depend_type is `syncct` or `on`, the argument string is a numeric string expressing the dependency count. Otherwise the string is a colon separated list of the job identifiers associated with the depend_type. Colons within each job identifier, used to indicate an alternative server port, must be escaped with a leading back slash.

Then the function *attrlist_create()* is called to create an svrattrl entry containing the attribute name and the encode string is inserted into the entry.

set_depend()

```
int set_depend(attribute *old, attribute *new, enum set_op op)
```

The value of the depend attribute old is set according to the operation:

Set old is replaced with new.

Incr [Not currently supported.]

Decr [Not currently supported.]

If the type of dependency to be "set" already exists in the "old" attribure, it is deleted via *del_depend()*. The dependency from the "new" attribute is copied via *dup_depend()*.

comp_depend()

```
int comp_depend(attribute *pattr, attribute *with)
```

Not used, does nothing, always returns -1.

free_depend()

```
void free_depend(attribute *pattr)
```

The depend attribute lists are freed and in the attribute flag {ATR_VFLAG_SET} is cleared.

build_depend()

```
static int build_depend(list_head *head, char *key, char *value)
```

Args:

head of list of depend structures.

key is keyword, name of dependency type.

valueis the value string, following the equal sign.

Returns:

0 if ok,

non-0if error.

The keyword is used to determine the dependency type. If it does not match a legal value, [PBSE_BADATVAL] is returned.

Since certain combinations of dependencies are illegal, the existing dependencies are scanned and the types noted. If the new dependency would create an illegal combination, [PBSE_BADATVAL] is returned. The illegal combinations are:

- syncwith with syncct, on, or any of the after forms.
- syncct with syncwith or another syncct.

If the base depend structure does not already exist for the type of dependency being created, one of the correct type is allocated. The value string is parsed by calling *parse_comma_string()* and an appropriate depend_job structure is allocated. Note that within a job id, a colon indicating an alternative server port must be escaped with a leading back slash in the external form. Otherwise, it would be taken as the colon that separates multiple job ids. Within the *depend_job* structure, the back slash is not needed and in fact gets in the way of comparing job ids. So the back slash is removed.

Note that a command line value of `depend=type` without a colon and following value is a means of clearing that type of dependency. build_depend makes an "empty" depend structure for that type. If the job is being altered, *set_depend()* will replace the existing entries for that type of dependency with the new (non-existent) ones, in effect, clearing the old entries.

clear_depend()

```
static void clear_depend(struct depend *pd, int type, int exist)
```

Args:

pd pointer to depend structure to clear.

type of depend structure to set.

exist flag, if true the depend structure already exists and any associated depend_job structures should be freed.

The depend structure is cleared.

del_depend()

```
void del_depend(struct depend *pd)
```

Args:

pd pointer to depend structure to delete.

A depend structure and any associated depend_job structures are freeded.

5.3.11.12. req_rescq.c

The file *src/server/req_rescq.c* deals with batch requests to query {PBS_BATCH_Rescq}, reserve {PBS_BATCH_ReserveResc}, and release {PBS_BATCH_ReleaseResc} resources.

req_rescq()

```
void req_rescq(struct batch_request *preq)
```

Args:

preq pointer to the {PBS_BATCH_Rescq} Query Resource batch request.

If the number of resource items, strings in the resource array, is less than one, the request is reject with [RM_ERR_BADPARAM]. The 4 integer arrays (available, allocated, reserved, down) to hold the returns are malloced and initialized to zero.

Each string in the resource list is parsed for the resource name and value. Depending on the resource name, the appropriate function is called. At the present time, only nodes is supported and the supporting function is *node_avail()*. If an unrecognized type of resource is specified, the request is rejected with [RM_ERR_BADPARAM].

req_rescreserve()

```
void req_rescreserve(struct batch_request *preq)
```

Args:

preq pointer to the {PBS_BATCH_ReserveResc} Reserve Resource batch request.

At the present time, the only *reserverable resources* handled via this request are nodes.

The client must have manager or operator privilege to make this request. If the number of resource items, strings in the resource array, is less than one, the request is reject with [RM_ERR_BADPARAM].

If the suppied resource handle is not null, {RESOURCE_T_NULL}, any existing resources allocated to that handle are released by calling *node_unreserve()*. Otherwise, a new resource handle is generated to be returned.

For each resource string in the array, the corresponding resource support function is called. For nodes the function is *node_reserve*().

If the reservation is only partially successful (some but not all nodes were reserved), [PBSE_RMPART] is returned. The resource handle is returned.

req_rescfree()

```
void req_rescfree(struct batch_request *preq)
```

Args:

preq pointer to the Reserve Resource batch request.

At the present time, the only *reserverable resources* handled via this request are nodes.

node_unreserve() is called to free (release or unreserve) the nodes.

5.3.11.13. req_rerun.c

The file *src/server/req_rerun.c* contains the server function for processing the Run Job batch request.

req_rerunjob()

```
void req_rerunjob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

The job structure is located. The job state must be {JOB_STATE_RUNNING} and the substate {JOB_SUBSTATE_RUNNING} or the request is rejected. The job rerunable attribute must be set to y or the request is rejected.

The job substate is set to {JOB_SUBSTATE_RERUN}. The function *send_signal()* is called to request that MOM send SIGKILL to the process group. The function *post_rerun()* will handle the reply from MOM about the signal request.

Latter, when MOM notifies the server of job termination, the post-execution processing routine, *req_jobobit()*, will note the rerun substate of JOB_SUBSTATE_RERUN, set {JOB_SVFLG_HASRUN} in the job server flags (ji_svrflags) and requeue the job. The flags {JOB_SVFLG_CHKPT} and {JOB_SVFLG_ChkptMig} are cleared to prevent the job from being set up for restart when next run, see *send_job*().

account_record() is called with {PBS_ACCT_RERUN} to note the rerun in the accounting file.

post_rerun()

```
static void post_rerun(struct work_task *pwt)
```

Args:

pwt pointer to a work task entry.

This routine processes the reply from MOM regarding the signal job request sent in req_rerun(). If MOM had no problem, the work task entry (and therefore the request structure) is released.

If MOM rejected the request, about the only valid reason would be that she did not know if the job id. Why this might happen I don't know, but it has once or twice. Anyway, the job is directly requeued.

5.3.11.14. req_runjob.c

The file *src/server/req_runjob.c* contains the server functions for processing the Run Job batch request and general placing a job into execution.

req_runjob()

```
void req_runjob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

This function handles the Run Job and Async Run Job requests. These requests requires the requesting user to have operator or administrator privilege, otherwise the request is rejected. The client may be another server performing a synchronous dependency job start, the Scheduler, or the qrun command. Servers/schedulers always have privilege. This is checked by calling *chk_job_torun()*.

If the request is the Async Run request, the request is acknowledged now to prevent any delays. The pointer to the request, preq, is nulled to prevent any later attempt to use it since the request structure is freed by the acknowledgement.

The function *svr_startjob()* is called to initiate the job into execution. If svr_startjob() returns a error, the Run Job request is rejected by req_runjob(). For a normal (non async) Run request, the request is acknowledged by one of the follow up routines, *svr_stagein()* or

post_sendmom().

req_stagein()

```
void req_stagein(struct batch_request *request)
```

Args:

request
pointer to the Stage In batch request.

This starts the file stage-in process. It is normally invoke by the scheduler. If the job does not have files to stage in, the request is rejected with [PBSE_IVALREQ].

svr_stagein() is called to send the copy file request to MOM. That function is requested to update the job to state {JOB_STATE_QUEUED} and substate {JOB_SUBSTATE_STAGEIN} during the stage in operation.

svr_stagein()

```
static int svr_stagein(job *pjob, struct batch_request *preq, int state,
int substate)
```

Args:

pjob pointer to the job.

preq pointer to the Run Job batch request.

state The next job state.

substate
The next job substate.

Returns:

0 if the Copy Files was successfully sent to Mom.

non-zero
error reply otherwise.

The function *cpy_stage()* is called to build a Copy Files batch request. A copy of the job id is created and a pointer to it is placed in the batch request, *rq_extra*. This string is used to find the job structure in *post_stagein()* rather than saving the value of pjob. It is remotely possible that the job might be deleted before Mom replies to the request, in which case, the pointer to the job would be invalid.

If the Copy Files request was built by cpy_stage(), then there are indeed files to copy. The Copy Files request is sent to Mom by calling *relay_to_mom()* with the host address saved in the job structure, ji_qs.ji_un.ji_exect.ji_momaddr. The job state and substate are set to *state* and *substate*.

At this point, a reply is sent to the original batch request, rather than wait the possibly long time it may take Mom to copy the requested files. This does mean that a failure of the copy will cause a asynchronous wait being placed on the job.

If the Copy Files request was not built by cpy_stage(), there were no files listed in the stage-in attribute. The routine *svr_strtjob2()* is called to start the job execution and its return is our return.

post_stagein()

```
static void post_stagein(struct work_task *task)
```

Args:

task pointer to work task structure.

This function is called when the reply to a Copy Files request to Mom, initiated in *svr_stagein()*, is received by the server. The job for which the request was issued is located by calling *find_job()* with the job id saved in the copy request, see svr_stagein(). If the job is not found (was deleted, unlikely but possible), the function just returns.

If the copy request return is zero, the next action is determined by the current substate of the job. If it is {JOB_SUBSTATE_STAGEGO}, *svr_strtjob2()* is called to send the job to Mom for execution. Note that the batch request pointer, the second parameter, is null. The original request has already be acknowledged in svr_stagein(). If the job substate is not JOB_SUBSTATE_STAGEGO, it is {JOB_SUBSTATE_STAGEIN} and the state and substate are updated by calling *svr_evaljobstate()* and *svr_setjobstate()*. The job is most likely to be placed in state {JOB_STATE_QUEUED} and substate {JOB_SUBSTATE_STAGECMP}.

If the return from MOM is non-zero, the copy failed and the job is placed in a waitting state, {JOB_STATE_WAITING}, substate {JOB_SUBSTATE_STAGEFAIL}. The execution time attribute, JOB_ATR_exectime, is set for {PBS_STAGEFAIL_WAIT} seconds in the future. This is done to keep the job from being rescheduled over and over in a short amount of time. A mail message is and sent to the job owner requesting that he/she investigate and fix the problem.

svr_startjob()

```
int svr_startjob(job *pj, struct batch_request *request)
```

Args:

pj Pointer to job structure of a job to run.

request
to run the job to which must be responded, or NULL if server staring jobs on initialization.

Returns:

0 If contact with MOM was successful (see below).

non-zero
if job could not be placed into execution.

This function attempts to place the job into running state. It is called when the Run Job Batch Request is received, this may be from the scheduler or the operator.

The short file name used as the base for saving the job structure and script must be made available to Mom, she will used the same name as we know there will not be a conflict with other jobs. To ship it to Mom, this name is placed in a read-only attribute, JOB_ATR_hashname.

If the job has the JOB_ATR_stagein attribute set, then *svr_stagein()* is called to direct Mom to copy the files. It is passed the state and substate of {JOB_STATE_RUNNING} and {JOB_SUBSTATE_STAGEGO} to indicate that the job will be run as soon as the files are staged-in. If svr_stagein() returns non-zero indicating it was unable to contact Mom, the Run Job request is rejected. If svr_stagein() is able to contact Mom, it will reply to the request (see the

commentary in svr_stagein).

If there are no files to stage in, *svr_strtjob2()* is called.

svr_strtjob2()

```
static int svr_strtjob2(job *pjob, struct batch_request *request)
```

Args:

pj Pointer to job structure of a job to run.

request
to run the job to which must be responded, or NULL if server staring jobs on initialization.

Returns:

0 If contact with MOM was successful (see below).

non-zero
if job could not be placed into execution.

The job state and substate are set to {JOB_STATE_RUNNING} and {JOB_SUBSTATE_PRERUN}. Then *send_job()*, see svr_movejob.c, is called to "move" the job to MOM. This creates a child process to send the job. When the child process completes, the routine *post_sendmom()* is given control to update the job substate to {JOB_SUBSTATE_RUNNING}, or to requeue the job depending on success or failure of the move. post_sendmom() will also repond as required to the Run Job batch request if it exists.

post_sendmom()

```
static void post_sendmom(struct work_task *task)
```

Args:

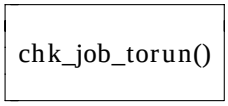
task pointer to work task entry which caused the dispatch of this function. In the work task, wt_parm1 points to the job, and if wt_parm2 is not NULL, it point to a Run Job batch request.

This function is equivalent to post_routejob for the case of sending a job to MOM for execution. When the child process exits, post_sendmom is dispatched as a result of the work task associated with the child.

If the send was successfully, the job is placed in state {JOB_STATE_RUNNING} and substate {JOB_SUBSTATE_RUNNING}. Note, for a very short job, there can be a race condition between the completion of the child process that sent the job to mom and the Obit notice from MOM, see req_jobobit(). It might be that the job substate has already been set to exiting. Also a rerun request could have changed the substate to indicate the rerun. Hence the state and substate is not updated if the substate is not {JOB_SUBSTATE_PRERUN} as set in *svr_strtjob2()*.

If there is an out standing Run Job batch request, pointed to wt_parm2, it is acknowledged. The time of the start is recorded in ji_chkpttime which is overloaded for this purpose for accounting. If the job is being restarted from a checkpoint file, *account_record()* is called with {PBS_ACCT_RESTRT}, otherwise *account_jobstr()* is called to make the accounting entry. The job Session Id attribute is updated by calling *stat_mom_job()*. If this job is the parent job of any dependent jobs waiting on this job to start, the dependent jobs are notified by calling *depend_on_exec()*.

If the send failed, and the substate is {JOB_SUBSTATE_ABORT} we assume the send was interrupted because the job is being deleted, and we do nothing except reject the batch request, if it exists. Otherwise, the job is requeued for a later retry; and if there is a batch request, it is rejected.




```
static job *chk_job_torun(struct batch_request *preq)
```

Args:

preq pointer to the batch request (run job or stagein).

Returns:

a pointer to the job specified in the request if all is well, otherwise null.

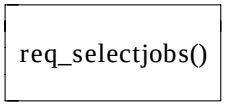
The job is located via *chk_job_request()*. The request will be rejected if the job is in {JOB_STATE_TRANSIT} or {JOB_STATE_EXITING} state, or substates {JOB_SUBSTATE_STAGEGO}, {JOB_SUBSTATE_PRERUN}, or {JOB_SUBSTATE_RUNNING}. If the request is to stage in files, it will also be rejected if the substate is {JOB_SUBSTATE_STAGEIN}.

The requesting client must have operator or administrator privilege (which the Scheduler does). The job must be in an execution queue.

A host for execution may be specified in the request. If this is the null string, either the local host is assumed or if the the job is to be restarted from a checkpoint, then the prior execution host is assumed as the new execution host. If the host name is not the null string and if the job is being restarted from a checkpoint, then the execution host must be the same as the earlier execution host or [PBSE_BADHOST] is returned. The name of the host on which to execute the job is saved in the job structure in ji_destin for svr_statjob(). This host name is also converted to a host address which is saved in ji_qs.ji_un.ji_exect.ji_momaddr. The attribute JOB_ATR_exec_host, execution host, is set to the selected/specified host name.

5.3.11.15. req_select.c

The file *src/server/req_select.c* contains the server function for processing the Select Job batch request and the Select−Status (selstat) Job batch request.




```
void req_selectjobs(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

This function handles both the Select Job and the special Select-Status Job request. The latter is provided primarily to enable the job Scheduler to obtain status about jobs that it should consider. It is a waste of bandwidth to receive status about jobs in routing queues or (depending on policy) held or waiting jobs in execution queues. There are two differences in the treatment of the requests, first the return values differ and second the sequence of processing. The Select Job request has as a return a list of job identifiers which meet the selection criteria. The Select-Status, or selstat, request has as its return a set of job status replies, one for each job which meets the selection criteria. The Select Job is straight foreward to process, just go through the list of jobs for those that match the selection criteria.

However, for Selstat, the same problem with running jobs exists as does for Status Job, the server's resources used information for some running jobs may be stale. A status request to MOM is required to update the server's information.

For both requests, each attribute specified in the request is decoded into a selection list which contains the decoded attribute value, the selection operator, and a pointer to the attribute definition, see *build_selist()*. For Select Job, the flow process to the final selection step in *sel_step3()*. For Selstat, two passes are required, the first in *sel_step2()* preselects the jobs and gets an status update from MOM for any that need it. Then *sel_step3()* re-selects the jobs for the reply. Information about the request are passed to both sel_step2() and sel_step3() in a *stat_cntl* structure as used by req_stat_job().

One of the specified attributes, {ATTR_q} or "destination" is not a true attribute and receives special treatment in build_selist() if present.... If {ATTR_q} was specified, then the search for jobs will be limited to the list headed by that queue. Otherwise, the search is among all jobs managed by the server.

sel_step2()

```
static void sel_step2(struct stat_cntl *cntl)
```

Args:

cntl pointer to a stat_cntl structure used to keep state for the search through the list of jobs.

The search starts with the job identified in the stat_cntl struture:

null The case for the first entry, start from the top of the list.

job name
The job on which broke the search the prior round (i.e. caused a stat request to MOM). If this job is missing, restart at the beginning.

With in the loop, the "next" job is obtained. This is either the first job if starting from the top, or the job following the one we left off with in the prior round.

For each job in the search list which the user is entitled to query status, the function *select_job()* is called to determine if the job meets the selection criteria, see sel_step3. For each "selected" job, if the job is running and the information from MOM is stale, older than {PBS_RESTAT_JOB} seconds as recorded in *ji_momstat*, the loop is broken to send a request to mom to update all jobs, see *stat_to_mom()*. The current job id is saved in the stat_cntl structure. When MOM replies, *sel_step2* will be restarted with the next job. When all jobs have been checked, *sel_step3()* is called to repeat the selection and build the status replies.

```
static void sel_step3(struct stat_cntl *cntl)
```

Args:

cntl pointer to a stat_cntl structure used to keep state for the search through the list of jobs.

Here is where the reply to the request is actually built. The stat_cntl structure is passed in from either *req_selectjobs()* for a simple Select Jobs request or from *sel_step2()* for a Select-status request. We loop through the list of jobs in the queue or server looking (again) for those that meet the criteria, *select_job()* is called for those jobs which the user is privileged to see. If the requesting user does not have special privilege, the ability to query jobs owned by other uses is determined by the setting of the server attribute query_other_jobs. For the batch request type of:

Select If a job meets the criteria, its job id is entered into the select reply list.

Selstat If a job meets the criteria, *status_job()* is called to append the status of that job to the reply.

After the search is complete, any space allocated to the selection list is freed. The select reply list, or job status list is included in the reply to the client.

select_job()

```
static int select_job(job *pj, struct select_list *psel)
```

Args:

pj pointer to a candidate job.

psel pointer to the selection list set up from the request.

Returns:

0 if job does not meet criteria.

1 if job does meet criteria.

For each attribute in the list pointed to by psel which has a value that has been set, the attribute's *at_comp()* function is called to determine the relationship between the requested attribute and the job attribute. If the relation matches that specified in the corresponding member of the operator array, the comparison continues. Otherwise the job is not selected, zero is returned. If all attributes match, then the job is selected. One is returned.

There is one attribute which must be special cased. If -u, is specified, it is a list of job owners, not the user-list job attribute. If the job owner is in the list, we accept the job.

sel_attr()

```
static int sel_attr(attribute *pattr, struct select_list *select)
```

Args:

pattr pointer to an attribute.

selectpointer to a select_list entry.

Returns:

1 if attribute matches the selection criteria.

0 if not.

The attribute value and the value in the selection list are compared via a call to the appropriate *at_comp()* routine. The comparison result is matched against the selection operator. If it fits 1 is returned, otherwise 0 is returned.

build_selentry()

```
static int build_selentry(svrattrl *plist, attribute_def *pdef, int perm,
                          struct select_list **rtnentry);
```

Args:

plist pointer to a member of the list of attributes in request on which to select.

pdef pointer to the attribute definition for the above attribute.

perm the users access permissions.

rtnentry
RETURN: the address of the created entry is returned here.

Returns:

zero if ok, error code if not.

A single selection list entry is created for the specified attribute. The entry contains a pointer to the attribute definition structure. This provides access to the comparison routine (at_comp). It also contains the decode attribute value from the request and the selection operator.

If the privilege level is not sufficient to read the attribute, or the attribute cannot be selected in the manner requested [some attributes are restricted to an equal/not equal test], an error is returned.

free_sellist()

```
static void free_sellist(struct select_list *pslist)
```

Args:

pslistpointer to a select list structure.

A select list, created by build_selist(), is freed.

build_selist()

```
static int build_selist(svrattrl *list, int permission,
                        struct select_list **select, queue **pque, int *bad)
```

Args:

list pointer to a list of svrattrl structures from the select request.

permission
the client's privilege level.

selectRETURN: pointer to a pointer to a select list. The location of the create select list is returned here.

pque RETURN: pointer to a queue pointer. If job search is limited to a queue, this is set.

bad RETURN: pointer to an integer which will be set to the index (starting with 1) of a bad attribute.

Returns:

0 if the selection list was built.

error number if an error occurred.

The parameters above marked as returns.

For each member of the svrattrl (attribute) list from the request, a select_list structure is allocated, the attribute is decoded into the structure, the operator is set from the request, and

the structure is linked into the select_list. All this is done via the call to *build_selentry()*.

If an ATTR_q (-q) pseudo-attribute was specified, a search is make for a queue of that name and a pointer to it is returned in pque.

Another special case is when the attribute is for -s, JOB_ATR_state, the actual attribute is single character of type {ATTR_TYPE_CHAR}, but the selection may be a string of multiple letters, see the -s option is qselect(1). Hence there is a special attribute definition structure for this case which decodes a string and supplies a special comparison routine, *comp_state()* which compares each letter of the selection string with the job's state.

5.3.11.16. req_shutdown.c

The file *src/server/svr_shutdown.c* contains the functions to gracefully terminate the server.

req_shutdown()

```
void req_shutdown(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

Returns:

0 if success.

non
if error.

The requesting user must have operator or administrator privilege or the request is rejected. The address of the shutdown request is saved in *pshutdown_request* for the function *shutdown_ack()*. Then the function *svr_shutdown()* is called with the type of shutdown requested.

shutdown_ack()

```
void shutdown_ack()
```

This function is called from the server's main routine just before it exits. The purpose is to check if the shutdown is because of a request (qterm) and reply to it.

svr_shutdown()

```
void svr_shutdown(int type)
```

Args:

type The type of shutdown requested.

The server state is set to indicate the type of shutdown:

-
{SV_STATE_SHUTIMM}
for type immediate,
{SHUT_IMMEDIATE},

or for receipt of signal SIGTERM.
-
{SV_STATE_SHUTDEL}
for type delay,
{SHUT_DELAY},
-
{SV_STATE_DOWN}
for type quick,
{SHUT_QUICK}.
to restrict services. Note, a SHUT_IMMEDIATE or a SIGTERM while the server is in state SV_STATE_SHUTIMM will force the server into SV_STATE_DOWN. The type of shutdown is recorded in the event log. If the shutdown type is quick, return now; the main loop will be broken.

For each job managed by the server, if the job is in the {JOB_STATE_RUNNING} state, the following actions are performed:

- The {JOB_SVFLG_HOTSTART} and {JOB_SVFLG_HASRUN} bits are turned on in *ji_svrflags*.
- If checkpoint/restart is supported and the job checkpoint attribute is not "n", then an attempt is made to checkpoint and terminate the job is made by calling *shutdown_chkpt()*.
- Else if the job cannot be checkpointed or the checkpoint fails, then attempt to re-run the job or kill it off by calling *rerun_or_kill()*.

shutdown_chkpt()

```
static int shutdown_chkpt(job *job)
```

Args:

job pointer to the job to checkpoint.

Returns:

0 if the checkpoint request (hold request) was successfully set to MOM.

non-zero
error number if not.

A batch_request structure is allocated and set up as a Hold Job request. This request is sent to MOM, *relay_to_mom()*, for action. The routine *post_chkpt()* will in invoked when MOM responds.

post_chkpt()

```
static void post_chkpt(struct work_task *task)
```

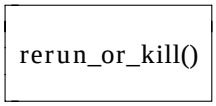
Args:

task pointer to the work task entry.

This function is called when MOM replies to a request sent by *shutdown_chkpt()*. If the checkpoint/hold was successful, either the {JOB_SVFLG_CHKPT} or {JOB_SVFLG_ChkptMig} bit is set in the job server flags, *ji_qs.ji_svrflag* depending on the checkpoint type return information

from MOM. The checkpoint type is found in the *brp_auxcode* word of the reply to the checkpoint request.

Otherwise, we attempt to rerun the job or kill it off by calling *rerun_or_kill()*.




```
void rerun_or_kill(job *pjob, char *text)
```

Args:

pjob pointer to job to rerun or kill off.

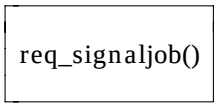
text message to log, the reason this function is being called.

If the job attribute JOB_ATR_rerunable is true, a {SIGKILL} signal request is sent to MOM. The job substate is set to {JOB_SUBSTATE_RERUN} to indicate to post job execution processing that the job is not to be discarded.

If the job cannot be rerun, and the server state is not {SV_STATE_SHUTDEL}, *job_abt()* is called to kill off the job.

5.3.11.17. req_signal.c

The file *src/server/req_signal.c* contains the server function for processing the Signal Job batch request.




```
void req_signaljob(struct batch_request *req)
```

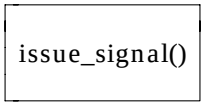
Args:

req pointer to the batch_request structure.

The job must be in state {JOB_STATE_RUNNING}. The signal value supplied in the request is a string, it may either be a numeric string or an alphanumeric signal name. The special names `suspend` and `resume` are reserved for the special suspend/resume functions. Use of these names require manager or operator privilege.

The request is forwarded to MOM by *relay_to_mom()*. Note, if the signal value is a numeric string, MOM will convert it to the corresponding integer value. If it is a name, which may or may not have the"SIG" prefix, the name is converted to the correct signal value. If the name is not known on the execution system, the request is rejected with error [PBSE_UNKSIG].

When the MOM replies, the function *post_signal_req()* is invoked to generate the reply to the client.




```
int issue_signal(job *pjob, char *signal, void (*func)(struct work_task *),
                 void *extra)
```

Args:

pjob pointer to job structure of job to be signaled.

signalalphabetic signal name or numerical string value of signal to send to job.

func the function to invoke when the reply to the signal request is received.

extrabit of information to insert in generated signal job batch_request structure.

Returns:

0 if successful.

-1 if error.

This function is provided to allow the server itself to initiate a signal to a running job. A Signal Job batch request structure is allocated via *alloc_br()* and initialized. The void pointer extra is inserted into the structure in rq_extra. The request is sent to the MOM in charge of the job by calling *relay_to_mom()*. On the reply from MOM, the function *release_req()* is invoked which just frees the batch_request structure.

An error is returned from issue_job only if it cannot allocate the batch_request structure or if relay_to_mom fails. We have no idea what MOM did with the signal.

When MOM replies to the Signal Job request, the function specified as func will be invoked via the work task mechanism. *This function MUST free the batch request and close the connection*. The easiest way is to call *release_req()*.

The extra parameter, and in fact the func post process function were added to issue_signal() to generalize it (how does one spell "kludge") for req_delete.c.

post_signal_req()

```
static void post_signal_req(struct work_task *task)
```

Args:

task pointer to the deferred child work task.

When MOM replies to a Signal Job request forwarded to her on behalf of an external client, this function will receive her reply and relay its code to the client.

If either of the special signal names, suspend or resume, was issued and MOM acknowledged the request without error, the flag {JOB_SVFLG_Suspend} is updated in ji_svrflags (set for suspend, cleared for resume). The job_state attribute value letter is changed to S or R by calling *set_statechar()*.

5.3.11.18. req_stat.c

The file *src/server/req_stat.c* contains the server functions for providing status about

- A job or set of jobs in reply to a Status Job batch request. The client may request status of a single job by supplying the job id, or a set of job by supplying a destination id. If a destination id is supplied, then status of all jobs at that destination, a queue, that the user is entitled to status is returned.
- A queue or all the queues in owned by the server.
- The server itself.

req_stat_job()

```
void req_stat_job(struct batch_request *req)
```

Args:

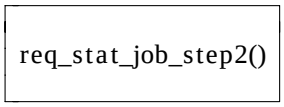
req pointer to the batch_request structure.

If the id supplied in the request, rq_id, is not null and begins with a numeric character, the request is for status of a single job whose id is specified.

If the id in the request is not null and begins with a alphabetic character, then the id specifies a queue. An attempt is made to locate the queue of that name. If the queue does not exist on this server, [PBSE_UNKQUE] is returned to the user.

Else if the id in the request is null, or starts with the '@' character, then the request is for all jobs in the server.

A private status control structure is allocated and initialized to hold the type of status and pointer to the job or queue as required. This structure is passed to *req_stat_job_step2()*.




```
static void req_stat_job_step2(struct stat_cntl control)
```

Args:

control

is a pointer to privately defined status control structure.

This function, *stat_step_2()*, is a effect of the complication of having MOM be responsible for running jobs. When a user requests status of a running job, the user expects to see information about resource utilization by that job. This implies that the server must obtain reasonably current status information from MOM for each job the client requested status. Additional complication arises from the desire to kept the Server free from waiting on any other server, that is no synchronous requests. As the server works through the list of jobs for which the client requested status, rather than ask MOM for an update and block waiting on her reply, each time the server goes to a MOM, a work task is established and the server returns to its main loop. This adds two additional routines *stat_to_mom()* and *stat_update()*.

The checking state of jobs and asking MOM for recent updates and then building the final reply to the client is done in two separate passes. This is to eliminates the possibility of starting to build the status for a job and having to go to MOM, only to have the job disappear before we hear from MOM.

The first part of *req_stat_job_step2()* checks each job for which status is requested. The type of request, single job, jobs in queue, all jobs, as well as the last job checked is passed in the status control structure. The "last job checked" is null the first time in, this causes stat_step_2 to start with the first job in the queue or server list, or the single job in the request.

If any job is running and the last update from MOM was received more than {PBS_RESTAT_JOB} seconds ago, then it goes to MOM again. PBS_RESTAT_JOB is used to keep the server from flooding MOM with status request for a anxious user. The function *stat_to_mom()* is it to the appropriate MOM. If the user asked for status of a single job, that is all we ask from MOM, otherwise we ask MOM for status of all her jobs. This may save additional requests later. At this point req_stat_job_step2 returns back to the main server loop. When MOM replies, the action picks up in *stat_update()* which updates the job status information and re-invokes stat_step_2() passing it a pointer to the *stat_cntl* structure used to maintain the position amoung the jobs. The process continues with the next job. This explains the funny initialization of *pjob* with in the while() loop. Note, if the job disappears while the server is waiting for MOM to reply to the status, the server just starts over as find_job() returns a null, the starting condition.

The second part of req_stat_job_step2 (which should be step 3) is to loop back through the jobs and build up the status reply to be returned to the user. This is done by calling *status_job()* for each job for which status is being provided. Then, at long last, the status can be returned to the client. Note, if status_job returns any non-zero status other than [PBSE_PERM], that error is returned to the client. If PBSE_PERM is returned, that job is ignored, it is invisible to the client.

stat_to_mom()

```
int stat_to_mom(job *pjob, struct stat_cntl *control)
```

Args:

pjob pointer to the job.

control
pointer to the status control structure.

Returns:

0 if no errors.

error number if problem.

A Status Job batch request is created and initialized, see *alloc_br()*. This request has a pointer to the status control structure. A connection is opened to MOM by calling *svr_connect()*. The connection is maintained until MOM replies. The status request is sent to MOM by calling *issue_request()*.

stat_update()

```
static void stat_update(struct work_task *task)
```

Args:

task pointer to the deferred child work task.

This function is invoked by *process_reply()* when the reply to a status request is received from MOM. Per the overall paradigm (does a paradigm make four nickels?), process_reply calls a specific processing routine identified in a work task structure associated with the connection. This work task points to the original batch request structure. In this case, the specific processing routine is stat_update and the batch request structure also points to the private status control structure.

For each object status element returned, the job structure is located by calling *find_job()* with the job name from the reply. The job attributes contained in the reply are passed to *modify_job_attr()* which updates the job structure. Note, the {ATR_DFLAG_FSET} flag is set in the permissions passed to modify_job_attr. This allows "Read Only" attributes, such as the Session ID to be modified.

If *ji_momstat* is zero in the job structure, this is the first update since the job started to run. Hence we should save the job info to disk with a call to *job_save()* with SAVEJOB_FULL. ji_momstat is set non-zero so we will not save after future updates from MOM.

If the job structure could not be found, it might have been deleted after we issued the request to MOM. We just ignore the situation here. When *req_stat_job_step2()* discovers the missing job, it will restart the update process from the beginning of the queue or server's list. Without the job, we cannot continue to the next because the link field has been unlinked and

freed.

In either case, the batch request built to send to MOM is freed and the connection is broken. Typically the the routine that called stat_to_mom(), likely req_stat_job_step2(), is specified in the status control structure. This routine is re-invoked to continue with the next job. Should no routine be specified, see *stat_mom_job()*, the control structure is freed even though it was not allocated here. This saves an extra function just to do that.

stat_mom_job()

```
void stat_mom_job(job *pjob)
```

Args:

pjob pointer to a single job.

This routine is a special front end to *stat_to_mom()* to allow functions outside of this source file to issue a status call to MOM. The primary user is *post_sendmom()*. We need to obtain the session id of the job newly placed into execution.

A status control structure is built and passed along with the job pointer to stat_to_mom(). In this case, the function to invoke after MOM replies is null.

req_stat_que()

```
void req_stat_que(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

The reply structure is initialized. If the id in the request is either the null string or a null pointer, then status of all queues at the server is being requested. The routine *status_que()* is called in turn or each queue managed by the server.

Otherwise, it is a request for status of a single specified queue. The queue is located and *status_que()* is called for that queue. If the specified queue does not exist, then [PBSE_UNKQUE] is returned.

status_que()

```
void status_que(queue *pque, struct batch_request *preq, list_head *preqattr)
```

Args:

pque pointer to the queue structure.

pliststat
pointer to the head of the list to which a status structure is appended.

preq pointer to the batch request, used to access the requested attribute list and client permissions.

A status structure is allocated, the object type is set to "queue," and the object name to the queue name. The structure is linked to pliststat.

The private function *status_attrib()* is called to encode and attach the attributes of the queue to the reply.

req_stat_svr()

```
void req_stat_svr(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

A status structure is allocated, the object type is set to "server," and the object name to the server name. The structure is linked to pliststat.

The private function *status_attrib()* is called to encode and attach the attributes of the server to the reply.

update_state_ct()

```
static void update_state_ct(attribute_def *padef, attribute *pattr,
                            int ct_array)
```

Args:

padefpointer to an attribute definition.

pattr pointer to an attribute value.

ct_array
pointer to the array of integers which holds the count of jobs per state.

This function is used to update the "jobs per state" attribute of queue and the server. It is called whenever a status request is made of the queue or server. The count of the number of jobs in each state is maintained in private data space within the queue or server structure. These values are converted to strings and placed in the public attribute.

The data space for the public Jobs by State attribute is a fixed character array in the server or queue structure. Note the special *decode_null()* and *set_null()* routines associated with this attribute.

5.3.11.19. stat_job.c

The file *src/server/stat_job.c* contains functions to support the Status Job Request. These are separated to make them available for use in MOM.

status_job()

```
int status_job(job *pj, batch_request *preq, svrattrl *pal,
               list_head *pliststat, int *bad)
```

Args:

pj pointer to the job structure.

preq pointer to the batch request.

pal pointer to first of a list of svrattrl structs containing attributes to be returned.

pliststat
UPDATED: pointer to the head of the list to which a status structure is appended.

bad UPDATED: set if one of the specific attribute in pal is invalid.

Returns:

0 if no error.

non zero
error number if error occurred.

The privilege to read (request status of) the job is validated. If the client does not have operator or manager permission, then the request is accepted only if the client is the job owner or the server allows all jobs to be read, see server attribute SRV_ATR_query_others. If the client is denied access, [PBSE_PERM] is returned.

A status structure is allocated, the object type is set to job, and the object name is set to the job identifier.

The state attribute {JOB_ATR_state} is updated from the ji_state field in the job structure.

The attributes of the job are encoded and attached to the reply structure by *status_attrib()*.

status_attrib()

```
static void status_attrib(svrattrl *pal, attribute_def *padef,
                          attribute *pattr, int limit, int priv,
                          list_head *phead, int *bad)
```

Args:

pal pointer to the list of requested attributes.

padefpointer to the attribute definition structure array for the object.

pattr pointer to the parent objects attributes.

limit the number of attributes in the above arrays.

priv the privilege of the client.

pheadpointer to the head of the list in the reply structure to which the encoded attributes are linked.

bad UPDATED: set to the index of the first invalid attribute in pal.

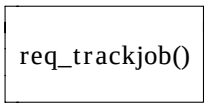
If no specific attributes of the statused object were requested, the list pointed to by pal will be empty (null), then each attribute of the job which is *readable* with the client level of privilege is encoded into a svrattrl structure by calling the *at_encode()* routine for the attribute. The svrattrl entry is appended to the list headed in the status structure.

If specific attributes were specified in the batch request, the list pointed to by pal is not empty, then only those attributes which are known to the server, and readable are returned to the client. For each attribute above, the corresponding attribute entry is located and encoded into a svrattrl as above.

Note that MOM's version of this routine is simplier. MOM encodes for the status reply only those attributes listed in an array of specified attributes, *mom_rtn_list*, contained in this file.

5.3.11.20. req_trackjob.c

The file *src/server/req_track.c* contains the server functions for recording job tracking information received in a Track Job batch request. The information is recorded in a member of a *tracking* array. There is a pointer, sv_track, to the array in the server structure, as well as its current size of the array, sv_tacksize, and a flag, sv_trackmodified, indicating if the structure has been modified.



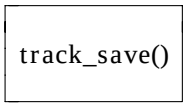

```
void req_trackjob(struct batch_request *req)
```

Args:

req pointer to the batch_request structure.

The tracking array is searched for a matching job id. In case it is not found, a pointer is kept to where in the array to insert a new record. If an entry with a matching job id is located and its hopcount is less than that in the request, it is updated with the new information from the request. Otherwise a new entry is allocated, set with the information from the request, and linked into the list.

The *sv_trackmodified* flag is set in the server to indicate the list has been modified since the last time it was saved. information




```
static void track_save()
```

This function saves job tracking entries to disk. If the server flag *sv_trackmodified* is not set, there are no updated entries, so just exit.

The save file specified in path_track is opened and the save buffer is written out. Then the save file is closed. The *sv_trackmodified* flag is cleared.

5.3.12. Job Router Overview

The purpose of the Job Router is to find a destination queue which matches the requirements for a job in a route queue. Each queue given as a destination for a route queue is tried. If the destination is local (in the same server that contains the route queue), the communication with the destination queue is internal. If not, a process is created to deal with sending the job over the network.

Each attempt to send a job to a queue starts with a Queue Job Request which includes information about the requirements for the job. If the queue can accommodate the job, it accepts the queue request. If not, it rejects it. If the error return indicates the rejection is permanent, the queue name is added to a list kept with each job of destinations to not try again.

5.3.12.1. job_route.c

The major functions in file *src/server/job_route.c* which make up the Job Router are described below.

add_dest()

```
badplace *add_dest(job *pjob)
```

Args:

pjob The job which has an entry made in its bad destination list.

Returns

pointer
if call is successful.

NULLIf call is not successful.

is_bad_dest()

```
badplace *is_bad_dest(job *pjob, char *dest)
```

Args:

pjob The job to check for the destination.

dest The destination to look for.

Returns

pointer
If dest is found.

NULLIf dest is not found.

The list of badplace structures attached to the job is searched for one with the specified destination. If found a pointer to it is returned, otherwise a null pointer is returned.

default_router()

```
int default_router(job *pjob, pbs_queue *pque, long retry)
```

Args:

pjob pointer to job to route.

pque pointer to queue in which the job resides.

retry next time to retry the route.

Returns:

0 if job is being routed or is still ok in the queue, non-zero if cannot be and should be aborted.

An attempt is made to route the job to each destination listed in order in the queue attribute QR_ATR_RouteDestin. Upon having attempted the last destination, if *ji_retryok* in the job structure is false, no destination would accept the job, that is logged and [PBSE_ROUTEREJ] is returned. If *ji_retryok* is true, at least one destination can be retried at *retry* time, zero is returned.

Foreach destination, *is_bad_dest()* is called to check if the current destination is listed in the job structure as a "bad" destination, one which has permanently rejected the job. If bad, the

next destination is tried. The function *svr_movejob()* is invoked to attempt the move (route) the job to the current trial destination. If it returns -1, the current destination is added to the bad list by calling *add_dest()*. If the move succeeded, or is underway (move to a different server), we return zero. If svr_movejob() returns 1, the move failed, but may be retried, so *ji_retryok* is set true and the next destination is tried.

job_route()

```
int job_route(job *job)
```

Args:

job The job which is to be routed.

Returns:

0 If call is successful. Note, the job may still be "owned" by the local server.

non-zero
error number if call failed.

Check the job state. If the job is in state {JOB_STATE_TRANSIT}, ignore it, it is already routing. If the job is in {JOB_STATE_HELD} and attribute QR_ATR_RouteHeld is not true or the job is in state {JOB_STATE_WAITING} and attribute QR_ATR_RouteWaiting is not true, then we will ignore the job shortly. If the job is in any other state other than the above or {JOB_STATE_QUEUED}, a record is added to the log and the job is ignored.

Next we check the queue in which the job resides. It must be started, QA_ATR_Started true, and if the queue attribute QA_ATR_MaxRun is set the number of jobs in the queue in state {JOB_STATE_TRANSIT} must be less than that specified in the attribute.

If the job has been laying around in the queue for longer than the allowable life time, QR_ATR_RouteLifeTime, return [PBSE_ROUTEEXPD]. The retry time is calculated to be the current time plus either the value of the queue attribute QR_ATR_RouteRetryTime if set, or the default retry time {PBS_NET_RETRY_TIME}. If the job is to be ignore because of its state we do that now (after the test for life time).

We are now in the main routing loop. If the job has been through all the possible destinations without being routed we check the retry flag, ji_retryok. If it is cleared, all destinations rejected the job for reasons which seem permanent, [PBSE_ROUTEREJ] is returned. If any destination rejected the job for "temporary" reasons, unable to contact the server, or the queue was not enabled, the route retry time for the job, ji_un.ji_routet.ji_rteretry, is set to the retry time and zero is returned.

Otherwise, we have more destinations to try. The next one is selected and *is_bad_dest()* called to determine if it is on the "bad" list. If not, *svr_movejob()* is called to attempt to route the job. If svr_movejob returns an indication that the destination gave a permanent rejection, the destination is added to the bad list by *add_dest()*. If the rejection is temporary, the retry flag, ji_retryok, is set and we go on to the next candidate destination. Otherwise, the route is in progress or has be completed (if local) and so zero is returned.

queue_route()

```
void queue_route(queue *que)
```

Args:

que pointer to a routing queue.

For each job whose route retry time, ji_un.ji_routet.ji_rteretry, has been reached, we call *job_route()*. If job_route() returns [PBSE_ROUTEREJ], rejected by all destinations, or [PBSE_ROUTEEXPD], life in queue expired, the job is aborted.

5.3.12.2. svr_movejob.c

The major functions in file *src/server/svr_movejob.c* which make up the Job Mover are described below.

svr_movejob()

```
int svr_movejob(job *job, char *destination, batch_request *request)
```

Args:

job The job which is to be routed.

destination
The destination queue where the job will be sent.

request
The batch request from the client or NULL if this is from route.

Returns:

0 If move is complete. The job is now owned by the destination queue.

-1 If call failed. The job has not moved.

1 A "temporary" failure. The call failed but may be tried again.

2 The move is deferred (in progress). A child has been created to process it and will return sometime in the future.

Copy the destination into the job structure. If the destination is local to this server, call *local_move()*, else call *net_move()*.

local_move()

```
int local_move(job *job, batch_request *request)
```

Args:

job The job which is to be routed.

request
The batch request from the client or NULL if this is from route.

Returns:

0 If route is complete. The job is now owned by the destination queue.

-1 If call failed. The job has not moved.

1 The move failed but may be retried.

Search for the destination queue, if it does not exist return -1. If the queue is not enabled, return 1. If the job is not being move at the specific request of the administrator, then check the resource requirements of the job against the queue limits via The function *svr_chkque()*

is called to check the destination queue state and the resource requirements of the job against the queue limits. The type of move (route, non-privileged user move, privileged move) determines what items are enforced in svr_chkque(). If the job requirements fit the destination queue limits, unlink job from current queue via *svr_dequejob()*, reset the queue rank job attribute JOB_ATR_qrank to a new value (job goes to the end of the queue), and link into queue via *svr_enquejob()*.

net_move()

```
int net_move(job *job, batch_request *request)
```

Args:

job The job which is to be moved, or routed.

request
The batch request from the client or NULL if this is from route.

Returns:

2 If no error occurred. The job is in the state JOB_STATE_TRANSIT. A child has been created which will return with a status indicating success or failure.

-1 If call failed. The job has not changed state.

Returns from child:

0 If route is complete. The job is now owned by the destination queue.

1 If call failed. The job has not moved.

2 The move failed but may be retried.

This function serves double duty. It is used to route a job (from a routing queue, see *job_route()*), or to move a job (a move request) to another batch server.

The server name (host name) and service port is determined by passing the destination substring following a "@" character to *parse_servername()*. The host address is obtained from *get_hostaddr()*. The job state is set to {JOB_STATE_TRANSIT}. This information, along with the type of move and post child processing function, is passed to *send_job()* to actually fork a child to send the job.

If the batch_request pointer is not null, the move is the direct result of a Move Job batch request. The move_type parameter is set to {MOVE_TYPE_Move}, the post child processing function desired is *post_movejob()*, and the data pointer to place in the work task points to the request. Otherwise, the move results from a route operation. The move_type parameter is set to {MOVE_TYPE_Route}, the post child function is *post_routejob()*, and the data pointer is set to NULL (after all, there is no request to which to point).

send_job()

```
int send_job(job *pjob, pbs_net_t address, int port, int move_type,
             void (*post_func)(struct work_task *),
             void *data_pointer)
```

Args:

pjob pointer to the job to be sent.

address
of the destination server (host).

port number for the service (server or MOM).

move_type
the type of send: move, route, or execute (to MOM).

post_func
address of a function to invoke after completion of the move/route.

data_pointer
pointer to the data of interest to the post child function, saved in the work task.

Returns:

2 if the child was successfully created (see svr_movejob).

-1 if error, pbs_errno set to the error number.

The death-of-child signal is blocked until the work task is set and the child is underway. A child process is forked to do the queue job request sequence.

The parent creates a work task to be dispatched on death of the child. The job pointer is passed to *set_task()* to be placed in wt_parm1. The data_pointer item, either NULL or a pointer to the batch request is inserted into wt_parm2. The post processing routine had better expect what is in wt_parm2. The dispatched function depends on the type of move. It is passed in as post_func, and is typically:

post_routejob() if the move type is route.
post_movejob() if the move type is move.
post_to_mom() if the move type is execute.

Now, unblock the death-of-child signals and return 2.

The created child process, the router performs the following actions. It sets up a signal catcher to insure an error return. The job attributes are encoded into a list of svrattrl structures. The encoding mode is according to the destination, {ATR_ENCODE_MOM} if the job is being sent to MOM, move type is {MOVE_TYPE_Exec}; and {ATR_ENCODE_SVR} if the job is being routed to another server, move type is {MOVE_TYPE_Route}. The svrattrl structures contain the attrl structures required by the API routines in libpbs.a. The attrl sub-structures are correctly linked by calling *attrl_fixlink()*. The path name of the job's script file is set up based on the file prefix information in ji_qs.ji_fileprefix.

The following steps are tried several times:

If this is not the first time around the loop, there must have been an error the prior time. Disconnect from the server. Call *should_retry_route()* to determine if we should retry, if not exit with a status of 1.

Connect to the destination server by calling *svr_connect()*. If the connection fails for a reason marked by svr_connect as {PBS_NET_RC_FATAL}, the failure is recorded in the log and and exit status of one (1) indicates the permanent failure. If the failure is not permanent, continue with the next cycle around the loop.

If the job is already in substate {JOB_SUBSTATE_TRNOUTCM}, we are attempting to complete an interrupted job send operation. We skip steps up to sending the "ready to commit".

Call the API routine *_pbs_queuejob()* to send the job attributes. If the job has a checkpoint file at MOM, *JOB_SVFLG_CHKPT* is set and if the move is a send to MOM, then skip steps up to the commit step.

Call the API routine *_pbs_jscript()* to send the script file.

If the move type is to MOM and the job has already been run once, {JOB_SVFLG_HASRUN} set, then copy over the job's standard output, error and (if exists) migratable checkpoint file.

Now block all signals so the final stages of the transfer cannot be stopped by the server. Send the ready to commit by calling the API routine *_pbs_rdytocmt()*. If it is rejected, unblock signals and continue the next cycle around the loop.

The receiving server now has everything and except when sending to MOM for execution, we purge our copy to prevent duplicate jobs. If the move type is not type **execute**, delete the job files by calling *job_purge()*. Send the commit, call *_pbs_commit()*.

Disconnect from the destination server, and indicate a successful move with an exit of 0.

post_routejob()

```
void post_routejob(struct work_task *pwt)
```

Args:

pwt A pointer to the work task entry.

This function is invoked from a work_task entry when the job router process terminates. The work_task member wt_parm1 points to the job being routed and wt_aux is set to the exit status of the router. If the router exit status shows the job was sent ok:
- if files where already staged-in, call *remove_stagein()*,
- delete the job by calling *job_purge()* and return 0.

If the exit status indicates a permanent failure, its either a "bad" destination or the router caught a signal. If the job substate is set to {JOB_SUBSTATE_ABORT}, the server has received a request to delete the job, so stop the routing; another work task will complete the delete process. Otherwise the destination is bad, mark it not to be tried again for this job, *add_dest()*. On either a permanent or temporary failure, attempt to route to the next destination by recalling *job_route()*. If job_route returns any error abort the job.

post_movejob()

```
void post_movejob(struct work_task *pwt)
```

Args:

pwt pointer to the work task entry.

This function is invoked from a work task entry when the job router process has terminated. The route was the result of a Move Job request. The pointer to the batch_request structure for the move request is in wt_parm1. The exit status of the child process which attempted the move (route) is in wt_aux.

When the child process which was forked to perform a route operation in response to a Move Job batch request terminates, this function is called by the work task dispatch routine.

A reply is returned to the client based on the exit status of the routing child process. If the status is zero, there were no errors and the job has been routed to a new server. If files had been staged-in for the job, they are deleted by calling *remove_stagein()*. The job is purged and a success reply is returned to the client.

If errors occurred, the job still exists on this server. An error reply is returned to the client and the job is requeued by setting the state to the value returned by *svr_evaljobstate()* and calling *svr_setjobstate()*.

should_retry_route()

```
static int should_retry_route(int error)
```

Args:

error to be examined to determine retry or not retry.

Returns:

1 if the route should be retried.

-1 if the route should not be retried.

this function looks at the error passed as a parameter and determines if the route should be retried.

5.3.13. Header Files

5.3.13.1. attribute.h

The structures, symbols, and access function prototypes needed to declare and define attributes are located in this header file.

Attributes are represented in one or both of two forms, external and internal. When an attribute is moving external to the server, either to the network or to disk (for saving), it is represented in the external form, a *svrattrl* structure. This structure holds the attribute name as a string. If the attribute is a resource type, the resource name is encoded as a string, else it is null. The value of the attribute (or resource) is encoded into a third string. The structure contains a length field for all three strings and a field which gives the over all size of the svrattrl structure and the appended strings.

Internally, attributes exist in two separate structures. The attribute type is defined by a definition structure, *attribute_def*, which contains the name of the attribute, flags, and pointers to the functions used to access the value. This info is "hard coded". There is one "attribute definition" per (attribute name, parent object type) pair.

The attribute value is contained in another structure, *attribute*, which contains the value with in a union of the possible value types. The possible types are:

ATR_TYPE_LONG
: the data is arithmetic or boolean and fits in a C long type internal to the structure.

ATR_TYPE_CHAR
: the data is a single character and is maintained internal to the structure.

ATR_TYPE_STR
: the data is null terminated sting. Storage for the data is on the heap and a pointer to it is in the attribute structure.

ATR_TYPE_ARST
: the data is an array of strings. The value in the attribute structure points to a *array_strings* structure on the heap. This structure has an array of pointers to each string. The strings are maintained on the heap in contiguous storage.

ATR_TYPE_SIZE
: the data is a size. It is maintained as a long integer and two flag sets which specify K,M,G,T and bytes or words.

ATR_TYPE_RESC
: the data is list of resources, see resource.h. Each resource is on the heap.

ATR_TYPE_LIST
the data is list of other structures. Each member of the list is on the heap.

ATR_TYPE_ACL
the data is an Access Control List. It is maintained as an array of strings, ATR_TYPE_ARST, but marked differently to aid in the saving to/recovery from disk.

Privilege to access an attribute is defined by the bit wise "inclusive or" of the following as set in the attribute definition:

ATR_DFLAG_USRD
readable (status can be obtained) by a non-privileged user client.

ATR_DFLAG_USWR
writtable (can be set) by a non-privileged user client.

ATR_DFLAG_OURD
Reserved.

ATR_DFLAG_OUWR
Reserved.

ATR_DFLAG_OPRD
readable by a client with operator privilege.

ATR_DFLAG_OPWR
writtable by a client with operator privilege.

ATR_DFLAG_MGRD
readable by a client with manager privilege.

ATR_DFLAG_MGWR
writtable by a client with manager privilege.

ATR_DFLAG_SvRD
readable (will be sent to) another server or the scheduler.

ATR_DFLAG_SvWR
writtable (can be set by) another server or the scheduler.

ATR_DFLAG_MOM
Sent to MOM with the job when it is to be run. Those and only those attributes (and resources) so marked are sent to MOM. Applies to Job attribute/resources only.

The following bit wise flags are used by the Server, they are set in the attribute definition structure:

ATR_DFLAG_ALTRUN
the job attribute or resource can be altered while the job is running.

ATR_DFLAG_NOSTAT
the attribute is returned to a client only on specific request for this attribute. Can be used to shorten the list seen with a “qstat -f”.

ATR_DFLAG_SELEQ
in a select operation, see qselect(1), the only legal operations are equal (.eq.) and not-equal (.ne).

ATR_DFLAG_RASSN
the job resource entry is to summed on the the server's resources_used attribute when the job is placed into execution, and subtracted when the job terminates.

ATR_DFLAG_RMOMIG
currently not used.

The following flags are maintained by the server in the attribute (value) structure:

ATR_VFLAG_SET
the attribute/resource is set, i.e. the value has meaning.

ATR_VFLAG_MODIFY
the attribute/resource has been modified either by a decode or set operation.

ATR_VFLAG_DEFLT
the value is set to a system defined default value. The value is neither saved nor sent to another server as the default may be different.

5.3.13.2. resource.h

This header file contains the definitions and declarations for resources. As discussed earlier, resources are a special case of an attribute, a linked list of attribute values headed in an attribute such as resource_list. Resources use similiar structures as attributes. Certain types, type related functions, and flags may differ between the two.

Within the resource structure, the value is contained in an attribute substructure, this is done so the various attribute decode and encode routines can be "reused".

Unlike "attributes" which are typically identical between servers within an administrative domain, resources vary between systems. Hence, the resource instance has a pointer to the resource definition rather than depending on a predefined index. Three routines are declared within the header file that are useful in finding or adding resources:

find_resc_def()
returns a pointer to the resource definition structure for a given resource name.

find_resc_entry()
returns a pointer to a resource entry in a resource list which points to the the supplied resource definition. Null is returned if no such entry exists within the list.

add_resource_entry()
will add an unset entry to the list.

All the flags and permission bits discussed under *attribute.h* apply to resources.

5.3.13.3. batch_request.h

This file contains the giant union into which all batch request are converted. Where possible, the fields are fixed length so the structure can be malloc-ed in one piece.

5.3.13.4. credential.h

This file contains the structures and constants used in producing a PBS authentication credential.

5.3.13.5. job.h

This header file contains the structure definition used by the Server and MOM to hold the job information. Note that there two parts to the job structure: the interior portion, sub-structure jobfix, contains the fixed length data for each job that is saved to disk; the remainder of the structure contains data that can be reconstructed and need not be saved.

A note on the Job State and Substate, the State is a gross indication of the job state which is returned to the user. The Substate is the actual state of the "job state engine."

5.3.13.6. list_link.h

This file contains the structure definitions, function prototypes, and access macros for managing a doubly linked list. The structures defined are:

list_link
This structure contains the forward and backward pointer for each list entry. It is typically placed as the first sub-structure of the structure defining the list entry.

```
typedef struct list_link {
    struct list_link *ll_prior;
```

```
    struct list_link *ll_next;
    void             *ll_struct;
} list_link;
```

list_head
A list_head is identical to a link_link structure with member ll_struct set to NULL.

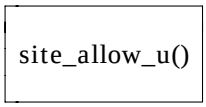
The macros CLEAR_LINK and CLEAR_HEAD are defined in this header file. The macros GET_NEXT and GET_PRIOR as also defined here. They expand either to in-line code or a function call depending on the setting of the symbol {NDEBUG}.

5.3.14. Site Modifiable Files

The files and functions described in this section provide a site the ability to customize PBS to meet special requirements. The supplied version of the c source files may be found in the *src/lib/Libsite* directory and are linked via the libsite.a library. How to modify these files is discussed in the IDS chapter on libsite.a. In addition, there are a set of header files, loaded into the target tree include directory which provide the capability to add new attributes.

5.3.14.1. site_allow_u.c

The file *src/lib/Libsite/site_allow_u.c* contains the following function:




```
int site_allow_u(char *user, char *host)
```

Args:

user The name of the user making a connection to the server.

host The name of the host from which the user is making a connection.

Returns:

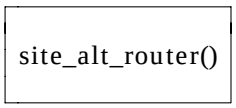
zero If the user is to be allowed access, zero (0) is returned. This is the default.

non-zero
If the user is to be denied access, a non-zero error code, typically [PBSE_PERM] should be returned.

The provided version always returns zero. A site may add code to perform whatever checks it wishes. Realize however, that this will be called on every new connection. A procedure that takes time will impact performance.

5.3.14.2. site_alt_rte.c

The file *src/lib/Libsite/site_alt_rte.c* contains the following function:




```
int site_alt_router(job *pjob, pbs_queue *pque, long retry)
```

Args:

pjob pointer to job to be routed.

pque pointer to queue in which the job currently resides.

retry next route retry time.

Returns:

zero if job is still alive (in queue or being routed), an PBS error code, [PBSE_ROUTEREJ], if the job has been rejected by all; the job will be killed.

As provided, this routine just calls the default router function, *default_router()*. A site may replace this function and "activate" it for a queue by setting the queue attribute QR_ATR_AltRouter (alt_router) to true. Please study the default router, *default_router()* to understand the required procedures which must be performed:

Destinations

are listed in the queue attribute QR_ATR_RouteDestin.

svr_movejob()

should be used to perform the route. It will return:

-1 if the destination rejected the job for a reason which is considered permanent; the destination should not be retried.

0 The route succeeded. This implies the route was to a local queue, see next return entry.

2 The route to a remote queue is under way (sending the job). The job will have been placed in Transiting state. When the sending completes, either (a) the job will have been moved and deleted locally, (b) the move failed and the destination added to the bad list, or (c) it can be retired, job requeued in route queue in a state other than Transiting.

1 The route (local) failed, but can be retried later.

5.3.14.3. site_check_u.c

The file *src/lib/Libsite/site_check_u.c* contains the following functions:

site_acl_check()

```
int site_acl_check(job *pjob, pbs_queue *pque)
```

Args:

pjob pointer to the job structure.

pque pointer to the candidate queue

Returns:

0 if job is allowed into the queue

non-zero

if job is not allowed into the queue

This routine determines if a job is allowed into a certain queue.

site_check_user_map()

```
int site_check_user_map(job *pjob, char *luser)
```

Args:

pjob pointer to the job structure.

luser the local user name.

Returns:

0 if user allowed to execute as login described by password entry.

-1 if user not allowed.

This routine determines if the job owner is privileged to execute as the user described by a password entry. The local user name is the login name selected by *geteusernam()* from the user-list attribute of the job in question.

The PBS default distribution module determines privilege by:

1. If the submitting host is the current host and the job owner name is the same as the login name selected, the privilege is granted.
2. If the hosts are different, privilege is granted by calling *ruserok(3N)*.

   This is not strictly POSIX conforming as POSIX does not define ruserok(). However until 1003.22 actually has a standard for distributed security...

5.3.14.4. site_map_user.c

The file *src/lib/Libsite/site_map_user.c* contains the following function:

site_map_user()

```
char *site_map_user(char *user, char *host)
```

Args:

user The user name on the specified host.

host The host name.

Returns:

pointer
to to the mapped user name.

This function provides a place holder for a mapping of a user name on one system to the user name on a common reference system. Given a user name and a host name, this routine will return a "mapped" or "common" user name. It is used for mapping in two situations:

- Authorization of requests − A site may run with users having different login names on different systems. If a user submits a job from system A and wishes to status it from system B where he has a different name, there must be some means to map the two users to a common name.
- Mapping job owner to execution user name where the submitting (qsub) host and the execution host have different name spaces for users.

The routine as supplied assumes a common name across all systems. Therefore it just returns the user name as given.

This function is called from the routine *svr_chk_owner()* in pbs_server which is used to determine if the requestor is the job owner, and from *geteusernam()* to determine the local execution name.

A site is free to modify this routine to map as required. No input data should be modified. If a different name is to be returned, it should be saved in a static character array of size {PBS_MAXUSER} as defined in pbs_ifl.h.

5.3.14.5. Adding Attributes to PBS

A site can add attributes to the server, to queues, or to jobs. Three sets of header files are provided for this purpose. The files are copied from empty (except for comments) template files (*.ht) in the src/include directory to the object (target) include directory when the target tree is set up. They are named `site_*_attr_*.h`. The first asterisk stands for one of `svr`(server), `que`(queue), or `job`; and the second asterisk stands for `enum` or `def`. An additional two files, named `site_qmgr_*_print.h` are provided for including the server and queue attributes in the output of the qmgr "print" sub-command.

The .ht files in the source tree should not be modified. Any modifications there may be lost with the next release of PBS. The .h files placed in the target/include directory will not be over written.

Files:
site_svr_attr_def.h
site_svr_attr_enum.h
site_qmgr_svr_print.h

Together, these files provide the ability to add attributes to the server. The attribute itself is defined in site_svr_attr_def.h and an enumerated index is added in site_svr_attr_enum.h. Attribute is defined by adding structures of the following form in site_svr_attr_def.h:

```
{   "attribute_name",
    decode_*,
    encode_*,
    set_*,
    comp_*,
    free_*,
    action_*,
    perm_flags,
    ATR_TYPE_#,
    PARENT_TYPE_SERVER
},
```

The quote marks and commas are required as shown. The asterisk (*) and pound sign (#) are replaced with the data type as found in *attribute.h* . The common data types are:

Data Type	*	#	free_*
Boolean	b	LONG	free_null
Long int	l	LONG	free_null
A character	c	CHAR	free_null
String	str	STR	free_str
Array of strings	arst	ARST	free_arst
Size	size	SIZE	free_null

Within a single attribute definition, the routines and data types must agree.

The enumeration with in site_svr_enum.h can be any name, but a name of the form: `SVR_SITE_ATR_name` is recommend to prevent name space conflicts. For each attribute element added (one set of stuff with in the braces) in site_svr_attr_def.h, there **MUST BE** one enumeration lable added in site_svr_attr_enum.h

The attribute names, as given in the site_svr_attr_def.h entries, may be added in site_qmgr_svr_print.h. If added, these attributes will be included in the output of a `print server` qsub sub-command. The format is:

```
"name_one",
"name_two",
```

For example, to add two new attributes named **foo** (a boolean) and **bar** (a string), the following are added:

In site_svr_attr_def.h

```
    {   "foo",
        decode_b,
        encode_b,
        set_b,
        comp_b,
        free_null,
        NULL_FUNC,
        NO_USER_SET,
        ATR_TYPE_LONG,
        PARENT_TYPE_SERVER
    },
    {   "bar",
        decode_str,
        encode_str,
        set_str,
        comp_str,
        free_str,
        NULL_FUNC,
        NO_USER_SET,
        ATR_TYPE_STR,
        PARENT_TYPE_SERVER
    },
```

In site_svr_attr_enum.h:

```
    SVR_SITE_ATR_foo,
    SVR_SITE_ATR_bar,
```

And in site_qmgr_svr_print.h:

```
    "foo",
    "bar",
```

Files:
site_que_attr_def.h
site_que_attr_enum.h
site_qmgr_que_print.h

The same information as given for the server attributes apply to defining queue attributes. The exception (there has to be at least one, right) is the parent type can be PARENT_TYPE_QUE_ALL for an attributes that applies to both execution and routing queues, PARENT_TYPE_QUE_EXC for execution queues only, or PARENT_TYPE_QUE_RTE for routing queues only.

Files:
site_job_attr_def.h
site_job_attr_enum.h

Again the same information holds. The parent type is PARENT_TYPE_JOB. There is no qmgr header file for job attributes.

[This page is blank.]

6. Job Scheduler

The Job Scheduler is a daemon that is run in conjunction with the PBS server. The Job Scheduler determines which job(s) to run, suspend, hold, or terminate based on a set site-specific policy.

PBS provides several implementations of a scheduler. A site may choose to use the Yacc/Lex based procedural language scheduler known as the BASL scheduler, the Tcl based scheduler, or to develop their own scheduler using the C framework which is provided.

6.1. The BASL Scheduler

The BASL language is a C-like procedural language. It provides a number of constructs and predefined functions that facilitate dealing with scheduling tasks. The idea behind BASL is that a scheduler writer writes a very simple program in BASL, compiles it, and then runs it. BASL is a high level language optimized for scheduler development. This language allows to user to write a simple or intermediate complexity in about 30-100 lines of code.

6.1.1. BASL Scheduler Overview

BASL consists of three major parts: (1) BASL language grammar, (2) Pseudo-compiler, and a (3) set of assist functions or helper functions. The idea behind BASL(2) is that a scheduler writer writes the main part of the scheduling code (sched_main()) in a pseudo-C language called BASL, then translates the code into C via *basl2c*, and finally, integrates the code with the PBS libraries by using a C compiler to produce the actual scheduler executable: *pbs_sched*. sched_main() will be called after each scheduling iteration and when the command received from the server is not one of: SCH_QUIT, SCH_ERROR, SCHED_NULL, SCHED_RULESET, SCHED_CONFIGURE, SCHED_RULESET.

The resulting *pbs_sched* will be able to accept the following arguments in the commandline:

[-L logfile] [-S port] [-d home] [-p print_file] [-a alarm] [-c configfile]

where -S is for specifying the scheduler port to use when talking to the local server, and alarm is for setting the time in seconds to wait for a schedule run to finish (default: 180s). Just like the other PBS schedulers, this BASL-written scheduler takes care of setting up local socket to communicate with the server running on the same machine, cd-ing to the priv directory, opening log files, opening configuration file (if any), setting up locks, forking the child to become a daemon, initializing a scheduling cycle (i.e. get node attributes that are static in nature), setting up the signal handlers, and finally sitting on a loop waiting for a scheduling command from the server. When an appropriate scheduling command is received, sched_main() (whose body was initially written in BASL), is called. Another view of BASL scheduling system is shown in figure 6−2.

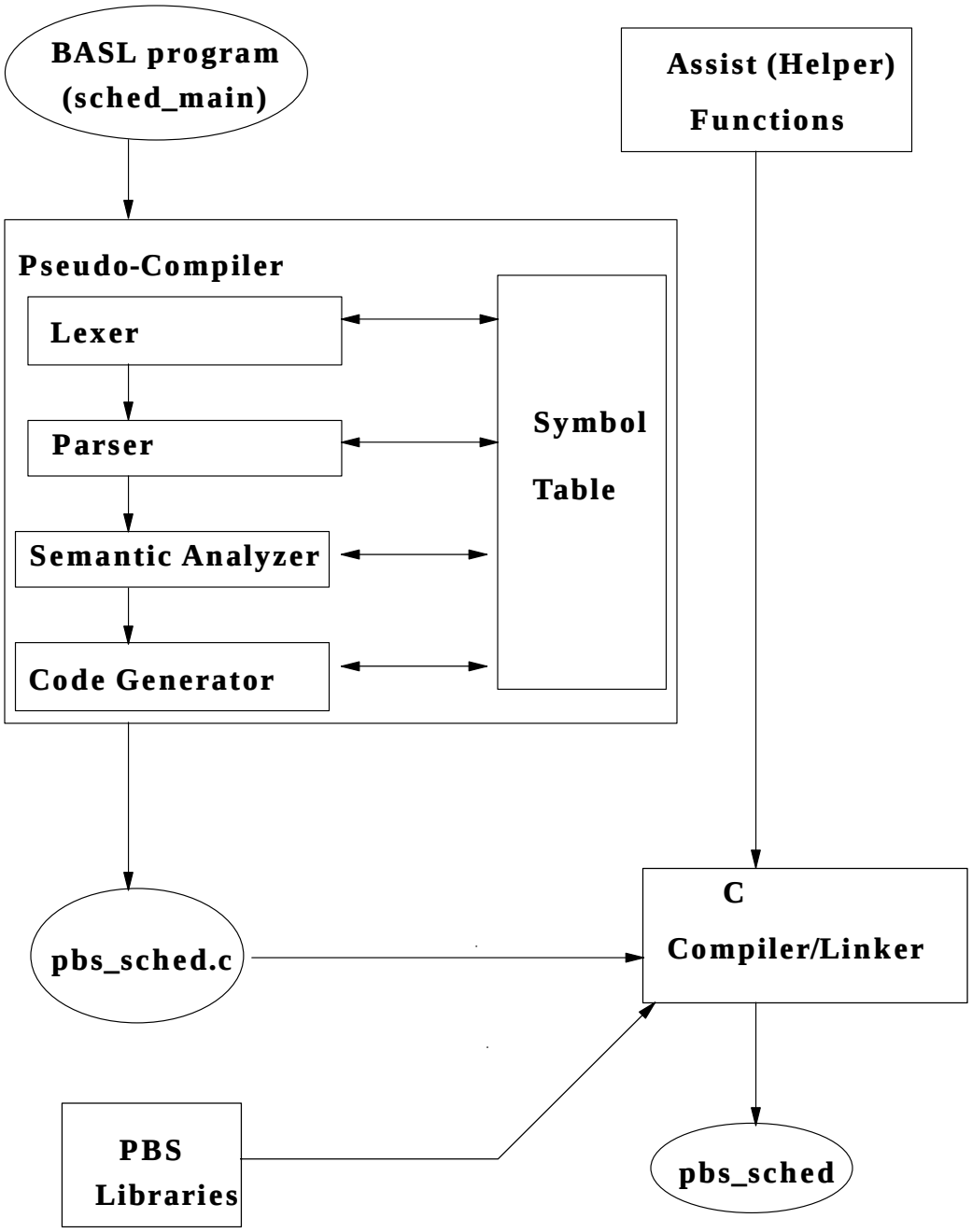


Figure 6 - 3 : BASL Software Architecture

6.1.2. Grammar

The Basl grammar subsystem consists of 5 parts: (1) Lexer, (2) Parser, (3) Symbol Table, (4) Semantic analyzer, and (5) Code Generator. Lexer is in charge of scanning an input file for valid tokens (input strings). Parser takes care of putting/combining the tokens together in a usable way. Semantic analyzer deals with checking to make sure that variables and operators are used in a consistent way. Symbol table holds information about matched tokens. Code generator is in charge of translating BASL statements into C statements.

Lex/yacc is used to specify and generate the code for the Lexer and Parser. The source codes involved in the grammar are found in the subdirectory *ParLex*.

6.1.2.1. Lexer

The files involved with the Lexer subsystem are Lexer.fl, ParLexGlob.h, Lexer.h, Lexer.c. Lexer.fl is the *lex specification* file that contains a set of patterns which lex match against the input. ParLexGlob.h, Lexer.h and Lexer.c contain various data structures and functions that assist lex in the scanning for input.

6.1.2.1.1. File: Lexer.fl

The tokens that will be matched by the BASL lexer are: *sched_main, Void, Int, Float, Dayofweek, DateTime, String, Server, Que, Job, CNode, Size, Set, Range, while, if, else, return, print, for, foreach, in, switch, case, break, continue, exit, Fun, EQ, NEQ, LT, LE, GT, GE, MIN, MAX, AND, OR, default, MON, TUE, WED, THU, FRI, SAT, SUN, ++, --, [+-]?[0-9]+, [+-]?[0-9]+[kmgtpKMGTP]?[bwBW], OP_EQ, OP_NEQ, OP_LE, OP_LT, OP_GE, OP_GT, OP_MAX, OP_MIN, SYNCRUN, ASYNCRUN, DELETE, RERUN, HOLD, RELEASE, SIGNAL, MODIFYATTR, MODIFYRES, SUCCESS, FAIL, SERVER_ACTIVE, SERVER_IDLE, SERVER_SCHED, SERVER_TERM, SERVER_TERMDELAY, QTYPE_E, QTYPE_R, SCHED_DISABLED, SCHED_ENABLED, FALSE, TRUE, TRANSIT, QUEUED, HELD, WAITING, RUNNING, EXITING, CNODE_OFFLINE, CNODE_DOWN, CNODE_FREE, CNODE_RESERVE, CNODE_INUSE_EXCLUSIVE, CNODE_INUSE_SHARED, CNODE_TIMESHARED, CNODE_CLUSTER, CNODE_UNKNOWN, NULLSTR, NOSERVER, NOQUE, NOJOB, NOCNODE, EMPTYSETSERVER, EMPTYSETQUE, EMPTYSETJOB, EMPTYSETCNODE, ASC, DESC [+-]?[0-9]+[.][0-9]*, ["][^"0*["], [a-zA-Z]+[a-zA-Z0-9_]*, //.*, 0*, [ ]+,..*

6.1.2.1.2. File: ParLexGlob.h

This contains the following structure to hold information about each of the tokens that the lexer has recognized during scanning for input:

```
struct MYTOK
{
  char lexeme[LEXEMSZ];
  int line;
  int len;
  int type;
  int varFlag;
};
```

lexeme contains the matched string, line is where in the input file the matched string was found, len is the string length of the matched token, type is some classification assigned to the token. This can be: {UNKNOWN, INTTYPE, FLOATTYPE, STRINGTYPE, STATUSTYPE, DAYOFWEEKTYPE, SERVERTYPE, QUETYPE, JOBTYPE, SIZETYPE, INTRANGETYPE, FLOATRANGETYPE, DAYOFWEEKRANGETYPE, SERVERSETTYPE, QUESETTYPE, JOBSETTYPE, SIZESETTYPE, PARAMTYPE, FUNTYPE, SERVERSTATETYPE, QUESTATETYPE, JOBSTATETYPE, SIZESTATETYPE, DATETIMETYPE, CNODETYPE, VOIDTYPE, DATETIMERANGETYPE, CNODESETTYPE, SIZERANGETYPE, GENERICTYPE, KEYWORDTYPE.} and varFlag gives some indication on whether or not the matched token is indeed a constant, variable, or some other thing.

6.1.2.1.3. File: Lexer.c

LexerInit

```
void LexerInit(void)
```

Initializes the lexer by simply writing a simple startup message into the lexer stdout stream.

LexerTokenPut

```
void LexerTokenPut(char *lexem, int lin, int len, int typ, int varFlag)
```

This just fills up the MYTOK structure with values given in the parameter list.

LexerPrintToken

```
void LexerPrintToken(int linenum, char *yytext, int yylen)
```

if there's a lexer stdout stream, and the lexer debug flag is turned on, then the values to the given parameters are printed.

LexerPutDF

```
void LexerPutDF(int df)
```

Sets the lexer debug flag to `df`.

LexerCondPrint

```
void LexerCondPrint(char *str)
```

Prints `str` if there's a lexer stdout stream and the lexer debug flag is turned on.

LexerErr

```
void LexerErr(int e)
```

If there's a lexer stdout stream, print the message string associated with the error number given by `e`. This will issue an exit to the program.

6.1.2.2. Parser

The files involved with the Parser subsystem are Parser.b, Parser.h, and Parser.c. Parser.b is the *yacc specification* file describing valid "sentences" in the BASL grammar. Parser.h and Parser.c contain various data structures and functions that assist yacc in the parsing for correct grammar.

6.1.2.2.1. File: Parser.b

The following is the syntax definition for the Batch Scheduler Language (BASL) used by the PL (procedural language) job scheduler program. Some notes concerning the semantics analyzer and code generator are also added.

```
prog::= defs1 globAssign "sched_main" blockWithDefs

blockWithDefs::= '{' defs stats '}'
```

NOTE:

A variable's scope, which determines in what functions the variable has been defined/declared, is kept track via the ParserVarScope global variable. Whenever a function block is entered, ParserVarScope is incremented by 1.

```
block::= '{' stats '}'   /* no definition */

defs1::=  ''             /* empty */
        | defs1 defFun
        | defs1 def
```

NOTE:

Any global variable declared (ParserLevel == 0) will get a "static" keyword attached to it during BASL-to-C translation so that the variable will only be accessible from the local sched_main file, minimizing the chance of name collision when the translated code is compiled and linked with the PBS libraries.

```
globAssign::= ''         /* empty */
            | globAssign statAssign

defFun::= "Int" identifier        '(' params ')' blockWithDefs
        | "Float" identifier      '(' params ')' blockWithDefs
        | "Void" identifier       '(' params ')' blockWithDefs
        | "Dayofweek" identifier  '(' params ')' blockWithDefs
        | "DateTime" identifier   '(' params ')' blockWithDefs
        | "String" identifier     '(' params ')' blockWithDefs
        | "Size" identifier       '(' params ')' blockWithDefs
        | "Server" identifier     '(' params ')' blockWithDefs
        | "Que" identifier        '(' params ')' blockWithDefs
        | "Job" identifier        '(' params ')' blockWithDefs
        | "CNode" identifier      '(' params ')' blockWithDefs
        | "Set Server" identifier '(' params ')' blockWithDefs
        | "Set Que" identifier    '(' params ')' blockWithDefs
        | "Set Job" identifier    '(' params ')' blockWithDefs
        | "Set CNode" identifier  '(' params ')' blockWithDefs
```

NOTE: In the definition of functions, the identifier (function name) must be unique. On the generated code, identifier will be prefixed with "basl_" to avoid possible name conflict when the code is compiled and linked with other libraries like PBS.

```
params::=   ''   /* empty */
          | paramDeclare moreParams

paramDeclare::=    "Int" identifier
                 | "Float" identifier
                 | "Void" identifier
                 | "Dayofweek" identifier
                 | "DateTime" identifier
                 | "String" identifier
                 | "Size" identifier
                 | "Server" identifier
                 | "Que" identifier
                 | "Job" identifier
                 | "CNode" identifier
                 | "Set Server" identifier
                 | "Set Que" identifier
                 | "Set Job" identifier
                 | "Set CNode" identifier
                 | "Range Int" identifier
                 | "Range Float" identifier
                 | "Range Dayofweek" identifier
                 | "Range DateTime" identifier
                 | "Range Size" identifier
                 | "Fun Int" identifier
                 | "Fun Float" identifier
                 | "Fun Void" identifier
                 | "Fun Dayofweek" identifier
                 | "Fun DateTime" identifier
                 | "Fun String" identifier
                 | "Fun Size" identifier
                 | "Fun Server" identifier
                 | "Fun Que" identifier
                 | "Fun Job" identifier
                 | "Fun CNode" identifier
                 | "Fun Set Server" identifier
                 | "Fun Set Que" identifier
                 | "Fun Set Job" identifier
                 | "Fun Set CNode" identifier

moreParams::= ''
              | ',' paramDeclare moreParams
```

NOTE:
The identifiers that appear in the list of parameters must be unique.

```
defs::= ''    /* empty */
        | defs def

def::=    "Int" identifier ';'
        | "Float" identifier ';'
        | "Dayofweek" identifier ';'
        | "DateTime" identifier ';'
        | "String" identifier ';'
        | "Size" identifier ';'
        | "Server" identifier ';'
```

```
        | "Que" identifier ';'
        | "Job" identifier ';'
        | "CNode" identifier ';'
        | "Set Server" identifier ';'
        | "Set Que" identifier ';'
        | "Set Job" identifier ';'
        | "Set CNode" identifier ';'
        | "Range Int" identifier ';'
        | "Range Float" identifier ';'
        | "Range Dayofweek" identifier ';'
        | "Range DateTime" identifier ';'
        | "Range Size" identifier ';'
```

NOTE:

The identifier used must be unique; no other identifier in the same level of the same name must have been declared.

During code generation, the following translations occur:

```
        Int   identifier;             -> int identifier;
        Float identifier;             -> double identifier;
        Dayofweek identifier;         -> Dayofweek identifier;
        DateTime identifier;          -> DateTime identifier;
        String identifier;            -> char *identifier = NULLSTR;
        Size identifier;              -> Size identifier;
        Server identifier;            -> Server identifier = NOSERVER;
        Que identifier;               -> Que identifier = NOQUE;
        Job identifier;               -> Job identifier = NOJOB;
        CNode identifier;             -> CNode identifier = NOCNODE;
        Set Server identifier;        -> SetServer *identifier = EMPTYSETSERVER;
        Set Que identifier;           -> SetQue *identifier = EMPTYSETQUE;
        Set Job identifier;           -> struct SetJobElement *identifier = EMPTYSETJOB;
        Set CNode identifier;         -> SetCNode *identifier = EMPTYSETCNODE;
        Range Int identifier;         -> IntRange identifier;
        Range Float identifier;       -> FloatRange identifier;
        Range Dayofweek identifier;   -> DayofweekRange identifier;
        Range DateTime identifier;    -> DateTimeRange identifier;
        Range Size identifier;        -> SizeRange identifier;

stats::=   ''    /* empty */
        | stats stat

stat::= ';'     /* empty statement */
        | expr
        | statIf
        | statAssign
        | statPrint ';'
        | statFor
        | statSwitch
        | statForeach
        | statWhile
        | statContinue ';'
        | statBreak ';'
        | statREturn
        | statExit
```

```
        | block

statAssign::= identifier eqs expr ';'
```

NOTE:

1. Compatible assignment types:

identifier	expression
Int	Int, Float
Float	Int, Float
Dayofweek	Dayofweek
DateTime	DateTime
String	String
Size	Size
Que	Que
Job	Job
CNode	CNode
Server	Server
Range Dayofweek	Range Dayofweek
Range DateTime	Range DateTime
Range Size	Range Size
Server	Server
Que	Que
Job	Job
CNode	CNode
Range Int	Range Int
Range Float	Range Float

2. If identifier is of string type, generate code to modify the identifier's scope to reflect actual scope, and also generate code that will free up any temporarily allocated strings during the assignment operation.

3. If identifier is of Que type, generate code that will modify the scope of the identifier only if it is not already of global scope (mallocTableSafeModScope). Then free up any temporarily allocated Que structures (those with scope of -1).

```
statWhile:: = "while" '(' expr ')' block
```

NOTE:

1. expr's type must be INTTYPE or FLOATTYPE.
2. A global variable called inLoop exists to keep track on whether or not the parser is currently inside a loop construct. As the parser is inside the while loop, the inLoop counter is incremented. Leaving the loop will cause the in-Loop counter to be decremented.

```
statForeach::= "foreach" '(' identifier "in" identifier ')' block
```

NOTE:

1. The valid types of identifiers are:

1st identifier	2nd identifier
Server	Set Server

Que	Set Que
Job	Set Job
CNode	Set CNode

```
expr::=   expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '%' expr
        | '-' expr
        | '+' expr
        | exprTerms EQ exprTerms
        | exprTerms NEQ exprTerms
        | exprTerms LT exprTerms
        | exprTerms LE exprTerms
        | exprTerms GT exprTerms
        | exprTerms GE exprTerms
        | expr AND expr
        | expr OR expr
        | '!' expr
        | identifier postop
        | '(' expr ')'
        | exprTerms
```

1. Consistent types for +:

left expression type	right expression type
STRINGTYPE	STRINGTYPE
SIZETYPE	SIZETYPE
INTTYPE or FLOATTYPE	INTTYPE or FLOATTYPE

2. Consistent types for -, *, /:

left expression type	right expression type
SIZETYPE	SIZETYPE
INTTYPE or FLOATTYPE	INTTYPE or FLOATTYPE

3. Consistent types for modulus (%):

left expression type	right expression type
INTTYPE	INTTYPE

4. Consistent types for unary minus (-) and unary plus (+):

expression type
INTTYPE
FLOATTYPE
SIZETYPE

5. Consistent types for EQ, NEQ, LT, LE, GT, GE:

left expression type	right expression type
Dayofweek	Dayofweek
DateTime	DateTime
String	String
Size	Size
Server	Server
Que	Que
Job	Job
CNode	CNode
Server	Server
Que	Que
Set Job	Set Job
Set CNode	Set CNode
Int, Float	Int, Float

6. Consistent types for AND, OR:

left expression type	right expression type
INTTYPE or FLOATTYPE	INTTYPE or FLOATTYPE

7. Consistent types for !expr:

expression type
INTTYPE
FLOATTYPE

8. Consistent types for post operators ++ and --:

expression type
INTTYPE
FLOATTYPE

9. After every expression, if the expr's type is String, then code to free up temporarily allocated strings (scope -1) is generated. Same with Que type.

```
exprTerms::=   consts
         | identifier '(' args ')'
         | identifier
```

NOTE:

1. For function calls, "identifier(args)", identifier must have been previously declared. Also, the arguments' types must match the proto types except in special functions like the following:
   a. Job QueJobFind(Que que, Fun <ReturnType> function(Job job), {EQ, NEQ, GE, GT, LE, LT}, <ReturnType> value)
   b. Job QueJobFind(Que que, Fun <ReturnType> function(Job job), {MAX, MIN})
   c. Que QueFilter(Que que, Fun <ReturnType> function(Job job), {EQ,NEQ,GE,GT,LE,LT}, <ReturnType> value)
   d. Que QueFilter(Que que, Fun Job function(Job job), {MAX, MIN})

   For the above functions, CodeGenBuffSaveQueJobFind() and CodeGen-

BuffSaveQueFilter() are called to generate code. In forms (a) and (c), the <ReturnType> of arguments 2 and 4 must match, and the code generated depends on the <ReturnType>.

2. Functions are classified into 2 category: internally-defined (or built-ins), and externally-defined (as defined by the scheduler writer). If identifier refers to a function that has been externally defined, then a "basl_" prefix is added to the function name during code generation so as to avoid name collision when the code is linked with system, PBS libraries.
3. Any identifier used must have been previously declared.

```
statPrint::=   "print" '(' identifier ')'
             | "print" '(' consts ')'
```

1. Allowed types for identifier, consts:

TYPE
Int
Float
Dayofweek
String
Size
Que
Job
CNode
Server
Range Int
Range Float
Range Dayofweek
Range DateTime
Range Size

```
statFor::= "for" '(' statForAssign ';' identifier cprOp expr ';'
           statForAssign ')' block
```

NOTE:

1. Upon entering the for loop, inLoop counter is incremented; it is decremented upon exit.

```
statForAssign:   identifier eqs expr
               | identifier postop
```

NOTE:

1. In the first form, identifier and expr must be of type INTTYPE or FLOATTYPE.
2. In the second form, identifier must be of type INTTYPE or FLOATTYPE.

```
statIf::=   "if" '(' expr ')' block
      | "if" '(' expr ')' block "else" block
```

NOTE: expr's return type is either INT or FLOAT.

```
statReturn::=   "return" '(' identifier ')'
          | "return" '(' consts ')'
          | "return" '(' ')'
```

NOTE:

1. The type returned by identifier and consts must match the calling function's return type.
2. if the identifier is of string type, then generate code that will modify its scope to be -1 (become a temporary string) so that the malloc-ed storage for it does not get cleared up yet upon return. It will get freed on the next call to varstrFree() to free up temporary strings, which takes place after an expression involving the function call is executed.
3. if the identifier is of Que type, then generate code that will modify its scope to be -1 (a temporary queue) only if its current scope is not global (mallocTableSafeModScope). This is so that malloc-ed storage for the Que identifier does not get cleared up yet. It will get freed on the next call to mallocTableFreeByScope() of temporary queue variables, which takes place after an expression involving the function call is executed.
4. The third form "return()" is allowed if the return type of the enclosing function is of type Void.

```
statSwitch::= "switch" '(' identifier ')' '{' caseList defCase '}'
```

NOTE:

1. The identifier's type cannot be of type Void.
2. inSwitch variable keeps track of whether or not the parser is currently inside a switch construct. CurrSwitchVar holds the switch variable token. So when the parser enters the switch statement, ParserLevel and inSwitch variables are incremented, and CurrSwitchVar is set appropriately. Decrement ParserLevel, inSwitch upon leaving.

```
caseList::= ''
          | caseList caseElement

caseElement::=    "case" intConst        ':' block
                | "case" floatConst      ':' block
                | "case" dayofweekConst  ':' block
                | "case" datetimeConst   ':' block
                | "case" sizeConst       ':' block
                | "case" stringConst     ':' block
                | "case" serverConst     ':' block
                | "case" queConst        ':' block
                | "case" jobConst        ':' block
                | "case" cnodeConst      ':' block
                | "case in" constRange   ':' block
                | "case in" identifier   ':' block

defCase::=   ''
           | 'default' ':' block
```

NOTE:

1. The identifier's type in statSwitch must match the type given in caseElement.
2. The case labels are not allowed to be duplicated.
3. The allowed constRange and identifier types in "case in" depends on the identifier's type on the "switch" statement as follows:

switch identifier type	constRange, identifier type
SERVERTYPE	SERVERSETTYPE
QUETYPE	QUESETTYPE
JOBTYPE	JOBSETTYPE
CNODETYPE	CNODESETTYPE
INTTYPE	INTRANGETYPE
FLOATTYPE	FLOATRANGETYPE
DAYOFWEEKTYPE	DAYOFWEEKRANGETYPE
DATETIMETYPE	DATETIMERANGETYPE
SIZETYPE	SIZERANGETYPE

```
statContinue::= "continue" ';'
```
NOTE: The continue statement must have been invoked within a loop. This is done by checking to see if inLoop counter is > 0.

```
statBreak::= "break" ';'
```
NOTE: The break statement must have been invoked within a loop. This is done by checking to see if inLoop counter is > 0.

```
statExit::= "exit" '(' intConst ')' ';'
```
NOTE: Before the "exit" statement is generated, generate code also that will free up storage allocated at the current variable scope.

```
consts::=   intConst
          | floatConst
          | dayofweekConst
          | datetimeConst
          | stringConst
          | sizeConst
          | cprOp
          | constRange
          | 'MAX'
          | 'MIN'
          | serverConst
          | queConst
          | jobConst
          | cnodeConst
          | setServerConst
          | setQueConst
          | setJobConst
          | setCnodeConst
          | stringConst

constRange:   intConstRange
            | floatConstRange
            | dayofweekConstRange
            | datetimeConstRange
            | sizeConstRange

intConst::= [+-]?[0-9]+
            | "SUCCESS"
            | "FAIL"
            | "SERVER_ACTIVE"
            | "SERVER_IDLE"
```

```
            | "SERVER_SCHED"
            | "SERVER_TERM"
            | "SERVER_TERMDELAY"
            | "QTYPE_E"
            | "QTYPE_R"
            | "SCHED_DISABLED"
            | "SCHED_ENABLED"
            | "FALSE"
            | "TRUE"
            | "TRANSIT"
            | "QUEUED"
            | "HELD"
            | "WAITING"
            | "RUNNING"
            | "EXITING"
            | "CNODE_OFFLINE"
            | "CNODE_DOWN"
            | "CNODE_FREE"
            | "CNODE_RESERVE"
            | "CNODE_INUSE_EXCLUSIVE"
            | "CNODE_INUSE_TIMESHARED"
            | "CNODE_TIMESHARED"
            | "CNODE_CLUSTER"
            | "CNODE_UNKNOWN"
            | "SYNCRUN"
            | "ASYNCRUN"
            | "DELETE"
            | "RERUN"
            | "HOLD"
            | "RELEASE"
            | "SIGNAL"
            | "MODIFYATTR"
            | "MODIFYRES"
            | "OP_EQ"
            | "OP_NEQ"
            | "OP_LT"
            | "OP_LE"
            | "OP_GE"
            | "OP_GT"
            | "OP_MAX"
            | "OP_MIN"

floatConst::= [+-]?[0-9]+[.][0-9]*

dayofweekConst::=   "SUN"
                  | "MON"
                  | "TUE"
                  | "WED"
                  | "THU"
                  | "FRI"
                  | "SAT"

datetimeConst::=   '(' intConst '|' intConst '|' intConst ')'
                 | '(' intConst ':' intConst ':' intConst ')'
```

```
                | '(' intConst '|' intConst '|' intConst '@' intConst ':' intConst ':' in

serverConst::=   NOSERVER

queConst::=  NOQUE

jobConst::=  NOJOB

cnodeConst::=    NOCNODE

setServerConst::= EMPTYSETSERVER

setQueConst::= EMPTYSETQUE

setJobConst::= EMPTYSETJOB

setCnodeConst::= EMPTYSETCNODE

stringConst::= NULLSTR
```

NOTE:

1. A time constant (hh:mm:ss) must satisfy the condition:
   0 <= hh <= 23, 0 <= mm <= 59, 0 <= ss <= 61.
2. A date constant (mon | day | year) must satisfy the condition:
   1 <= mon <= 12, 1 <= day <= 31, 0 <= year.
3. A time/date constant (mon | day | year@hh:mm:ss) must satisfy 1) and 2).

```
stringConst::= ["][a-zA-Z0-9 0*["]

sizeConst::= [+-]?[0-9]+[kmgtpKMGTP]?[bwBW]

intConstRange::= '(' intConst ',' intConst ')'
```

NOTE: For an int constant range, the 1st part must be <= 2nd part.

```
floatConstRange::= '(' floatConst ',' floatConst ')'
```

NOTE: For a float constant range, the 1st part must be <= 2nd part.

```
dayofweekConstRange::= '(' dayofweekConst ',' dayofweekConst ')'
```

NOTE: For a dayofweek constant range, the 1st part must be <= 2nd part.

```
datetimeConstRange::= '(' datetimeConst ',' datetimeConst ')'
```

NOTE: For a datetime constant range, if both the 1st part and 2nd part are the full date/time construct, then 1st part <= 2nd part.

```
sizeConstRange::= '(' sizeConst ',' sizeConst ')'
```

NOTE: The 1st part must be <= 2nd part.

```
cprOp::=  "LE"
        | "LT"
        | "GE"
        | "GT"
        | "EQ"
        | "NEQ"

args::=    ''
        | arg argList
```

```
argList::=   ''
        | ',' arg argList

arg::=    identifier
        | consts

eqs::=    '='
```

NOTE:

When '{' has been encountered, ParserLevel variable is incremented. When '}' is encountered, if the block contains any kind of String type variable, then code is generated to call varstrFreeByScope to free up strings that have been malloc-ed at that block (scope is determined via the variable ParserVarScope). Also, if the block that the parser is in is sched_main()'s, then code for varstrFreeByScope(-1) is generated to free up all temporary malloc-ed strings. Also if the block contains declarations for Que type variables, then code to free up malloc-ed storage for the temporary Que structures is generated.

Code to free up malloc-ed storage (varstrFree(), mallocTableFree(), varstrModScope(), mallocTableSafeModScope()) are generated appropriately before a return statement, exit statement, or level 1 right curly bracket.
"generated appropriately" means that if any string has been declared at level 1 in the enclosing function, then varstrFree() code will be generated along with the appropriate variable scope to free; similarly, mallocTableFree() code will be generated if any Que type entity has been declared at level 1 in the enclosing function.

6.1.2.2.2. File: Parser.c

yyerror

```
void yyerror(char *ep)
```

A slightly modified version of the yacc's yyerror() call where addition message about linenum is printed out to the parser's stdout stream.

ParserInit

```
void ParserInit(void)
```

Initializes variables ParserLevel (keeps track of program nestings) and ParserVarScope (keeps track of variable's readability within a BASL program).

ParserPrintToken

```
void ParserPrintToken(char *lexeme, int lin, int len, int typ)
```

If there's a parser stdout stream and the parser debug flag is turned on, then print out the values of the given parameters.

ParserPutDF

```
void ParserPutDF(int df)
```

Set the parser debug flag to the given `df` value.

ParserLevelIncr

```
void ParserLevelIncr(void)
```

Increments the ParserLevel variable, and sends a message to parser stdout stream saying that it has done so.

ParserLevelDecr

```
void ParserLevelDecr(void)
```

Decrement the ParserLevel variable, and sends a message to parser stdout stream saying that it has done so.

ParserLevelGet

```
int ParserLevelGet(void)
```

Returns the value to the ParserLevel variable.

ParserVarScopeIncr

```
void ParserVarScopeIncr(void)
```

Increments the ParserVarScope variable, and sends a message to parser stdout stream saying that it has done so.

ParserVarScopeGet

```
int ParserVarScopeGet(void)
```

Returns the value to the ParserVarScope variable.

ParserCondPrint

```
void ParserCondPrint(char *str)
```

Prints `str` to parser stdout stream (if any) and if the parser debug flag is turned on.

ParserErr

```
void ParserErr(int e)
```

Prints the message string associated with error `e` to parser stdout stream (if any).

ParserCurrFunPtrPut

```
void ParserCurrFunPtrPut(Np np)
```

Makes `np` be the current pointer to a node containing a function token.

ParserCurrFunPtrGet

```
Np ParserCurrFunPtrGet(void)
```

Returns the current pointer to the node containing a function token.

ParserCurrFunParamPtrPut

```
void ParserCurrFunParamPtrPut(Np np)
```

Makes `np` be the current pointer to a node that is holding a function parameter token.

ParserCurrFunParamPtrGet

```
Np ParserCurrFunParamPtrGet(void)
```

Returns the current pointer to a node that is holding a function parameter token.

ParserCurrSwitchVarPut

```
void ParserCurrSwitchVarPut(struct MYTOK token)
```

Makes `token` be the switch variable.

ParserCurrSwitchVarGet

```
struct MYTOK ParserCurrSwitchVarGet(void)
```

Returns the current switch variable token.

6.1.2.3. Symbol Table

The symbol table contains data structures that are accessed by the Lexer, Parser, Semantic analyzer, and code generator in order to perform various tasks leading to the translation of BASL code into C. The symbol table stores the variable and structure names, labels, and all other names used in the program. The files involved with the symbol table are Node.h, Node.c, List.h, List.c, SymTabGlob.h, SymTab.h, and SymTab.c.

6.1.2.3.1. File: Node.h

The main data structure manipulated by the routines in this file is the Node class shown in the following:

```
struct FUNDESCR
{
       int paramCnt;
       struct Node *paramPtr;
};

struct Node
{
       char lexeme[LEXEMSZ];
       int  type;            /* Semantic type */
       int  lineDef;
       int  level;
       int  funFlag;
       struct FUNDESCR funDescr;
       struct Node *rptr;
}
typedef struct Node *np;
```

6.1.2.3.2. File: Node.c

NodeNew

```
Np NodeNew(char *lexem, int typ, int lin, int leve, int funFla)
```

Args:

lexem	A match token string.
type	Internal data type of lexem.
lin	The line number in the program input file where the lexem was found.

leve The nesting level of the matched token.

funFla The flag that says whether or not the lexem is part of a function definition.

Create a new Node structure by malloc-ing its storage, and then fill it with values given by the function arguments. A pointer to this Node structure is returned.

NodeInit

```
void NodeInit(Np nx, char *lexem, int typ, int lin, int leve, int funFla)
```

Args:

nx A pointer to an existing node structure.

lexem A match token string.

type Internal data type of lexem.

lin The line number in the program input file when the lexem was found.

leve The nesting level of the matched token.

funFla The flag that says whether or not the lexem is part of a function definition.

Fill an existing Node structure with values given by the function arguments.

NodePrint

```
void NodePrint(Np nx)
```

Args:

nx A pointer to an existing node structure.

Print out to the Node stdout stream (if any) the values of the given Node structure.

NodeFunDescrPrint

```
void NodeFunDescrPrint(Np nx)
```

Args:

nx A pointer to an existing node structure.

Just print the function description values for the given Node structure.

NodeFunDescrFindByLexeme

```
Np NodeFunDescrFindByLexeme(Np nx, char *lexem)
```

Args:

nx A pointer to an existing node structure.

lexem The token to match.

Return the pointer to the node that is a parameter pointer structure, whose lexeme value is the one given in this function's argument.

NodeCmp

```
int NodeCmp(Np nx, char *lexem)
```

Args:

nx A pointer to an existing node structure.

lexem The token to match.

Compares the Node nx's lexem with that of the argument lexem (given). Returns 1 if the former is > latter, -1 if the former is < latter, 0 if they are the same.

NodeErr

```
void NodeErr(int e)
```

Args:

e error number.

Prints the error message associated with error number e.

NodeCondPrint

```
void NodeCondPrint(char *str)
```

Args:

str The string message to print out.

If Node debug flag is on and if there's a Node stdout stream, then print the given message.

NodePutDF

```
void NodePutDF(int df)
```

Args:

df The new debug flag value.

Updates the Node debug flag value to df.

NodeGetLexeme

```
char *NodeGetLexeme(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the lexeme attribute value of nxp.

NodeGetType

```
int NodeGetType(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the type attribute value of nxp.

NodeGetLineDef

```
int NodeGetLineDef(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the lineDef attribute value of nxp.

NodeGetLevel

```
int NodeGetLevel(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the level attribute value of nxp.

NodeGetFunFlag

```
int NodeGetFunFlag(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the funFlag attribute value of nxp.

NodeGetParamPtr

```
Np NodeGetParamPtr(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns pointer to the Node nxp's 1st parameter.

NodeGetLexeme

```
char *NodeGetLexeme(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the lexeme attribute value of nxp.

NodeGetType

```
int NodeGetType(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the type attribute value of nxp.

NodeGetLineDef

```
int NodeGetLineDef(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the lineDef attribute value of nxp.

NodeGetLevel

```
int NodeGetLevel(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the level attribute value of nxp.

NodeGetFunFlag

```
int NodeGetFunFlag(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns the funFlag attribute value of nxp.

NodeGetParamPtr

```
Np NodeGetParamPtr(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Returns pointer to the Node nxp's 1st parameter.

NodePutLexeme

```
void NodePutLexeme(Np nxp, char *lexem)
```

Args:

nxp A pointer to a Node structure.

lexem new lexem value.

Makes lexem be the new nxp lexeme value.

NodePutType

```
void NodePutType(Np nxp, int type)
```

Args:

nxp A pointer to a Node structure.

type The new type for the Node's lexeme.

Replaces the type attribute value of nxp to the given argument.

NodePutLineDef

```
void NodePutLineDef(Np nxp, int lin)
```

Args:

nxp A pointer to a Node structure.

lin The new line for the Node's lexeme.

Replaces the lineDef attribute value to the given argument.

NodePutLevel

```
void NodePutLevel(Np nxp, int level)
```

Args:

nxp A pointer to a Node structure.

level The new level for the Node's lexeme.

Replaces the level attribute value of nxp to the given argument.

NodePutFunFlag

```
void NodePutFunFlag(Np nxp, int funFla)
```

Args:

nxp A pointer to a Node structure.

funFla The new function flag for the Node's lexeme.

Replaces the funFlag attribute value of nxp to the given argument.

NodePutParamPtr

```
void NodePutParamPtr(Np nxp, Np paramPtr)
```

Args:

nxp A pointer to a Node structure.

paramPtr
A pointer to a Node's paramPtr structure.

Replaces the pointer value to the Node nxp's 1st parameter to the given argument.

NodePutParamCnt

```
void NodePutParamCnt(Np nxp, int paramCnt)
```

Args:

nxp A pointer to a Node structure.

paramCnt
# of parameters to the function node.

Replaces the paramCnt attribute of Node nxp to the given argument.

NodeParamCntIncr

```
void NodeParamCntIncr(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Increments the paramCnt attribute of Node nxp.

NodeParamCntDecr

```
void NodeParamCntDecr(Np nxp)
```

Args:

nxp A pointer to a Node structure.

Decrements the paramCnt attribute of Node nxp.

6.1.2.3.3. File: List.c

ListPutDF

```
void ListPutDF(int df)
```

Args:

df the new debug flag value.

Set the List debug flag value to df.

ListCondPrint

```
void ListCondPrint(char *str)
```

Args:

str The message to print out.

If List debug flag is on, and there's a List stdout stream, then print out the message str.

ListIsEmpty

```
int ListIsEmpty(List L)
```

Args:

L A pointer to the head of the list of Nodes.

Returns 1 if the L is NULL; 0 otherwise.

ListPrint

```
void ListPrint(List L)
```

Args:

L A pointer to the head of the list of Nodes.

Prints every member of a list of Nodes.

ListInsertFront

```
List ListInsertFront(List L, Np nxp)
```

Args:

L A pointer to the head of the list of Nodes.

nxp A new node to insert into the list.

Insert the Node nxp in front of L, and returns nxp.

ListParamLink

```
void ListParamLink(Np funNp, Np parNp)
```

Args:

funNp A pointer to the head of the list of Nodes.

parNp A new node to insert into the list.

Inserts parNp at the end of function Node funNp's parameter list.

ListInsertSortedN

```
List ListInsertSortedN(List L, Np nxp)
```

Args:

L A pointer to the head of the list of Nodes.

nxp A new node to insert into the list.

Insert the Node nxp into the List L in a way that the increasing lexicographical ordering of lexemes is maintained.

ListInsertSortedD

```
List ListInsertSortedD(List L, char *lexem, int typ, int lineDe, int leve, int funFla)
```

Args:

L A list of Nodes.

lexem A lexeme for the new Node.

type A type for the new Node.

lineDe The line # where the lexeme was found.

level The nesting level of the Node.

funFla The function flag of the Node.

Creates a new Node with the given values, and insert the Node into the list in a sorted manner. Returns the pointer to the new List.

ListIsMember

```
int ListIsMember(List L, Np nxp)
```

Args:

L A list of Nodes.

nxp A Node pointer.

Returns 1 if nxp is a member of the List of Nodes L; 0, otherwise.

ListGetLast

```
Np ListGetLast(List L)
```

Args:

L A list of Nodes.

Returns the last node in the List of Nodes L.

ListGetSucc

```
Np ListGetSucc(List L, Np nxp)
```

Args:

L A list of Nodes.

nxp A Node pointer.

Returns the next Node element after nxp.

ListDeleteNode

```
List ListDeleteNode(List L, Np nxp)
```

Args:

L A list of Nodes.

nxp A Node pointer.

Deletes the node pointed by nxp, and free up any malloc-ed storage for it.

ListDelete

```
List ListDelete(List L)
```

Args:

L A list of Nodes.

Deletes the entire List of Nodes L. Malloc-ed areas are freed.

ListDeleteLevel

```
List ListDeleteLevel(List L, int leve)
```

Args:

L A list of Nodes.

leve A Node level.

Deletes all Nodes in List L that have a level leve. Return a new List L.

ListFindNodeByLexeme

```
Np ListFindNodeByLexeme(List L, char *lexeme)
```

Args:

L A list of Nodes.

lexeme A lexeme to look for.

Returns the Node in the List of Nodes that contains the given lexeme.

ListFindNodeByLexemeInLevel

```
Np ListFindNodeByLexemeInLevel(List L, char *lexeme, int leve)
```

Args:

L A list of Nodes.

lexeme A lexeme to look for

leve A level to look for.

Returns the Node in the List of Nodes that contains the given lexeme and level.

ListFindNodeByLexemeInLine

```
Np ListFindNodeByLexemeInLine(List L, char *lexeme, int line)
```

Args:

L	A list of Nodes.
lexeme	A lexeme to look for
line	A line to look for.

Returns the Node in the List of Nodes that contains the given lexeme and line.

ListMatchNodeByLexemeInLine

```
Np ListMatchNodeByLexemeInLine(List L, char *lexeme, int line)
```

Args:

L	A list of Nodes.
lexeme	A lexeme to look for
line	A line to look for.

Returns the Node in the List of Nodes that match the given lexeme and line.

ListFindNodeBeforeLexemeInLine

```
Np ListFindNodeBeforeLexemeInLine(List L, char *lexeme, int line)
```

Args:

L	A list of Nodes.
lexeme	A lexeme to look for
line	A line to look for.

Returns the Node in the List of Nodes that is before the node containing the given lexeme and line. If the node found that contain the lexeme and line does not have a previous node (head of List), then that node itself is returned.

ListMatchNodeBeforeLexemeInLine

```
Np ListMatchNodeBeforeLexemeInLine(List L, char *lexeme, int line)
```

Args:

L	A list of Nodes.
lexeme	A lexeme to look for
line	A line to look for.

Returns the Node in the List of Nodes that is before the node matching the given lexeme and line. If the node found that match the lexeme and line does not have a previous node (head of List), then that node itself is returned.

ListFindNodeByLexemeAndTypeInLevel

```
Np ListFindNodeByLexemeAndTypeInLevel(List L, char *lexeme, int leve,
                                      int line, int (*compare_func)())
```

Args:

- L A list of Nodes.
- lexeme A lexeme to look for
- level A level to look for.
- line A line to look for.
- compare_func
  The compare function to use when comparing lexemes.

Returns the Node in the List of Nodes that contains lexeme, leve, and line. compare_func is used to determine whether or not the Node contains lexeme.

ListFindAnyNodeInLevelOfType

```
Np ListFindAnyNodeInLevelOfType(List L, int leve, int type)
```

Args:

- L A list of Nodes.
- leve A level to look for.
- type A type to look for.

Returns any Node in the List of Nodes that contains leve with 'type'.

ListErr

```
void ListErr(int e)
```

Args:

- e error number.

Prints out the message associated with error number e.

6.1.2.3.4. File: SymTab.c

The symbol table is nothing more but a linked List.

SymTabInit

```
void SymTabInit(void)
```

Calls SymTabKeywordsInit() to initialize the symbol table.

SymTabPutDF

```
void SymTabPutDF(int df)
```

Args:

df new debug flag value.

Sets the Symbol table debug flag to the value of the given argument.

SymTabCondPrint

```
void SymTabCondPrint(char *str)
```

Args:

str The message string to print out.

Prints out to Symbol table output stream (if any) the message 'str' only if the debug flag is set.

SymTabIsEmpty

```
int SymTabIsEmpty(void)
```

Returns 1 if the symbol table is empty (head of the list is NULL).

SymTabPrint

```
int SymTabPrint(void)
```

Prints to stdout stream the entire symbol table.

SymTabInsertFront

```
void SymTabInsertFront(STEP nxp)
```

Args:

nxp The node to insert in front of the symbol table.

Puts the node pointed to by 'nxp' at the head of the symbol table.

SymTabParamLink

```
void SymTabParamLink(STEP funNp, Np parNp)
```

Args:

- funNp Pointer to the function node.
- parNp Pointer to a parameter node.

Inserts the node parNp at the end of function node funNp's parameter list.

SymTabInsertSortedN

```
void SymTabInsertSortedN(STEP nxp)
```

Args:

- nxp A pointer to the node to be inserted into the symbol table.

Inserts the node nxp into the symbol table, maintaining the lexicographical ordering of the nodes' lexemes.

SymTabInsertSortedD

```
void SymTabInsertSortedD(char *lexem, int typ, int lineDe, int leve, int funFla)
```

Args:

- lexem A node lexeme value.
- typ A lexeme's type.
- lineDe A lexeme's lineDef value.
- level A lexeme's level value.
- funFla A lexeme's function flag value.

Creates a new node with values (lexem, typ, lineDe, leve, funFla), and this new node is inserted into the symbol table in a manner in which the lexicographical ordering of the nodes' lexemes is maintained.

SymTabIsMember

```
int SymTabIsMember(STEP nxp)
```

Args:

- nxp A pointer to a node.

Returns 1 if node 'nxp' is one of the nodes in the symbol table; 0 otherwise.

SymTabGetLast

```
STEP SymTabGetLast(void)
```

Returns the last element of the symbol table.

SymTabGetSucc

```
STEP SymTabGetSucc(STEP nxp)
```

Args:

nxp A pointer to a node.

Returns the node that comes after 'nxp' in the symbol table.

SymTabDeleteNode

```
void SymTabDeleteNode(STEP nxp)
```

Args:

nxp A pointer to a node.

Removes the node 'nxp' from the symbol table.

SymTabDelete

```
void SymTabDelete(void)
```

Removes all the node elements from the symbol table.

SymTabFindFunProtoByLexemeInProg

```
STEP SymTabFindFunProtoByLexemeInProg(char *lexeme)
```

Args:

lexeme Lexeme to search for.

Returns the function node that contains 'lexeme'.

SymTabFindNodeByLexemeInProg

```
STEP SymTabFindNodeByLexemeInProg(char *lexeme)
```

Args:

lexeme Lexeme to search for.

Returns the node that contains 'lexeme'.

SymTabFindNodeByLexemeInLevel

```
STEP SymTabFindNodeByLexemeInLevel(char *lexeme, int level)
```

Args:

lexeme Lexeme to search for.

level Level value to look for.

Returns the node that contains 'lexeme' and 'level'.

SymTabFindNodeByLexemeAndTypeInLevel

```
STEP SymTabFindNodeByLexemeAndTypeInLevel(char *lexeme, int level,
                                  int type, int (*compare_func)())
```

Args:

lexeme Lexeme to search for.

level Level value to look for.

type Type value to search for.

compare_func
Compare function to use when comparing the given 'lexeme' with the lexemes on the symbol table.

Returns the node that contains 'lexeme' (compared via 'compare_func'), 'level', and 'type'.

SymTabFindAnyNodeInLevelOfType

```
STEP SymTabFindAnyNodeInLevelOfType(int level, int type)
```

Args:

level Level value to look for.

type Type value to search for.

Returns any node that in 'level' that is of 'type'.

SymTabDeleteLevel

```
STEP SymTabDeleteLevel(int leve)
```

Args:

leve Level value whose nodes will be deleted.

Remove nodes whose level is 'leve'.

SymTabKeyWordsInit

```
void SymTabKeyWordsInit(void)
```

Initializes the symbol table by creating an "endmarker" node.

SymTabGetOrigin

```
STEP SymTabGetOrigin(void)
```

Returns the first node in the symbol table.

SymTabErr

```
void SymTabErr(int e)
```

Args:

e An error number.

Prints out to symbol table stdout stream the message string associated with error 'e'.

6.1.2.4. Semantic Analyzer

The files involved with the Semantic analyzer subsystem are Semantic.h, and Semantic.c. The semantics analyzer is responsible for checking to make sure that tokens are used together in a consistent and valid way.

6.1.2.4.1. File: Semantic.c

SemanticInit

```
void SemanticInit(void)
```

Initializes any internal variables used by the semantics analyzer.

SemanticPutDF

```
void SemanticPutDF(int df)
```

Args:

df The new debug flag value

Sets the semantics debug flag to the 'df' value.

SemanticCondPrint

```
void SemanticCondPrint(char *str)
```

Args:

str The message to print out.

Prints to semantics stdout (if any) the 'str' message only if the semantics debug flag is on.

SemanticErr

```
void SemanticErr(int e)
```

Args:

e Error number.

Prints the message associated with error number 'e'.

SemanticStatAssignCk

```
void SemanticStatAssignCk(struct MYTOK var, struct MYTOK expr)
```

Args:

var An identifier token to be assigned a new value.

expr An expression token whose value will be assigned to 'var'.

So this semantically checks the BASL grammar: var = expr. See grammar specification for consistent types for 'var' and 'expr'.

SemanticPlusExprCk

```
void SemanticStatPlusExprCk(struct MYTOK left_expr, struct MYTOK right_expr)
```

Args:

left_expr The left expression in a '+' arithmetic expression.

right_expr The right expression in a '+' arithmetic expression.

This semantically checks the BASL grammar: left_expr + right_expr. Allowed types to be added are String, Size, Int, and Float. See grammar specification for mutually consistent types for 'left_expr' and 'right_expr'.

SemanticMinusExprCk

```
void SemanticStatMinusExprCk(struct MYTOK left_expr, struct MYTOK right_expr)
```

Args:

left_expr The left expression in a '-' arithmetic expression.

right_expr The right expression in a '-' arithmetic expression.

This semantically checks the BASL grammar: left_expr - right_expr. Allowed types to be subtracted are Size, Int, and Float. See grammar specification for mutually consistent types for 'left_expr' and 'right_expr'.

SemanticStatMultDivExprCk

```
void SemanticStatMultDivExprCk(struct MYTOK left_expr, struct MYTOK right_expr)
```

Args:

left_expr The left expression in a '*', '/' arithmetic expression.

right_expr The right expression in a '*', '/' arithmetic expression.

This semantically checks the BASL grammar: left_expr * right_expr or left_expr / right_expr. Allowed types to be multiplied or divided are Size, Int, and Float. See grammar specification for mutually consistent types for 'left_expr' and 'right_expr'.

SemanticStatModulusExprCk

```
void SemanticStatModulusExprCk(struct MYTOK left_expr, struct MYTOK right_expr)
```

Args:

left_expr The left expression in a '%' arithmetic expression.

right_expr The right expression in a '%' arithmetic expression.

This semantically checks the BASL grammar: left_expr % right_expr. Allowed type to be remaindered is Int. See grammar specification for mutually consistent types for 'left_expr' and 'right_expr'.

SemanticStatCompExprCk

```
void SemanticStatCompExprCk(struct MYTOK left_expr, struct MYTOK right_expr)
```

Args:

left_expr The left expression in a logical expression involving EQ, NEQ, LE, LT, GE, GT.

right_expr The right expression in a logical expression involving EQ, NEQ, LE, LT, GE, GT.

This semantically checks the BASL grammar: left_expr <logical_op> right_expr. Allowed types to be logically operated are: Int, Float, Dayofweek, DateTime, String, Size, Server, Que, Job, CNode, Set Server, Set Que, Set Job, Set CNode. See grammar specification for mutually consistent types for 'left_expr' and 'right_expr'.

SemanticStatAndOrExprCk

```
void SemanticStatAndOrExprCk(struct MYTOK left_expr, struct MYTOK right_expr)
```

Args:

left_expr The left expression in a logical expression involving AND, OR.

right_expr The right expression in a logical expression involving AND, OR.

This semantically checks the BASL grammar: left_expr <AND|OR> right_expr. Allowed types to be AND/OR-ed are: Int, Float. See grammar specification for mutually consistent types for 'left_expr' and 'right_expr'.

SemanticStatNotExprCk

```
void SemanticStatNotExprCk(struct MYTOK expr)
```

Args:

expr The expression in a unary expression: !expr.

This semantically checks the BASL grammar: !expr. Allowed types to the ! operator are: Int, Float.

SemanticStatPostOpExprCk

```
void SemanticStatPostOpExprCk(struct MYTOK expr)
```

Args:

expr The expression in a unary expression: expr++, expr--.

This semantically checks the BASL grammar: expr++, expr--. Allowed types to the ++, -- operators are: Int, Float.

SemanticStatUnaryExprCk

```
void SemanticStatUnaryExprCk(struct MYTOK expr)
```

Args:

expr The expression in a unary expression: +expr, -expr.

This semantically checks the BASL grammar: +expr, -expr. Allowed types to the unary operators are: Int, Float, Size.

SemanticStatPrintTailCk

```
void SemanticStatPrintTailCk(struct MYTOK expr)
```

Args:

expr The expression in a return statement: return(expr).

This semantically checks the BASL grammar: return(expr). Allowed types to the return expr are: Int, Float, Dayofweek, DateTime, String, Size, Que, Job, CNode, Server, Range Int, Range Float, Range Dayofweek, Range DateTime, Range Size.

SemanticStatWhileHeadCk

```
void SemanticStatWhileHeadCk(struct MYTOK expr)
```

Args:

expr The expression in a while statement.

This semantically checks the BASL grammar: while(expr) { ... }. Allowed types to the expr are: Int, Float.

SemanticStatIfHeadCk

```
void SemanticStatIfHeadCk(struct MYTOK expr)
```

Args:

expr The expression in an if statement.

This semantically checks the BASL grammar: if(expr) { ... }. Allowed types to the expr are: Int, Float.

SemanticStatReturnTailCk

```
void SemanticStatReturnTailCk(struct MYTOK expr)
```

Args:

expr The return expression to check.

This semantically checks the BASL grammar: return(expr). 'expr' must match the enclosing function's return type. Allowed expr types are: Dayofweek, DateTime, String, Size, Server, Que, Job, CNode, Int, Float.

This semantically checks the BASL grammar: if(expr) { ... }. Allowed types to the expr are: Int, Float.

SemanticVarDefCk

```
void SemanticVarDefCk(struct MYTOK var)
```

Args:

var The variable to check.

This semantically checks the variable definition construct in a BASL grammar. 'var' must have not been previously declared.

SemanticForHeadCk

```
void SemanticForHeadCk(struct MYTOK exp6, struct MYTOK exp8)
```

Args:

exp6 An expression to semantically check.

exp8 An expression to semantically check.

This semantically checks the for head construct in a BASL grammar: for(statForAssign; exp6 cprOp exp8; statForAssign) ... exp6 and exp8 must have types that are either Int or Float.

SemanticForAssignCk

```
void SemanticForAssignCk(struct MYTOK exp1, struct MYTOK exp2)
```

Args:

exp1 An expression to semantically check.

exp2 An expression to semantically check.

This semantically checks the statForAssign statement found in a for head construct: expr1 = expr2. exp1 and exp2 must have types that are either Int or Float.

SemanticForPostAssignCk

```
void SemanticForAssignCk(struct MYTOK exp)
```

Args:

exp An expression to semantically check.

This semantically checks the post-operated assignment statement that can be found in a for head construct in a BASL grammar. exp must have type that is either Int or Float.

SemanticForeachHeadCk

```
void SemanticForeachHeadCk(struct MYTOK val1, struct MYTOK val2)
```

Args:

val1 The first identifier in a foreach statement.

val2 The second identifier in a foreach statement.

This semantically checks to make sure that val1 is one of {Server, Que, Job, CNode} and that they match up 1:1 with val2 values {Set Server, Set Que, Set Job, Set CNode}.

SemanticParamVarCk

```
int SemanticParamVarCk(struct MYTOK val)
```

Args:

val The parameter variable to check.

This semantically checks a particular parameter that appears in a function call. If the parameter value matches with its prototype type, then that type is returned. Predefined functions (type: YES_INT) and user-defined functions appearing (type: YES) as a parameter are considered the same.

SemanticParamConstsCk

```
int SemanticParamConstsCk(struct MYTOK val)
```

Args:

val The parameter variable to check.

This semantically checks a particular parameter that appears in a function call. If the parameter constant type matches with its prototype type, then that type is returned.

SemanticCaseInVarCk

```
void SemanticCaseInVarCk(struct MYTOK var)
```

Args:

var The parameter variable to check.

This semantically checks the type of the var appearing in a "case in var" switch body. Var must have one of the following types: Range Int, Range Float, Range Dayofweek, Range DateTime, Range Size, Set Server, Set Que, Set Job, Set CNode.

SemanticCaseTypeCk

```
void SemanticCaseTypeCk(struct MYTOK val)
```

Args:

val The case value to check.

This semantically checks the type of the case value (label) against the switch variable. Given "switch(var) { case val: ... }", the var type must be the same as the val type.

SemanticCaseInTypeCk

```
void SemanticCaseInTypeCk(struct MYTOK val)
```

Args:

val	The case value to check.

This semantically checks the type of the "case in" value (label) against the switch variable. Given "switch(var) { case in val: ... }", val can be one of {Set Server, Set Que, Set Job, Set CNode, Range Int, Range Float, Range Dayofweek, Range DateTime, Range Size} and must match 1:1 with one of the following switch variable's types: {Server, Que, Job, CNode, Int, Float, Dayofweek, DateTime, Size}.

SemanticTimeConstCk

```
void SemanticTimeConstCk(struct MYTOK h, struct MYTOK m, struct MYTOK s)
```

Args:

h	The hour entity in a time constant string.
m	The minute entity in a time constant string.
s	The seconds entity in a time constant string.

Given a time constant string, (h:m:s), this function checks to make sure that 0 <= h <= 23, 0 <= m <= 59, 0 <= s <= 61.

SemanticDateConstCk

```
void SemanticDateConstCk(struct MYTOK m, struct MYTOK d, struct MYTOK y)
```

Args:

m	The month entity in a date constant string.
d	The day entity in a date constant string.
y	The year entity in a date constant string.

Given a date constant string, (m|d|y), this function checks to make sure that 1 <= m <= 12, 1 <= d <= 31, 0 <= y.

SemanticIntConstRangeCk

```
void SemanticIntConstRangeCk(struct MYTOK lo, struct MYTOK hi)
```

Args:

lo The low value (left) in a Range Int constant.

hi The high value (right) in a Range Int const.

Given a Range Int constant, (low, high), this function checks to make sure that low <= high.

SemanticFloatConstRangeCk

```
void SemanticFloatConstRangeCk(struct MYTOK lo, struct MYTOK hi)
```

Args:

lo The low value (left) in a Range Float constant.

hi The high value (right) in a Range Float const.

Given a Range Float constant, (low, high), this function checks to make sure that low <= high.

SemanticDayofweekConstRangeCk

```
void SemanticDayofweekConstRangeCk(struct MYTOK lo, struct MYTOK hi)
```

Args:

lo The low value (left) in a Range Dayofweek constant.

hi The high value (right) in a Range Dayofweek constant.

Given a Range Dayofweek constant, (low, high), this function checks to make sure that low <= high.

SemanticDateTimeConstRangeCk

```
void SemanticDateTimeConstRangeCk(struct MYTOK lo, struct MYTOK hi)
```

Args:

lo The low value (left) in a Range DateTime constant.

hi The high value (right) in a Range DateTime constant.

Given a Range DateTime constant, (low, high), this function checks to make sure that low <= high if at least one of the values contain a date portion. If both low and high contain time portions only, then they will go through a different algorithm in some other function such as automatically filling in the missing time or date portions.

SemanticSizeConstRangeCk

```
void SemanticSizeConstRangeCk(struct MYTOK lo, struct MYTOK hi)
```

Args:

lo The low value (left) in a Range Size constant.

hi The high value (right) in a Range Size constant.

Given a Range Size constant, (low, high), this function checks to make sure that low <= high.

6.1.2.5. Code Generator

The files involved with the code generator subsystem are CodeGen.h, and CodeGen.c. The code generator is responsible for translating BASL statements into C statements. The code generator maintains 2 data structures, a stack (St) and a list (CodeGenBuff) containing the various translated C tokens. The stack is used to properly evaluate postfix arithmetic and logical expressions, and the CodeGenBuff holds the resulting C statements.

6.1.2.5.1. File: CodeGen.c

CodeGenStackNew

```
St CodeGenStackNew(Np np)
```

Args:

np The node to be placed as the first element of the stack.

Initializes a Stack data type (St) used by the code generator, giving it a node 'np' value. The top of the new stack is returned.

CodeGenStackPush

```
void CodeGenStackPush(Np np)
```

Args:

np A Node pointer to put into the stack.

Adds a new element to the stack St and this element contains 'np'.

CodeGenStackPop

```
void CodeGenStackPop(void)
```

This returns the element that is at the top of the stack, and removes that element of the stack. The stack element's malloc-ed storage is freed; the node that it contains is not freed since it maybe part of another list that contains the free routine.

CodeGenStackClear

```
void CodeGenStackClear(void)
```

This removes all the elements from the stack.

CodeGenStackPrint

```
void CodeGenStackPrint(void)
```

This prints all the elements on the stack.

CodeGenInit

```
void CodeGenInit(void)
```

This initializes all internal variables accessed by the code generator.

CodeGenPutDF

```
void CodeGenPutDF(int df)
```

Args:

df The new debug flag.

Set the code generator debug flag to 'df'.

CodeGenCondPrint

```
void CodeGenCondPrint(char *str)
```

Args:

str The new string to print out.

Prints out the message 'str' to the code generator stdout stream (if any) if the code generator debug flag is on.

CodeGenPrint

```
void CodeGenPrint(void)
```

Prints some information about the code generator like the descriptors to the generator buffer.

CodeGenErr

```
void CodeGenErr(int e)
```

Args:

e An error number.

Prints the message associated with error 'e'.

CodeGenBuffClear

```
void CodeGenBuffClear(void)
```

Removes all the elements in CodeGenBuff.

CodeGenBuffPrint

```
void CodeGenBuffPrint(void)
```

Prints out the contents of the CodeGenBuff list.

CodeGenBuffEmit

```
void CodeGenBuffEmit(void)
```

Prints out the lexemes in CodeGenBuff into the C output file stream. Necessary amount of indentation is added for blocks of statements. After printing the lexemes, the CodeGenBuff is flushed (cleared).

CodeGenBuffSwitchEmit

```
void CodeGenBuffSwitchEmit(void)
```

Same as CodeGenBuffEmit() except the amount of indentations in the output is 1 less.

CodeGenLastDef

```
void CodeGenLastDef(char *lexeme)
```

Args:

lexeme The lexeme to match.

Returns the last instance (maximum lineDef value) of 'lexeme' on the CodeGenBuff table. Lexemes that contain "(" or ")" are matched with any lexemes containing "(" or ")" no matter what the leading or trailing characters are. For example, if the lexeme is "str(", then all entries in the CodeGenBuff that contain the left and right parenthesis will match.

CodeGenBuffGetNp

```
void CodeGenBuffGetNp(char *lexeme, int lineDef)
```

Args:

lexeme The lexeme to match.

lineDef The lineDef value to match.

Returns the pointer to the node containing 'lexeme' with 'lineDef'.

matchPairs

```
static void matchPairs(char *leftsym, int rightsym)
```

Args:

leftsym The left pair symbol.

lineDef The right pair symbol.

Go through the elements of CodeGenBuff, and match 'leftsym' with 'rightsym'. For 'leftsym' that matches, push the corresponding node onto the code generator stack (St); for 'rightsym' that matches, remove node that is at the top of the stack (presumably this is the match), and update the rightsym's lineDef value in the CodeGenBuff buffer to that of the matching 'left-sym'. So matching leftsym and rightsym will have the same unique lineDef value.

CodeGenBuffSaveFirst

```
void CodeGenBuffSaveFirst(char *str)
```

Args:

str The string to insert into CodeGenBuff.

Inserts 'str' into CodeGenBuff making it the first entry. If 'str' contains a '(' or a ')', then the CodeGenBuff entries are modified so that all matching leftsym and rightsym have the same lineDef value.

CodeGenBuffSave

```
void CodeGenBuffSave(char *str)
```

Args:

str The string to insert into CodeGenBuff.

Appends 'str' to CodeGenBuff making it the last entry. If 'str' contains a '(' or a ')', then the CodeGenBuff entries are modified so that all matching leftsym and rightsym have the same lineDef value.

CodeGenBuffSaveBefore

```
void CodeGenBuffSaveBefore(char *str, char *lexeme, int inst)
```

Args:

str	The string to insert into CodeGenBuff.
lexeme	The succeeding lexeme.
inst	The lexeme's lineDef value.

Insert 'str' before the node containing 'lexeme' with 'inst' lineDef value in CodeGenBuff. If 'str' contains a '(' or a ')', then the CodeGenBuff entries are modified so that all matching left-sym and rightsym have the same lineDef value.

CodeGenBuffSaveAfter

```
void CodeGenBuffSaveAfter(char *str, char *lexeme, int inst)
```

Args:

str	The string to insert into CodeGenBuff.
lexeme	The preceding lexeme.
inst	The lexeme's lineDef value.

Insert 'str' after the node containing 'lexeme' with 'inst' lineDef value in CodeGenBuff. If 'str' contains a '(' or a ')', then the CodeGenBuff entries are modified so that all matching leftsym and rightsym have the same lineDef value.

CodeGenBuffDelete

```
void CodeGenBuffDelete(char *lexeme, int inst)
```

Args:

lexeme	A lexeme value.
inst	The lexeme's lineDef value.

Deletes the node in CodeGenBuff containing 'lexeme' with 'inst' lineDef value.

CodeGenBuffSaveFunFirst

```
void CodeGenBuffSaveFunFirst(char *str)
```

Args:

str	The string to insert into CodeGenBuff.

Same as CodeGenBuffSaveFirst() except the node containing 'str' will have a function flag indicator on it.

CodeGenBuffSaveFun

```
void CodeGenBuffSaveFun(char *str)
```

Args:

str	The string to insert into CodeGenBuff.

Same as CodeGenBuffSave() except the node containing 'str' will have a function flag indicator on it.

CodeGenBuffSaveFunBefore

```
void CodeGenBuffSaveFunBefore(char *str, char *lexeme, int inst)
```

Args:

str	The string to insert into CodeGenBuff.
lexeme	The succeeding lexeme.
inst	The lexeme's lineDef value.

Same as CodeGenBuffSaveBefore() except the node containing 'str' will have a function flag indicator on it.

CodeGenBuffSaveFunAfter

```
void CodeGenBuffSaveFunAfter(char *str, char *lexeme, int inst)
```

Args:

str	The string to insert into CodeGenBuff.
lexeme	The preceeding lexeme.
inst	The lexeme's lineDef value.

Same as CodeGenBuffSaveAfter() except the node containing 'str' will have a function flag indicator on it.

CodeGenStatPrint

```
void CodeGenStatPrint(void)
```

Adds a "printf" to CodeGenBuff.

CodeGenStatPrintTail

```
void CodeGenStatPrintTail(struct MYTOK expr)
```

Args:

expr Some expression entity.

Generates the code for 'print(expr)' BASL statement in CodeGenBuff. The corresponding "printf()" statement will be generated based on the data type of 'expr'.

CodeGenBuffGetLast

```
Np CodeGenBuffGetLast(void)
```

Return the last node in CodeGenBuff.

CodeGenBuffSaveSpecOper

```
void CodeGenBuffSaveSpecOper(operstr)
```

Args:

operstr An special operator string: could be sizeAdd(, sizeMult(, sizeMinus(, sizeDiv(.

This checks to make sure ++, and -- operators are preceded by an identifier expression. Also, this routine makes sure that the special operator string is placed in the right place in a heavily nested arithmetic expression.

CodeGenBuffSaveStrAssign

```
void CodeGenBuffSaveStrAssign(void)
```

Given a BASL assignment statement, var = expr, with var being of String type, then generate the C statement "dynamic_strcpy(&var, expr);".

CodeGenBuffSaveForeach

```
void CodeGenBuffSaveForeach(struct MYTOK var, struct MYTOK svar)
```

Args:

var The 1st identifier in the foreach statement.

svar The 2nd identifier in the foreach statement.

Given a BASL assignment statement, foreach(var in svar) {}, the following C translations occur:

```
 foreach(server in set_server) {}  -->
  for(server=set_server->head; server; server=server->nextptr) {}

 foreach(cnode in set_cnode) {}  -->
  for(cnode=set_cnode->head; cnode; cnode=cnode->nextptr) {}
```

foreach(que in set_que) {} -->
for(que=set_que->head; que; que=que->nextptr) {}

foreach(job in set_job) {} -->
for(firstJobPtr(&set_job, set_job->first); (job=set_job->job); nextJobPtr(&set_job)) {}

CodeGenBuffSaveSwitch

```
void CodeGenBuffSaveSwitch(struct MYTOK switchVar)
```

Args:

switchVar
The variable at the head of the switch statement.

Given a BASL assignment statement, switch(switchVar) { case caseVal: ... }, BASL-to-C translations occur resulting in "if(switchVar == caseVal) {}" or
"else if(switchVar == caseVal) {}" statements.

CodeGenBuffSaveSwitchIn

```
void CodeGenBuffSaveSwitchIn(struct MYTOK switchVar, struct MYTOK caseVal)
```

Args:

switchVar
The variable at the head of the switch statement.

caseVal A value appearing in a case label of a switch statement.

Given a BASL assignment statement, switch(switchVar) { case in caseVal: ... }, BASL-to-C translations occur resulting in "if(inXRange(switchVar, caseVal)) {}" or "else if(in-XRange(switchVar, caseVal)) {}" statements.

CodeGenBuffSaveQueJobFind

```
void CodeGenBuffSaveQueJobFind(void)
```

The following translations occur:

QueJobFind(que, arg2(), cpr, arg4) --> QueJobFindInt(que, arg2(), cpr, arg4)
if arg2's type = arg4's type == Int
QueJobFind(que, arg2(), cpr) --> QueJobFindInt(que, arg2(), cpr)
if arg2's type == Int

QueJobFind(que, arg2(), cpr, arg4) --> QueJobFindStr(que, arg2(), cpr, arg4)
if arg2's type = arg4's type == String
QueJobFind(que, arg2(), cpr) --> QueJobFindStr(que, arg2(), cpr)
if arg2's type == String

```
QueJobFind(que, arg2(), cpr, arg4) --> QueJobFindSize(que, arg2(), cpr, arg4)
                    if arg2's type = arg4's type == Size
QueJobFind(que, arg2(), cpr)     --> QueJobFindSize(que, arg2(), cpr)
                    if arg2's type == Size

QueJobFind(que, arg2(), cpr, arg4)
                  --> QueJobFindDateTime(que, arg2(), cpr, arg4)
                  if arg2's type = arg4's type == DateTime
QueJobFind(que, arg2(), cpr)  --> QueJobFindSize(que, arg2(), cpr)
                  if arg2's type == DateTime
```

CodeGenBuffSaveQueFilter

```
void CodeGenBuffSaveQueFilter(void)
```

The following translations occur:

```
QueFilter(que, arg2(), cpr, arg4) --> QueFilterInt(que, arg2(), cpr, arg4)
                    if arg2's type = arg4's type == Int
QueFilter(que, arg2(), cpr)     --> QueFilterInt(que, arg2(), cpr)
                    if arg2's type == Int

QueFilter(que, arg2(), cpr, arg4) --> QueFilterStr(que, arg2(), cpr, arg4)
                    if arg2's type = arg4's type == String
QueFilter(que, arg2(), cpr)     --> QueFilterStr(que, arg2(), cpr)
                    if arg2's type == String

QueFilter(que, arg2(), cpr, arg4) --> QueFilterSize(que, arg2(), cpr, arg4)
                    if arg2's type = arg4's type == Size
QueFilter(que, arg2(), cpr)     --> QueFilterSize(que, arg2(), cpr)
                    if arg2's type == Size

QueFilter(que, arg2(), cpr, arg4)
                  --> QueFilterDateTime(que, arg2(), cpr, arg4)
                  if arg2's type = arg4's type == DateTime
QueFilter(que, arg2(), cpr)  --> QueFilterSize(que, arg2(), cpr)
                  if arg2's type == DateTime
```

CodeGenBuffSaveSort

```
void CodeGenBuffSaveSort(void)
```

The following translations occur:

```
Sort(Set Job s, Int arg2(), Int arg3) --> SetJobSortInt(s, arg2(), arg3)
Sort(Set Job s, String arg2(), Int arg3) --> SetJobSortStr(s, arg2(), arg3)
Sort(Set Job s, Size arg2(), Int arg3) --> SetJobSortSize(s, arg2(), arg3)
Sort(Set Job s, DateTime arg2(), Int arg3) --> SetJobSortDateTime(s, arg2(), arg3)
```

Sort(Set Job s, Float arg2(), Int arg3) --> SetJobSortFloat(s, arg2(), arg3)

Sort(Set CNode s, Int arg2(), Int arg3) --> SetCNodeSortInt(s, arg2(), arg3)
Sort(Set CNode s, String arg2(), Int arg3) --> SetCNodeSortStr(s, arg2(), arg3)
Sort(Set CNode s, Size arg2(), Int arg3) --> SetCNodeSortSize(s, arg2(), arg3)
Sort(Set CNode s, DateTime arg2(), Int arg3) --> SetCNodeSortDateTime(s, arg2(), arg3)
Sort(Set CNode s, Float arg2(), Int arg3) --> SetCNodeSortFloat(s, arg2(), arg3)

Sort(Set Que s, Int arg2(), Int arg3) --> SetQueSortInt(s, arg2(), arg3)
Sort(Set Que s, String arg2(), Int arg3) --> SetQueSortStr(s, arg2(), arg3)
Sort(Set Que s, Size arg2(), Int arg3) --> SetQueSortSize(s, arg2(), arg3)
Sort(Set Que s, DateTime arg2(), Int arg3) --> SetQueSortDateTime(s, arg2(), arg3)
Sort(Set Que s, Float arg2(), Int arg3) --> SetQueSortFloat(s, arg2(), arg3)

Sort(Set Server s, Int arg2(), Int arg3) --> SetServerSortInt(s, arg2(), arg3)
Sort(Set Server s, String arg2(), Int arg3) --> SetServerSortStr(s, arg2(), arg3)
Sort(Set Server s, Size arg2(), Int arg3) --> SetServerSortSize(s, arg2(), arg3)
Sort(Set Server s, DateTime arg2(), Int arg3) --> SetServerSortDateTime(s, arg2(), arg3)
Sort(Set Server s, Float arg2(), Int arg3) --> SetServerSortFloat(s, arg2(), arg3)

6.1.3. Pseudo-Compiler

The source code for the pseudo-compiler front end *basl2c* is Basl2c.c. The compiler will take a program in BASL and translate it into intermediate language (C code). The compiler will check the structure and semantic correctness of the BASL program before generating the intermediate code.

6.1.3.1. File: Basl2c.c

loadUserAccessibleAssistFuncs

```
static void loadUserAccessibleAssistFuncs(void)
```

This is where predefined functions in BASL are loaded in order for the scheduler writer to call them. This is the function to modify in case new functions are added or when deleting functions from the list.

addIncludes

```
static void addIncludes(void)
```

This function defines the "#include" lines to be placed at the header of the resulting intermediate (C) code.

addMainSched

```
static void addMainSched(void)
```

Attaches some calls like SystemInit(), SystemStateRead() to the resulting intermediate code in order for it to function as a daemon scheduler.


```
main(int argc, char **argv)
```

The sequence of execution are: (1) get command line arguments (see BASL ERS for format of options), (2) initialize internal variables used by the Lexer, Parser, Symbol table, Semantic analyzer, and Code Generator, (3) generate the #include lines, (4) load the predefined functions' prototypes, (5) start parsing the input file. At the end of the parsing stage, then (6) delete the Symbol table, (7) close any opened output stream.

6.1.4. Assist Functions

The assist (helper) functions are made available to create scheduling constructs Job, Que, CNode, Server, and ResMom. The functions can be found under the *Assist* subdirectory.

6.1.4.1. General Purpose Functions

The source code found under the *Gen* subdirectory contains general-purpose data structures and functions that are used by the Lexer, Parser, Semantic analyzer, Code generator, and the predefined functions. The files involved are af.h and af.c. The main data structures used are:

```
struct time_struct {
   int h;
   int m;
   int s;
}
typedef struct time_struct Time;

struct date_struct {
   int m;
   int d;
   int y;
}
typedef struct date_struct Date;

struct datetime_struct {
   Time t;
   Date d;
}
typedef struct datetime_struct DateTime;

struct size_struct {
   long int num;       /* numeric part */
   unsigned int shift; /* K=10, M=20, G=30, T=40, P=50 */
   unsigned int units; /* BYTES=0, WORD=1 */
}
typedef struct size_struct Size;

struct intRange_struct {
    int lo;
    int hi;
```

```
}
typedef struct intRange_struct IntRange;

struct floatRange_struct {
     float lo;
     float hi;
}
typedef struct floatRange_struct FloatRange;

struct dayofweekRange_struct {
     float lo;
     float hi;
}
typedef struct dayofweekRange_struct DayofweekRange;

struct datetimeRange_struct {
     DateTime lo;
     DateTime hi;
}
typedef datetimeRange_struct DateTimeRange;

struct sizeRange_struct {
     Size  lo;
     Size  hi;
}
typedef sizeRange_struct SizeRange;

struct IntRes {
     struct IntRes *nextptr;
     char   *name;
     int    value;
}

struct SizeRes {
     struct SizeRes *nextptr;
     char   *name;
     Size   value;
}

struct StringRes {
     struct StringRes *nextptr;
     char   *name;
     char   *value;
}

struct dynamic_array {
     void   *ptr;    /* pointer to the dynamic array */
     int    numElems; /* # of elements in the array */
}

struct varstr_type {
     int   scope;    /* variable's scope */
     void  *pptr;    /* variable's parent ptr -
                        used to collectively free
```

```
                        up malloc-ed storage
                        linked to some main structure */
     void  *ptr;     /* ptr to malloc-ed storage of a */
                     /* variable string */
}
#define VARSTRLEN 500
static struct varstr_type *varstr[VARSTRLEN];

struct varstrIndex_type {
        struct varstr_type      *mptr;
        struct varstrIndex_type *link;
};

#define VARSTR_INDEX_LEN 480
static struct varstrIndex_type *varstrIndex[VARSTR_INDEX_LEN];

struct varstrSubIndex_type {
        struct varstrIndex_type         *ptr;
        struct varstrSubIndex_type      *link;
};

#define VARSTR_SUBINDEX_LEN 19
static struct varstrSubIndex_type *varstrSubIndex[VARSTR_SUBINDEX_LEN];

struct malloc_type {
     int   scope;    /* variable's scope */
     void  *pptr;    /* variable's parent ptr -
                        used to collectively free
                        up malloc-ed storage
                        linked to some main structure */
     void  *ptr;     /* ptr to malloc-ed storage */
}
#define MALLOCLEN 500
static struct malloc_type *mallocTable[MALLOCLEN];

struct mallocIndex_type {
        struct malloc_type      *mptr;
        struct mallocIndex_type *link;
};

#define MALLOC_INDEX_LEN 480
static struct mallocIndex_type *mallocIndexTable[MALLOC_INDEX_LEN];

struct mallocSubIndex_type {
        struct mallocIndex_type         *ptr;
        struct mallocSubIndex_type      *link;
};

#define MALLOC_SUBINDEX_LEN 19
static struct mallocSubIndex_type *mallocSubIndexTable[MALLOC_SUBINDEX_LEN];
```

dynamic_array is a table of dynamically allocated arrays. varstr hash table that holds information about malloc-ed strings, hashed according to ptr value. varstrIndex is a hash index table for the varstr table, hashed on the pptr attribute. varstrSubIndex is another hash index table for the varstr table, hashed on the scope attribute. mallocTable hash

table that holds information about malloc-ed non-string objects, hashed against the ptr value. `mallocIndexTable` another hash table for the mallocTable, this time, hashed against the pptr attribute. `mallocSubIndexTable` another hash table for the mallocTable, this time, hashed against the scope attribute. `IntRes, SizeRes, StringRes` hold various resource names and values that were obtained from a query of the Server.

6.1.4.1.1. File: af.c

varstrHash

```
static int varstrHash(unsigned long k)
```

Args:

k A hash key

Uses the multiplication method found in p. 228 of Cormen, Leiserson, Rivest book, "Introduction to Algorithms", to come up with a hashing function.

varstrIndexHash

```
static int varstrIndexHash(unsigned long k)
```

Args:

k A hash key

Uses the multiplication method found in p. 228 of Cormen, Leiserson, Rivest book, "Introduction to Algorithms", to come up with a hashing function.

varstrSubIndexHash

```
static int varstrSubIndexHash(unsigned long k)
```

Args:

k A hash key

Uses the multiplication method found in p. 228 of Cormen, Leiserson, Rivest book, "Introduction to Algorithms", to come up with a hashing function.

mallocTableHash

```
static int mallocTableHash(unsigned long k)
```

Args:

k A hash key

Uses the multiplication method found in p. 228 of Cormen, Leiserson, Rivest book, "Introduction to Algorithms", to come up with a hashing function.

mallocIndexTableHash

```
static int mallocIndexTableHash(unsigned long k)
```

Args:

k A hash key

Uses the multiplication method found in p. 228 of Cormen, Leiserson, Rivest book, "Introduction to Algorithms", to come up with a hashing function.

mallocSubIndexTableHash

```
static int mallocSubIndexTableHash(unsigned long k)
```

Args:

k A hash key

Uses the multiplication method found in p. 228 of Cormen, Leiserson, Rivest book, "Introduction to Algorithms", to come up with a hashing function.

varstrSubIndexAdd

```
void varstrSubIndexAdd(void *ptr)
```

Args:

ptr Pointer to varstrIndex_type

Adds a new entry into the varstrSubIndex table, hashed according to the ptr->mptr->scope value.

varstrSubIndexFree

```
void varstrSubIndexFree(void *ptr)
```

Args:

ptr Pointer to varstrIndex_type

Frees the varstrSubIndex table entry that carries a 'ptr' value.

varstrIndexAdd

```
void varstrIndexAdd(void *ptr)
```

Args:

ptr Pointer to varstr_type

Adds a new entry into the varstrIndex table, hashed according to the ptr->pptr value.

varstrIndexFree

```
void varstrIndexFree(void *ptr)
```

Args:

ptr Pointer to varstr_type

Frees the varstrIndex table entry that carries a 'ptr' value. Also, frees up any varstr-SubIndex table entry that hangs off of this entry.

varstrIndexFreeNoIndex

```
void varstrIndexFreeNoIndex(void *ptr)
```

Args:

ptr Pointer to varstr_type

Like varstrIndexFree except only the varstrSubIndex table entry that hangs off of this entry is freed.

varstrIndexFreeNoSubIndex

```
void varstrIndexFreeNoSubIndex(void *ptr)
```

Args:

ptr Pointer to varstr_type

Like varstrIndexFree except only the varstrIndex table entry that carries a 'ptr' value is freed, not any varstrSubIndex entry that hangs off it.

varstrAdd

```
void varstrAdd(void *ptr, int scope, void *pptr)
```

Args:

ptr Pointer to malloc-ed string.

scope Scope value of ptr.

pptr Some parent pointer in which 'ptr' is somewhat related to.

Adds a new entry into the varstr table, hashed according to its 'ptr' value.

varstrRemove

```
void varstrRemove(void *ptr)
```

Args:

ptr Pointer to malloc-ed string.

Frees up the slot occupied by 'ptr' in the varstr table, as well as any varstrIndex table entry that hangs off it.

varstrModScope

```
void varstrModScope(void *ptr, int scope)
```

Args:

ptr Pointer to malloc-ed string.

scope The ptr's new scope.

Modifies a ptr's scope in the varstr table to the given value. The corresponding varstr-SubIndex table entry is updated as well since that is hashed according to scope value.

varstrModPptr

```
void varstrModPptr(void *ptr, void *newpptr)
```

Args:

ptr Pointer to malloc-ed string.

newpptr ptr's parent pointer.

Modifies a ptr's pptr value in the varstr table to the given value. The corresponding varstrIndex table entry is updated as well since that is hashed according to parent ptr value.

inVarstr

```
int inVarstr(void *ptr)
```

Args:

ptr Pointer to malloc-ed string.

Returns 1 if 'ptr' is in the varstr table; 0 otherwise.

varstrPrint

```
void varstrPrint(void)
```

Prints out the elements of the varstr table.

varstrFree

```
void varstrFree(void *ptr)
```

Args:

ptr Pointer to malloc-ed string.

Issues a free() to the malloc-ed storage allocate to 'ptr', and clearing any varstrIndex, varstrSubIndex table entries associated with it.

varstrFreeNoIndex

```
void varstrFreeNoIndex(void *ptr)
```

Args:

ptr Pointer to malloc-ed string.

Issues a free() to the varstr entry that carries 'ptr', but do not free any varstrIndex table entry associated with it.

varstrFreeNoSubIndex

```
void varstrFreeNoSubIndex(void *ptr)
```

Args:

ptr Pointer to malloc-ed string.

Issues a free() to the varstr entry that carries 'ptr', but do not free any varstrSubIndex table entry associated with it.

varstrPrint

```
void varstrPrint(void)
```

Prints out the elements of the varstr, varstrIndex, and varstrSubIndex tables.

varstr2Free

```
void varstr2Free(void *ptr, void *ptr2)
```

Args:

ptr Pointer to malloc-ed string.

ptr2 Pointer to another malloc-ed string.

An optimization attempt that allows 2 pointers to be freed all at once.

varstrFreeByScope

```
void varstrFreeByScope(int scope)
```

Args:

scope The scope of string(s) to free up.

Frees up all malloc-ed strings whose scope value is as given.

varstrFreeByPptr

```
void varstrFreeByPptr(void *pptr)
```

Args:

pptr The parent pointer of the string(s) to free up.

Frees up all malloc-ed strings whose pptr value is as given.

varstrInit

```
void varstrInit(void)
```

initializes the hash tables: varstr, varstrIndex, varstrSubIndex.

dynamic_strcpy

```
void dynamic_strcpy(char **str1_ptr, const char *str2)
```

Args:

str1_ptr Pointer to the string to copy a new value into.

str2 New value of the string.

Copies value of 'str2' to the string pointed to by str1_ptr. If the latter is NULL, then a new string is malloc-ed; otherwise, it will be realloc-ed. The newly alloc-ed or realloc-ed storage is recorded in the varstr table. On realloc, any previous alloc-ed storage recorded in the varstr table is removed. The varstr scope of the string is automatically global (0).

dynamic_strcat

```
void dynamic_strcat(char **str1_ptr, const char *str2)
```

Args:

str1_ptr Pointer to the string to copy a new value into.

str2 value of a string to append.

Appends value of 'str2' to the string pointed to by str1_ptr. If the latter is NULL, then a new string is malloc-ed; otherwise, it will be realloc-ed. The newly alloc-ed or realloc-ed storage is recorded in the varstr table. On realloc, any previous alloc-ed storage recorded in the varstr table is removed. The varstr scope of the string is automatically global (0).

strToInt

```
int strToInt(char *str)
```

Args:

str String to convert.

Converts a 'str' into an int value using the strtol() call. The int value is returned.

strToFloat

```
Float strToFloat(char *str)
```

Args:

str String to convert.

Converts a 'str' into a float value using the strtod() call. The float value is returned.

strToDayofweek

```
Dayofweek strToDayofweek(char *str)
```

Args:

str String to convert.

Converts a 'str' into a Dayofweek value, mapping "SUN"->SUN, "MON"->MON, etc... The resulting value is returned.

strToDate

```
Date strToDate(char *str)
```

Args:

str String to convert.

Converts a 'str' of the form: (m|d|y@h:m:s), or (m|d|y) into a Date value. {0, 0, 0} will be returned if error was encountered during conversion.

strToTime

```
Time strToTime(char *str)
```

Args:

str String to convert.

Converts a 'str' of the form: (m|d|y@h:m:s), or (h:m:s) into a Time value. {-1, -1, -1} will be returned if error was encountered during conversion.

strToDateTime

```
DateTime strToDateTime(char *str)
```

Args:

str String to convert.

Converts a 'str' of the form: (m|d|y@h:m:s), (h:m:s), or (m|d|y) into a DateTime value. {-1, -1, -1, 0, 0, 0} will be returned if error was encountered during conversion.

strToSize

```
Size strToSize(char *str)
```

Args:

str String to convert.

Converts a 'str' of the form: "<numeric><suffix>" where <suffix> is: [k|K|m|M|g|G|t|T|p|P] [b|w] into the corresponding Size struct. If an error occurred during conversion such as BADVAL or BADSUFFIX, then {-1, 0, BYTES} Size struct is returned. NOTE: The algorithm used in the conversion is the same as the routines found in the server.

strsecsToDateTime

```
DateTime strsecsToDateTime(char *str)
```

Args:

val String to convert.

Converts a 'str', containing some # of seconds since epoch, into a DateTime value. {-1, -1, -1, 0, 0, 0} will be returned if error was encountered during conversion.

strToBool

```
Size strToBool(char *str)
```

Args:

str String to convert.

Converts a 'str', of the form: "True" or "False", to the int TRUE or FALSE.

strToSize

```
Size strToSize(char *str)
```

Args:

str String to convert.

Converts a 'str' of the form: "<numeric><suffix>" where <suffix> is: [k|K|m|M|g|G|t|T|p|P] [b|w] into the corresponding Size struct. If an error occurred during conversion such as BADVAL or BADSUFFIX, then {-1, 0, BYTES} Size struct is returned. NOTE: The algorithm used in the conversion is the same as the routines found in

sizeToStr

```
void sizeToStr(Size sizeval, char *cvnbuf)
```

Args:

sizeval The Size value to convert.

cvnbuf The converted String.

Converts a 'sizeval' into a string, placing the result in 'cvnbuf'.

strtimeToSecs

```
int strtimeToSecs(times)
```

Args:

times A time string of the form (hh:mm:ss[.ms]).

This returns the equivalent # of seconds for a given 'times' string. 'ms' can potentially be lost. If conversion fails, this will return a -1.

datecmp

```
int datecmp(Date d1, Date d2)
```

Args:

d1 1st Date value to compare.

d2 2nd Date value to compare.

Returns: < 0 if d1 is < d2; = 0 if d1 = d2; > 0 if d1 >= d2.

timecmp

```
int timecmp(Time t1, Time t2)
```

Args:

- t1 1st Time value to compare.
- t2 2nd Time value to compare.

Returns: < 0 if t1 is < t2; = 0 if t1 = t2; > 0 if t1 >= t2.

datetimecmp

```
int datetimecmp(DateTime dt1, DateTime dt2)
```

Args:

- dt1 1st DateTime value to compare.
- dt2 2nd DateTime value to compare.

Returns: < 0 if dt1 is < dt2; = 0 if dt1 = dt2; > 0 if dt1 >= dt2.

datetimeToSecs

```
int datetimeToSecs(DateTime dt)
```

Args:

- dt DateTime value to convert.

Converts a DateTime structure into the # of seconds since epoch (1|1|1970@0:0:0). The NOW time or date is substituted for missing time or date portions.

normalizeSize

```
int normalizeSize(Size *a, Size *b, Size *ta, Size *tb)
```

Args:

- a 1st size value to normalize.
- b 2nd size value to normalize.
- ta Where the normalize value to 'a' is placed.
- tb Where the normalize value to 'b' is placed.

Normalize 2 size value, a and b, adjusting them so that the shift counts are the same. The new values are placed in ta and tb, respectively. The shift that is "lower" of the 2 becomes the common denominator. Returns 0 if successful; -1 otherwise.

sizecmp

```
int sizecmp(Size a, Size w)
```

Args:

- a 1st size value to compare.
- w 2nd size value to compare.

Compares 2 Size structures, a and w, and returns +1 if a > w, 0 if a == w, -1 if a < w.

hashptr

```
static long int hashptr(void *ptr)
```

Args:

- ptr A pointer whose hash index to d_array will be obtained.

Returns a possible hash index for 'ptr'. Formula: ptr % MAXDARRAY.

getHashValue

```
static int getHashValue(void *ptr)
```

Args:

- ptr A pointer whose hash actual value to d_array will be obtained.

Returns the hash value for 'ptr', with hash collisions automatically resolved.

getHashValueToStore

```
static int getHashValueToStore(void *ptr)
```

Args:

- ptr A pointer that will be stored in system array d_array.

Returns the hash value where 'ptr' could be stored in d_array.

dynamicArraySize

```
static int dynamicArraySize(void *array)
```

Args:

- array A dynamic array.

Returns # of elements in 'array'.

initDynamicArray

```
void *initDynamicArray(size_t numElems, size_t elementSize)
```

Args:

numElems
: # of elements in the array to be created.

elementSize
: The size of each element in the array to be created.

Used calloc to create an array containing 'numElems' with each element of size 'elementSize'. A pointer to the newly-created array is returned, and also recorded in the 'd_array' table.

extendDynamicArray

```
void *extendDynamicArray(void *array, size_t minNumElems, size_t elementSize)
```

Args:

array
: A dynamic array to expand.

minNumElems
: # of elements in the array to be created.

elementSize
: The size of each element in the array to be created.

Expands the 'array' so that the minNumElems can at least fit. If 'array' is NULL, then initDynamicArray(minNumElems, elementSize) will be called. Information about the newly realloc-ed 'array' is recorded in the 'd_array' table (removing the previous old pointer), and the pointer to this newly- realloc-ed storage is also returned.

freeDynamicArray

```
void freeDynamicArray(void *array)
```

Args:

array
: A dynamic array to free up.

Free up the malloc-ed storage allocated to 'array', and remove its entry from the 'd_array' table.

printDynamicArrayTable

```
void printDynamicArrayTable(void)
```

Prints out the contents of the 'd_array' table.

datePrint

```
void datePrint(Date d)
```

Args:

d The Date structure to print out.

Print out 'd' in a human readable format.

timePrint

```
void timePrint(Time t)
```

Args:

t The Time structure to print out.

Print out 't' in human readable format.

datetimePrint

```
void timePrint(DateTime dt)
```

Args:

dt The DateTime structure to print out.

Print out 'dt' in human readable format.

sizePrint

```
void sizePrint(Size s, int readable)
```

Args:

s The Size structure to print out.

readable The format of output flag.

Print out 's' in human readable format if 'readable' flag is set to 1; otherwise, just print out the elements of the structure.

intRangePrint

```
void intRangePrint(IntRange r)
```

Args:

r The IntRange structure to print out.

Print out 'r' in human readable format.

floatRangePrint

```
void floatRangePrint(FloatRange r)
```

Args:

r The FloatRange structure to print out.

Print out 'r' in human readable format.

dayofweekPrint

```
void dayofweekPrint(Dayofweek dow)
```

Args:

dow The Dayofweek structure to print out.

Print out 'dow' in human readable format.

dayofweekRangePrint

```
void dayofweekRangePrint(DayofweekRange r)
```

Args:

r The DayofweekRange structure to print out.

Print out 'r' in human readable format.

dateRangePrint

```
void dateRangePrint(DateRange d)
```

Args:

d The DateRange structure to print out.

Print out 'd' in human readable format.

timeRangePrint

```
void timeRangePrint(TimeRange t)
```

Args:

t The TimeRange structure to print out.

Print out 't' in human readable format.

datetimeRangePrint

```
void timeRangePrint(DateTimeRange dt)
```

Args:

dt The DateTimeRange structure to print out.

Print out 'dt' in human readable format.

sizeRangePrint

```
void sizeRangePrint(SizeRange s, int readable)
```

Args:

s The SizeRange structure to print out.

readable The format of output flag.

Print out 's' in human readable format if 'readable' flag is set to 1; otherwise, just print out the elements of the Size structure.

strToIntRange

```
IntRange strToIntRange(char *str)
```

Args:

str The string to convert: "(low Int, high Int)"

Converts 'str' into an IntRange structure, returning the latter.

strToFloatRange

```
FloatRange strToFloatRange(char *str)
```

Args:

str The string to convert: "(low Float, high Float)"

Converts 'str' into a FloatRange structure, returning the latter.

strToDayofweekRange

```
DayofweekRange strToDayofweekRange(char *str)
```

Args:

str The string to convert: "(low dow, high dow)"

Converts 'str' into a DayofweekRange structure, returning the latter.

strToDateRange

```
DateRange strToDateRange(char *str)
```

Args:

str The string to convert: "(low date, high date)"

Converts 'str' into a DateRange structure, returning the latter.

strToTimeRange

```
DateRange strToTimeRange(char *str)
```

Args:

str The string to convert: "(low time, high time)"

Converts 'str' into a TimeRange structure, returning the latter.

strToDateTimeRange

```
DateRange strToDateTimeRange(char *str)
```

Args:

str The string to convert: "(low datetime, high datetime)"

Converts 'str' into a DateTimeRange structure, returning the latter.

sizeRangecmp

```
int sizeRangecmp(SizeRange r1, SizeRange r2)
```

Args:

r1 1st SizeRange structure to compare.

r2 2nd SizeRange structure to compare.

Compares 2 size ranges, r1 and r2, and returns 0 if they're the same; 1, otherwise.

sizeStrcmp

```
int sizeStrcmp(char *a, char *w)
```

Args:

- a 1st string to compare with format "<number><suffix>".
- w 2nd string to compare with format "<number><suffix>".

Compares 2 size-formatted strings, a and w, and returns +1 if a > w, 0 if a == w, and -1 if a < w.

sizeRangeStrcmp

```
int sizeRangeStrcmp(char *a, char *w)
```

Args:

- a 1st string to compare with format "(<number><suffix>, <number><suffix>)".
- w 2nd string to compare with format "(<number><suffix>, <number><suffix>)".

Compares 2 sizeRange-formatted strings, a and w, and returns +1 if a > w, 0 if a == w, and -1 if a < w.

toIntRange

```
IntRange toIntRange(int i1, int i2)
```

Args:

- i1 1st number in the range.
- i2 2nd number in the range.

Converts the 2 given numbers into an IntRange structure.

toFloatRange

```
FloatRange toFloatRange(double f1, double f2)
```

Args:

- f1 1st number in the range.
- f2 2nd number in the range.

Converts the 2 given numbers into a FloatRange structure.

toDayofweekRange

```
DayofweekRange toDayofweekRange(Dayofweek dow1, Dayofweek dow2)
```

Args:

- dow1 1st entity in the range.

dow2 2nd entity in the range.

Converts the 2 given entities into a DayofweekRange structure.

toDateRange

```
DateRange toDateRange(Date d1, Date d2)
```

Args:

d1 1st entity in the range.

d2 2nd entity in the range.

Converts the 2 given entities into a DateRange structure.

toTimeRange

```
TimeRange toTimeRange(Time t1, Time t2)
```

Args:

t1 1st entity in the range.

t2 2nd entity in the range.

Converts the 2 given entities into a TimeRange structure.

toDateTimeRange

```
DateTimeRange toDateTimeRange(DateTime dt1, DateTime dt2)
```

Args:

dt1 1st entity in the range.

dt2 2nd entity in the range.

Converts the 2 given entities into a DateTimeRange structure.

toSizeRange

```
SizeRange toSizeRange(Size sz1, Size sz2)
```

Args:

sz1 1st entity in the range.

sz2 2nd entity in the range.

Converts the 2 given entities into a SizeRange structure.

sizeAdd

```
Size sizeAdd(Size a, Size w)
```

Args:

a left operand.

w right operand.

Adds 2 Sizes together, returning the result. For values of different suffixes, normalization will take place; the suffix in the final result is the suffix that is the smaller of the two. If adding the two values would result in an overflow during the normalization step, then -1b is returned.

sizeSub

```
Size sizeSub(Size a, Size w)
```

Args:

a left operand.

w right operand.

Subtracts 2 Sizes together, returning the result. For values of different suffixes, normalization will take place; the suffix in the final result is the suffix that is the smaller of the two. If adding the two values would result in an overflow during the normalization step, then -1b is returned.

sizeMul

```
Size sizeMul(Size a, Size w)
```

Args:

a left operand.

w right operand.

Multiplies 2 Sizes together, returning the result. For values of different suffixes, normalization will take place; the suffix in the final result is the suffix that is the smaller of the two. If adding the two values would result in an overflow during the normalization step, then -1b is returned.

sizeDiv

```
Size sizeDiv(Size a, Size w)
```

Args:

a left operand.

w right operand.

Divides 2 Sizes together, returning the result. For values of different suffixes, normalization will take place; the suffix in the final result is the suffix that is the smaller of the two. If adding the two values would result in an overflow during the normalization step, then -1b is returned.

sizeUminus

```
Size sizeUminus(Size sz)
```

Args:

sz A Size operand.

Multiplies the numeric part of 'sz' by -1, returning the result.

strCat

```
char *strCat(char *str1, char *str2)
```

Args:

str1 left operand.

str2 right operand.

Concatenates 2 malloc-ed strings into one string, returning the result. The returned string is a pointer to a malloc-ed area whose scope in the varstr table will be -1. So, calling a varstr-FreeByScope(-1) will clean up the temporary storage.

mallocSubIndexTableAdd

```
void mallocSubIndexTableAdd(struct mallocIndex_type *ptr)
```

Args:

ptr The pointer value to add.

Adds a new entry (content is as given) to the mallocSubIndexTable, hashed against ptr->mptr->scope.

mallocSubIndexTableFree

```
void mallocSubIndexTableFree(struct mallocIndex_type *ptr)
```

Args:

ptr associated ptr value

Free up the entry of mallocSubIndexTable whose ptr value is 'ptr'.

mallocIndexTableAdd

```
void mallocIndexTableAdd(struct malloc_type *ptr)
```

Args:

ptr The pointer value to add.

Adds a new entry (content is as given) to the mallocIndexTable, hashed against ptr->pptr.

mallocIndexTableFree

```
void mallocIndexTableFree(struct malloc_type *ptr)
```

Args:

ptr associated ptr value

Free up the entry of mallocIndexTable whose associated ptr value is 'ptr', and also frees up any mallocSubIndexTable entry that hangs off it.

mallocIndexTableFreeNoIndex

```
void mallocIndexTableFreeNoIndex(struct malloc_type *ptr)
```

Args:

ptr associated ptr value

Like mallocIndexTableFree() except only the associated mallocSubIndexTable entry value is freed.

mallocIndexTableFreeNoIndex

```
void mallocIndexTableFreeNoSubIndex(struct malloc_type *ptr)
```

Args:

ptr associated ptr value

Like mallocIndexTableFree() except only the associated mallocIndexTable entry value is freed, and not any mallocSubIndexTable entry associated with it.

mallocTableAdd

```
void mallocTableAdd(void *ptr, void *pptr, int scope)
```

Args:

ptr The pointer value to add.

pptr The parent pointer to assign 'ptr'.

scope The scope value to assign 'ptr'.

Adds a new entry (content is as given) to the mallocTable, hashed against 'ptr'.

mallocTablePrint

```
void mallocTablePrint(void)
```

Prints out the contents of mallocTable, mallocIndexTable, mallocSubIndexTable

inMallocTable

```
int inMallocTable(void *ptr)
```

Args:

ptr A pointer value to look for in the mallocTable.

Returns 1 if 'ptr' is one of the entries on the mallocTable; 0, otherwise.

mallocTableInit

```
int mallocTableInit(void)
```

Initializes the mallocTable, mallocIndexTable, mallocSubIndexTable entries.

mallocTableFree

```
void mallocTableFree(void *ptr)
```

Args:

ptr A pointer value in the mallocTable to free up.

Free up the malloc-ed storage occupied by 'ptr', and also remove its entry from the mallocTable. Also, frees up any associated mallocIndexTable and mallocSubIndexTable entries.

mallocTableFreeNoIndex

```
void mallocTableFreeNoIndex(void *ptr)
```

Args:

ptr A pointer value in the mallocTable to free up.

Like mallocTableFree except associated mallocIndexTable entry is not freed.

mallocTableFreeNoSubIndex

```
void mallocTableFreeNoSubIndex(void *ptr)
```

Args:

ptr A pointer value in the mallocTable to free up.

Like mallocTableFree except associated mallocSubIndexTable entry is not freed.

mallocTableFreeNoSubIndex2

```
void mallocTableFreeNoSubIndex2(void *ptr)
```

Args:

ptr A pointer value in the mallocTable to free up.

Like mallocTableFree except associated mallocSubIndexTable entry is not freed, as well as the 'ptr' itself is not freed.

mallocTableFreeByPptr

```
void mallocTableFreeByPptr(void *pptr)
```

Args:

pptr A parent pointer value in the mallocTable.

Free up pointers to storage associated with 'pptr' (parent pointer value of 'pptr') and remove the corresponding slots from mallocTable. Also, any mallocTableSubIndex entry is freed.

mallocTableFreeByScope

```
void mallocTableFreeByScope(int scope, void (*freefunc)())
```

Args:

scope A scope value to look for in mallocTable.

freefunc The free function to use when freeing storage associated with 'scope'.

Free up pointers to storage associated with 'scope' value and remove the corresonding slots from mallocTable. Also, update the associated mallocSubIndexTable entry.

mallocTableModScope

```
void mallocTableModScope(void *ptr, int newscope)
```

Args:

ptr A pointer value to look for in mallocTable.

newscopeThe new scope for 'ptr'.

Modify 'ptr's scope value to 'newscope'. Appropriately update the mallocIndexTable() that hangs off of this.

mallocTableSafeModScope

```
void mallocTableSafeModScope(void *ptr, int newscope)
```

Args:

ptr A pointer value to look for in mallocTable.

newscopeThe new scope for 'ptr'.

Modify ptr's scope value to 'newscope' IF the current scope value is != 0. Appropriately update the mallocSubIndexTable() that hangs off of this.

inIntRange

```
int inIntRange(int i, IntRange range)
```

Args:

i An integer value.

range A range of numbers.

Returns 1 if i is in 'range'; 0 otherwise.

inFloatRange

```
int inFloatRange(double f, FloatRange range)
```

Args:

f A float value.

range A range of numbers.

Returns 1 if f is in 'range'; 0 otherwise.

inDayofweekRange

```
int inDayofweekRange(Dayofweek dow, DayofweekRange range)
```

Args:

dow A float value.

range A range of days of week.

Returns 1 if dow is in 'range'; 0 otherwise.

inDateRange

```
int inDateRange(Date d, DateRange range)
```

Args:

d A Date value.

range A range of Dates.

Returns 1 if d is in 'range'; 0 otherwise.

inTimeRange

```
int inTimeRange(Time t, TimeRange range)
```

Args:

t A Time value.

range A range of Times.

Returns 1 if t is in 'range'; 0 otherwise.

inDateTimeRange

```
int inDateTimeRange(DateTime dt, DateTimeRange range)
```

Args:

t A DateTime value.

range A range of DateTimes.

Returns 1 if dt is in 'range'; 0 otherwise.

NOTE: If date/times contain only time portions (i.e. hh:mm:ss) and low > hi, then adjust hi by 1 day.

inSizeRange

```
int inSizeRange(Size sz, SizeRange range)
```

Args:

sz A DateTime value.

range A range of DateTimes.

Returns 1 if 'sz' is in 'range'; 0 otherwise.

IntResCreate

```
static struct IntRes *IntResCreate(void)
```

Creates/mallocs a new IntRes structure, returning the pointer to it.

IntResValueGet

```
int IntResValueGet(struct IntRes *head, char *name)
```

Args:

head The 1st element in a list of IntRes structures.

name The resource name to look for in the list.

Return 'value', given the 'name' in the IntRes list.

IntResListPrint

```
void IntResListPrint(struct IntRes *head, char *descr)
```

Args:

head The 1st element in a list of IntRes structures.

descr Additional string description to print out.

Prints out the elements of IntRes list whose 1st element is 'head'. Print the message 'descr' along with the list output.

IntResValuePut

```
struct IntRes *IntResValuePut(struct IntRes *head, char *name, int value,
                                                    void *pptr)
```

Args:

head The 1st element in a list of IntRes structures.

name The resource name.

value The new resource value.

pptr The parent pointer to associate malloc-ed storage on the IntRes list.

If a resource 'name' is matched in IntRes list, then the matching element is updated so that its resource value is set to 'value'. If no such element exists, then a new entry is malloc-ed/created. Information about malloc-ed areas that were created as a result of this call will be recorded in the mallocTable and varstr table, with parent pointer values set to 'pptr'.

IntResListFree

```
void IntResListFree(struct IntRes *head)
```

Args:

head The 1st element in a list of IntRes structures.

Frees up the element of an IntRes list starting with 'head'.

SizeResCreate

```
static struct SizeRes *SizeResCreate(void)
```

Creates/mallocs a new SizeRes structure, returning the pointer to it.

SizeResValueGet

```
int SizeResValueGet(struct SizeRes *head, char *name)
```

Args:

head The 1st element in a list of SizeRes structures.

name The resource name to look for in the list.

Return 'value', given the 'name' in the SizeRes list.

SizeResListPrint

```
void SizeResListPrint(struct SizeRes *head, char *descr)
```

Args:

head The 1st element in a list of SizeRes structures.

descr Additional string description to print out.

Prints out the elements of SizeRes list whose 1st element is 'head'. Print the message 'descr' along with the list output.

SizeResValuePut

```
struct SizeRes *SizeResValuePut(struct SizeRes *head, char *name, int value,
                                                    void *pptr)
```

Args:

- head The 1st element in a list of SizeRes structures.
- name The resource name.
- value The new resource value.
- pptr The parent pointer to associate malloc-ed storage on the SizeRes list.

In SizeRes list, this function modifies an existing element which matches 'name', updating its resource value to 'value'. If no such element exists, then a new entry is malloc-ed/created. Information about malloc-ed areas that were created as a result of this call will be recorded in the mallocTable and varstr table, with parent pointer values set to 'pptr'.

SizeResListFree

```
void SizeResListFree(struct SizeRes *head)
```

Args:

- head The 1st element in a list of SizeRes structures.

Frees up the element of a SizeRes list starting with 'head'.

StringResCreate

```
static struct StringRes *StringResCreate(void)
```

Creates/mallocs a new StringRes structure, returning the pointer to it.

StringResValueGet

```
int StringResValueGet(struct StringRes *head, char *name)
```

Args:

- head The 1st element in a list of StringRes structures.
- name The resource name to look for in the list.

Return 'value', given the 'name' in the StringRes list.

StringResListPrint

```
void StringResListPrint(struct StringRes *head, char *descr)
```

Args:

head	The 1st element in a list of StringRes structures.
descr	Additional string description to print out.

Prints out the elements of StringRes list whose 1st element is 'head'. Print the message 'descr' along with the list output.

StringResValuePut

```
struct StringRes *StringResValuePut(struct StringRes *head, char *name, int value,
                                                    void *pptr)
```

Args:

head	The 1st element in a list of StringRes structures.
name	The resource name.
value	The new resource value.
pptr	The parent pointer to associate malloc-ed storage on the StringRes list.

In StringRes list, this function modifies an existing element which matches 'name', updating its resource value to 'value'. If no such element exists, then a new entry is malloc-ed/created. Information about malloc-ed areas that were created as a result of this call will be recorded in the mallocTable and varstr table, with parent pointer value set to 'pptr'.

StringResListFree

```
void StringResListFree(struct StringRes *head)
```

Args:

head	The 1st element in a list of StringRes structures.

Frees up the element of a StringRes list starting with 'head'.

6.1.4.2. ResMom

The source code found under the *ResMom* subdirectory contains data structures and functions that are used by the PBS ResMom (resource monitor) abstraction. ResMom responds to resource queries such as "loadave", "numCpus", and so on. The files involved are af_resmom.h and af_resmom.c. The main data structure used is:

```
struct resmom_struct {
  char *inetAddr;
  int  portNumber;
  int  connectFd;
}
typedef struct resmom_struct ResMom;
```

6.1.4.2.1. File: af_resmom.c

ResMomInetAddrGet

```
char *ResMomInetAddrGet(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Returns the official host name assigned to 'mom'.

ResMomPortNumberGet

```
int ResMomPortNumberGet(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Returns the network port number assgned to 'mom'.

ResMomConnectFdGet

```
int ResMomConnectFdGet(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Returns the connect file descriptor assgned to 'mom'.

ResMomInetAddrPut

```
void ResMomInetAddrPut(ResMom *mom, char *mom_name)
```

Args:

mom Pointer to a ResMom structure.

mom_name
A name for 'mom'.

Assigns (mallocs) mom_name to 'mom'. The string name has a global scope of 0.

ResMomPortNumberPut

```
void ResMomPortNumberPut(ResMom *mom, int port)
```

Args:

mom Pointer to a ResMom structure.

port Port number to 'mom'.

Assigns port as port number to 'mom'.

ResMomConnectFdPut

```
void ResMomConnectFdPut(ResMom *mom, int fd)
```

Args:

mom Pointer to a ResMom structure.

fd New connect file descriptor to 'mom'.

Assigns fd as connect file descriptor to 'mom'.

ResMomOpen

```
int ResMomOpen(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Opens a connection to the resource monitor using the "openrm()" call, and returns the resulting file descriptor. The 'connectFd' attribute of 'mom' is updated accordingly.

ResMomClose

```
int ResMomClose(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Closes a connection to the resource monitor using the "closerm()" call. Returns 0 if closerm() was successful; non-zero otherwise.

ResMomWrite

```
int ResMomWrite(ResMom *mom, char *buffer)
```

Args:

mom Pointer to a ResMom structure.

buffer A query string to send out to 'mom'.

Sends 'buffer' query to 'mom' using the addreq() call. Returns 1 if successful, non-zero otherwise.

ResMomRead

```
char *ResMomRead(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Returns the result of a previous resource query to sent to 'mom', using the getreq() call.

ResMomPrint

```
void ResMomPrint(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Prints out the elements of 'mom'.

ResMomInit

```
void ResMomInit(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Initializes 'mom'.

ResMomFree

```
void ResMomFree(ResMom *mom)
```

Args:

mom Pointer to a ResMom structure.

Frees up all malloc-ed storage associated with 'mom' structure.

6.1.4.3. CNode

The source code found under the *CNode* subdirectory contains data structures and functions that are used by the CNode abstraction. CNode stands for computational node, consisting of a shared memory, single OS image, and a set of CPUs. The files involved are af_cnode.h and af_cnode.c, af_cnodemap.h, af_cnodemap.c. The main data structures used are:

```
  struct IODevice {
    struct IODevice *nextptr;
    char *name;               /* unique identitiy of the device */
    Size spaceTotal;          /* total space on the device */
    Size spaceAvail;          /* space available on the device */
    Size spaceReserved;       /* space reserved for the jobs */
```

```
  int  inBw;                /* read bandwidth (bytes/s) or swap in rate */
  int  outBw;               /* write bandwidth (bytes/s) or swap out rate */
};

struct Network {
  struct Network *nextptr;
  char *type;               /* type of network - hippi, fddi, ... */
  int  bw;                  /* network bandwidth - in bytes/sec */
};

struct Memory {
  struct Memory *nextptr;
  char *type;               /* could be physical or virtual Mem */
  Size total;               /* total memory size */
  Size avail;               /* available memory */
};

struct cnode_struct {
  struct cnode_struct *nextptr;
  ResMom name;             /* the MOM representing the node */
  char   *properties;      /* comma-separated list of alias hostnames */
  char   *vendor;          /* system name */
  char   *os;              /* string describing the OS version */
  int    numCpus;          /* number of processors */
  int    state;            /* node state */
  int    type;             /* node type */
  int    queryMom;         /* flag */
  int    idletime;         /* time since last keystroke/mouse movement */
  int    cpuPercentIdle;   /* % of idletime experienced by all processors */
  int    cpuPercentSys;    /* % of time that all processors have spent */
                           /* running kernel code */
  int    cpuPercentUser;   /* % of time that all processors have spent */
                           /* running user code */
  int   cpuPercentGuest;   /* % of time that all processors have spent */
                           /* running a guest operating system */
  double loadave;          /* load average of all cpus in the node */
  struct Memory *mem;      /* memory */
  struct Network *network; /* list of network devices and their properties */
  struct IODevice *swap;   /* list of swap devices and their properties */
  struct IODevice disk;    /* list of disk devices and their properties */
  struct IODevice tape;    /* list of tape devices and their properties */
  struct IODevice srfs;    /* list of srfs devices and their properties */
  int    multiplicity;     /* during node requests, this is the # */
                           /* of nodes of this type requested */
};
typedef struct cnode_struct CNode;

struct SetCNode_type {    /* a Set of CNodes abstraction */
  CNode *head;
  CNode *tail;
  int   numAvail;
  int   numAlloc;
  int   numRsvd;
  int   numDown;
```

```
};
typedef struct SetCNode_type SetCNode;

struct CNodeAttrInfo {
  char *name;             /* name of a CNode struct member */
  int  type;              /* attribute type */
  void (*attrPutFunc)();  /* CNodePut function for attribute */
};

struct Resource {
  char *archType;
  char *nodeAttr;
  char *hostQuery_keyword;

};
```

6.1.4.3.1. File: af_cnode.c

IODeviceCreate

```
static struct IODevice *IODeviceCreate(void)
```

Mallocs a new struct IODevice structure, initializes the values of its elements, and then returns the pointer to the structure.

IODeviceSpaceTotalGet

```
static Size IODeviceSpaceTotalGet(struct IODevice *iod_head, char *name)
```

Args:

iod_head1st device in the device list.

name Name assigned to the device.

Returns the spaceTotal attribute value of a device named 'name' as found in the list of devices.

IODeviceSpaceAvailGet

```
static Size IODeviceSpaceAvailGet(struct IODevice *iod_head, char *name)
```

Args:

iod_head1st device in the device list.

name Name assigned to the device.

Returns the spaceAvail attribute value of a device named 'name' as found in the list of devices.

IODeviceSpaceReservedGet

```
static Size IODeviceSpaceReservedGet(struct IODevice *iod_head, char *name)
```

Args:

iod_head1st device in the device list.

name Name assigned to the device.

Returns the spaceReserved attribute value of a device named 'name' as found in the list of devices.

IODeviceInBwGet

```
static int IODeviceInBwGet(struct IODevice *iod_head, char *name)
```

Args:

iod_head1st device in the device list.

name Name assigned to the device.

Returns the inBw attribute value of a device named 'name' as found in the list of devices.

IODeviceOutBwGet

```
static int IODeviceOutBwGet(struct IODevice *iod_head, char *name)
```

Args:

iod_head1st device in the device list.

name Name assigned to the device.

Returns the outBw attribute value of a device named 'name' as found in the list of devices.

IODeviceListPrint

```
static void IODeviceListPrint(struct IODevice *iod_head, char *descr)
```

Args:

iod_head1st device in the device list.

descr A message string to print out.

Prints out the values of the list of IO devices headed by 'iod_head'.

IODeviceSpaceTotalPut

```
static struct IODevice *IODeviceSpaceTotalPut(struct IODevice *iod_head,
                                    char *name, Size total, void *pptr)
```

Args:

iod_head1st device in the device list.

name Name of the device.

total New spaceTotal attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the spaceTotal attribute value of the device named by 'name'. A new IODevice structure is created if no device named 'name' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

IODeviceSpaceAvailPut

```
static struct IODevice *IODeviceSpaceAvailPut(struct IODevice *iod_head,
                                    char *name, Size avail, void *pptr)
```

Args:

iod_head1st device in the device list.

name Name of the device.

avail New spaceAvail attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the spaceAvail attribute value of the device named by 'name'. A new IODevice structure is created if no device named 'name' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

IODeviceSpaceReservedPut

```
static struct IODevice *IODeviceSpaceReservedPut(struct IODevice *iod_head,
                                     char *name, Size reserve, void *pptr)
```

Args:

iod_head1st device in the device list.

name Name of the device.

reserve New spaceAvail attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the spaceReserved attribute value of the device named by 'name'. A new IODevice structure is created if no device named 'name' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

IODeviceSpaceInBwPut

```
static struct IODevice *IODeviceInBwPut(struct IODevice *iod_head,
```

```
                                    char *name, int inBw, void *pptr)
```

Args:

iod_head1st device in the device list.

name Name of the device.

inBw New inBw attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the inBw attribute value of the device named by 'name'. A new IODevice structure is created if no device named 'name' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

IODeviceSpaceOutBwPut

```
static struct IODevice *IODeviceOutBwPut(struct IODevice *iod_head,
                                    char *name, int outBw, void *pptr)
```

Args:

iod_head1st device in the device list.

name Name of the device.

outBw New outBw attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the outBw attribute value of the device named by 'name'. A new IODevice structure is created if no device named 'name' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

NetworkCreate

```
static struct Memory *NetworkCreate(void)
```

Mallocs a new struct Network structure, initializes the values of its elements, and then returns the pointer to the structure.

NetworkBwGet

```
static int NetworkBwGet(struct Network *net_head, char *name)
```

Args:

net_head1st network device in the device list.

name Name assigned to the network device.

Returns the bw attribute value of a network device named 'name' as found in the list of devices.

NetworkListPrint

```
static void NetworkListPrint(struct IODevice net_head)
```

Args:

net_head1st network device in the device list.

Prints out the various network names and respective bandwidths in the list of network devices.

NetworkBwPut

```
static struct IODevice *NetworkBwPut(struct IODevice *net_head,
                                     char *type, int bw, void *pptr)
```

Args:

net_head1st network device in the device list.

type Type of the network device.

bw New bw attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the bw attribute value of the network device named by 'type'. A new Network structure is created if no network device named 'type' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

MemoryCreate

```
static struct Memory *MemoryCreate(void)
```

Mallocs a new struct Memory structure, initializes the values of its elements, and then returns the pointer to the structure.

MemoryTotalGet

```
static Size MemoryTotalGet(struct Memory *mem_head, char *type)
```

Args:

mem_head
1st memory device in the device list.

type Name assigned to the memory device (i.e. physical or virtual).

Returns the total attribute value of a memory device named 'type' as found in the list of devices. -1B is returned if no such device is found.

MemoryAvailGet

```
static Size MemoryAvailGet(struct Memory *mem_head, char *type)
```

Args:

mem_head
: 1st memory device in the device list.

type
: Name assigned to the memory device (i.e. physical or virtual).

Returns the avail attribute value of a memory device named 'type' as found in the list of devices. -1B is returned if no such device is found.

MemoryListPrint

```
static void MemoryListPrint(struct Memory *mem_head)
```

Args:

mem_head
: 1st memory device in the device list.

Prints out the various memory types and respective attributes in the list of memory devices.

MemoryTotalPut

```
static struct Memory *MemoryTotalPut(struct Memory *mem_head,
                        char *type, Size newTot, void *pptr)
```

Args:

mem_head
: 1st memory device in the device list.

type
: Type of the memory device.

newTot
: New total attribute value.

pptr
: Some parent pointer to associate a device's malloc-ed storage.

Modifies the total attribute value of the memory device named by 'type'. A new Memory is created if no memory device named 'type' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

MemoryAvailPut

```
static struct Memory *MemoryAvailPut(struct Memory *mem_head,
                       char *type, Size newAvail, void *pptr)
```

Args:

mem_head
1st memory device in the device list.

type Type of the memory device.

newAvailNew avail attribute value.

pptr Some parent pointer to associate a device's malloc-ed storage.

Modifies the avail attribute value of the memory device named by 'type'. A new Memory is created if no memory device named 'type' exists. For any newly malloc-ed area, an association with 'pptr' is established. This function returns non-NULL if there's a new head of the list (takes place when a new device is added to the list); NULL otherwise.

CNodeResMomInetAddrGet

```
ResMom *CNodeResMomGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the ResMom structure representing 'node'.

CNodeNameGet

```
char *CNodeNameGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the official name of the 'node'.

CNodePropertiesGet

```
char *CNodePropertiesGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the properties attribute of the 'node'.

CNodeOsGet

```
char *CNodeOsGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the os attribute of the 'node'.

CNodeNumCpusGet

```
int CNodeNumCpusGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the numCpus attribute of the 'node'.

CNodeMemTotalGet

```
Size CNodeMemTotalGet(CNode *node, char *type)
```

Args:

node Pointer to a CNode structure.

type Type of memory.

Returns the mem[type]->total attribute of the 'node'. -1B if undefined.

CNodeMemAvailGet

```
Size CNodeMemAvailGet(CNode *node, char *type)
```

Args:

node Pointer to a CNode structure.

type Type of memory.

Returns the mem[type]->avail attribute of the 'node'. -1B if undefined.

CNodeStateGet

```
int CNodeStateGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the state attribute of the 'node'.

CNodeTypeGet

```
int CNodeTypeGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the type attribute of the 'node'.

CNodeQueryMomGet

```
int CNodeQueryMomGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the queryMom attribute of the 'node'.

CNodeIdletimeGet

```
int CNodeIdletimeGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the idletime attribute of the 'node'.

CNodeLoadAveGet

```
double CNodeLoadAveGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the loadAve attribute of the 'node'.

CNodeNetworkBwGet

```
int CNodeNetworkBwGet(CNode *node, char *type)
```

Args:

node Pointer to a CNode structure.

type type of network.

Returns the network[type]->bw attribute of the 'node'.

CNodeDiskSpaceTotalGet

```
Size CNodeDiskSpaceTotalGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a disk device.

Returns the disk[name]->spaceTotal attribute of the 'node'.

CNodeDiskSpaceAvailGet

```
Size CNodeDiskSpaceAvailGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a disk device.

Returns the disk[name]->spaceAvail attribute of the 'node'.

CNodeDiskSpaceReservedGet

```
Size CNodeDiskSpaceReservedGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a disk device.

Returns the disk[name]->spaceReserved attribute of the 'node'.

CNodeDiskInBwGet

```
int CNodeDiskInBwGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a disk device.

Returns the disk[name]->inBw attribute of the 'node'.

CNodeDiskOutBwGet

```
int CNodeDiskOutBwGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a disk device.

Returns the disk[name]->outBw attribute of the 'node'.

CNodeSwapSpaceTotalGet

```
Size CNodeSwapSpaceTotalGet(CNode *node, char *name)
```

Args:

- node Pointer to a CNode structure.
- name name of a swap device.

Returns the swap[name]->spaceTotal attribute of the 'node'.

CNodeSwapSpaceAvailGet

```
Size CNodeSwapSpaceAvailGet(CNode *node, char *name)
```

Args:

- node Pointer to a CNode structure.
- name name of a swap device.

Returns the swap[name]->spaceAvail attribute of the 'node'.

CNodeSwapSpaceReservedGet

```
Size CNodeSwapSpaceReservedGet(CNode *node, char *name)
```

Args:

- node Pointer to a CNode structure.
- name name of a swap device.

Returns the swap[name]->spaceReserved attribute of the 'node'.

CNodeSwapInBwGet

```
int CNodeSwapInBwGet(CNode *node, char *name)
```

Args:

- node Pointer to a CNode structure.
- name name of a swap device.

Returns the swap[name]->inBw attribute of the 'node'.

CNodeSwapOutBwGet

```
int CNodeSwapOutBwGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a swap device.

Returns the swap[name]->outBw attribute of the 'node'.

CNodeTapeSpaceTotalGet

```
Size CNodeTapeSpaceTotalGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a tape device.

Returns the tape[name]->spaceTotal attribute of the 'node'.

CNodeTapeSpaceAvailGet

```
Size CNodeTapeSpaceAvailGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a tape device.

Returns the tape[name]->spaceAvail attribute of the 'node'.

CNodeTapeSpaceReservedGet

```
Size CNodeTapeSpaceReservedGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a tape device.

Returns the tape[name]->spaceReserved attribute of the 'node'.

CNodeTapeInBwGet

```
int CNodeTapeInBwGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of a tape device.

Returns the tape[name]->inBw attribute of the 'node'.

CNodeTapeOutBwGet

```
int CNodeTapeOutBwGet(CNode *node, char *name)
```

Args:

node	Pointer to a CNode structure.
name	name of a tape device.

Returns the tape[name]->outBw attribute of the 'node'.

CNodeSrfsSpaceTotalGet

```
Size CNodeSrfsSpaceTotalGet(CNode *node, char *name)
```

Args:

node	Pointer to a CNode structure.
name	name of an srfs device.

Returns the srfs[name]->spaceTotal attribute of the 'node'.

CNodeSrfsSpaceAvailGet

```
Size CNodeSrfsSpaceAvailGet(CNode *node, char *name)
```

Args:

node	Pointer to a CNode structure.
name	name of an srfs device.

Returns the srfs[name]->spaceAvail attribute of the 'node'.

CNodeSrfsSpaceReservedGet

```
Size CNodeSrfsSpaceReservedGet(CNode *node, char *name)
```

Args:

node	Pointer to a CNode structure.
name	name of an srfs device.

Returns the srfs[name]->spaceReserved attribute of the 'node'.

CNodeSrfsInBwGet

```
int CNodeSrfsInBwGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of an srfs device.

Returns the srfs[name]->inBw attribute of the 'node'.

CNodeSrfsOutBwGet

```
int CNodeSrfsOutBwGet(CNode *node, char *name)
```

Args:

node Pointer to a CNode structure.

name name of an srfs device.

Returns the srfs[name]->outBw attribute of the 'node'.

CNodeCpuPercentIdleGet

```
int CNodeCpuPercentIdleGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the cpuPercentIdle attribute of the 'node'.

CNodeCpuPercentSysGet

```
int CNodeCpuPercentSysGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the cpuPercentSys attribute of the 'node'.

CNodeCpuPercentUserGet

```
int CNodeCpuPercentUserGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the cpuPercentUser attribute of the 'node'.

CNodeCpuPercentGuestGet

```
int CNodeCpuPercentGuestGet(CNode *node)
```

Args:

node Pointer to a CNode structure.

Returns the cpuPercentGuest attribute of the 'node'.

CNodeResMomPut

```
void CNodeResMomPut(CNode *node, ResMom *mom)
```

Args:

node Pointer to a CNode structure.

mom New mom structure.

Sets node->mom attribute value to *mom.

CNodePropertiesPut

```
void CNodePropertiesPut(CNode *node, char *properties)
```

Args:

node Pointer to a CNode structure.

properties
New properties

Sets node->properties attribute value to 'properties'.

CNodeVendorPut

```
void CNodeVendorPut(CNode *node, char *vendor)
```

Args:

node Pointer to a CNode structure.

vendor New vendor name.

Sets node->vendor attribute value to 'vendor'.

CNodeOsPut

```
void CNodeOsPut(CNode *node, char *os)
```

Args:

node Pointer to a CNode structure.

os New os type.

Sets node->os attribute value to 'os'.

CNodeNumCpusPut

```
void CNodeNumCpusPut(CNode *node, int ncpus)
```

Args:

node Pointer to a CNode structure.

ncpus Number of cpus.

Sets node->NumCpus attribute value to 'ncpus'.

CNodeMemTotalPut

```
void CNodeMemTotalPut(CNode *node, char *type, Size pmem)
```

Args:

node Pointer to a CNode structure.

type A type of memory.

pmem New memory total value.

Sets the node's mem[type]->total attribute value to 'pmem'.

CNodeMemAvailPut

```
void CNodeMemAvailPut(CNode *node, char *type, Size pmem)
```

Args:

node Pointer to a CNode structure.

type A type of memory.

pmem New memory total value.

Sets the node's mem[type]->avail attribute value to 'pmem'.

CNodeStatePut

```
void CNodeStatePut(CNode *node, int state)
```

Args:

node Pointer to a CNode structure.

state New state attribute value.

Sets the node's state attribute value to 'state'.

CNodeTypePut

```
void CNodeTypePut(CNode *node, int type)
```

Args:

node Pointer to a CNode structure.

type New type attribute value.

Sets the node's type attribute value to 'type'.

CNodeQueryMomPut

```
void CNodeQueryMomPut(CNode *node, int queryMom)
```

Args:

node Pointer to a CNode structure.

queryMom
New queryMom attribute value.

Sets the node's queryMom attribute value to 'queryMom'.

CNodeIdletimePut

```
void CNodeIdletimePut(CNode *node, int idletime)
```

Args:

node Pointer to a CNode structure.

idletime New idletime attribute value.

Sets the node's idletime attribute value to 'idletime'.

CNodeLoadAvePut

```
void CNodeLoadAvePut(CNode *node, double loadave)
```

Args:

node Pointer to a CNode structure.

loadave New loadave attribute value.

Sets the node's loadAve attribute value to 'loadave'.

CNodeDiskSpaceTotalPut

```
void CNodeDiskSpaceTotalPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of a disk device to update info on.
size	New size value to spaceTotal attribute.

Sets the node's disk[name]->spaceTotal attribute value to 'size'.

CNodeDiskSpaceAvailPut

```
void CNodeDiskSpaceAvailPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of a disk device to update info on.
size	New size value to spaceAvail attribute.

Sets the node's disk[name]->spaceAvail attribute value to 'size'.

CNodeDiskSpaceReservedPut

```
void CNodeDiskSpaceReservedPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of a disk device to update info on.
size	New size value to spaceReserved attribute.

Sets the node's disk[name]->spaceReserved attribute value to 'size'.

CNodeDiskInBwPut

```
void CNodeDiskInBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of a disk device to update info on.
bw	New bandwidth value to inBw attribute.

Sets the node's disk[name]->inBw attribute value to 'bw'.

CNodeDiskOutBwPut

```
void CNodeDiskOutBwPut(CNode *node, char *name, int bw)
```

Args:

node Pointer to a CNode structure.

name Name of a disk device to update info on.

bw New bandwidth value to outBw attribute.

Sets the node's disk[name]->outBw attribute value to 'bw'.

CNodeSwapSpaceTotalPut

```
void CNodeSwapSpaceTotalPut(CNode *node, char *name, Size swaptot)
```

Args:

node Pointer to a CNode structure.

name Name of a swap device to update info on.

swaptot New value to spaceTotal attribute.

Sets the node's swap[name]->spaceTotal attribute value to 'swaptot'.

CNodeSwapSpaceAvailPut

```
void CNodeSwapSpaceAvailPut(CNode *node, char *name, Size swapavail)
```

Args:

node Pointer to a CNode structure.

name Name of a swap device to update info on.

swapavail
New value to spaceAvail attribute.

Sets the node's swap[name]->spaceAvail attribute value to 'swapavail'.

CNodeSwapSpaceReservedPut

```
void CNodeSwapSpaceReservedPut(CNode *node, char *name, Size swapres)
```

Args:

node Pointer to a CNode structure.

name Name of a swap device to update info on.

swapres New value to spaceReserved attribute.

Sets the node's swap[name]->spaceReserved attribute value to 'size'.

CNodeSwapInBwPut

```
void CNodeSwapInBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of a swap device to update info on.
bw	New bandwidth value to inBw attribute.

Sets the node's swap[name]->inBw attribute value to 'bw'.

CNodeSwapOutBwPut

```
void CNodeSwapOutBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of a swap device to update info on.
bw	New bandwidth value to outBw attribute.

Sets the node's swap[name]->outBw attribute value to 'bw'.

CNodeTapeSpaceTotalPut

```
void CNodeTapeSpaceTotalPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of a tape device to update info on.
size	New value to spaceTotal attribute.

Sets the node's tape[name]->spaceTotal attribute value to 'size'.

CNodeTapeSpaceAvailPut

```
void CNodeTapeSpaceAvailPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of a tape device to update info on.
size	New value to spaceAvail attribute.

Sets the node's tape[name]->spaceAvail attribute value to 'size'.

CNodeTapeSpaceReservedPut

```
void CNodeTapeSpaceReservedPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of a swap device to update info on.
size	New value to spaceReserved attribute.

Sets the node's tape[name]->spaceReserved attribute value to 'size'.

CNodeTapeInBwPut

```
void CNodeTapeInBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of a tape device to update info on.
bw	New bandwidth value to inBw attribute.

Sets the node's tape[name]->inBw attribute value to 'bw'.

CNodeTapeOutBwPut

```
void CNodeTapeOutBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of a tape device to update info on.
bw	New bandwidth value to outBw attribute.

Sets the node's tape[name]->outBw attribute value to 'bw'.

CNodeSrfsSpaceTotalPut

```
void CNodeSrfsSpaceTotalPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of an srfs device to update info on.
size	New value to spaceTotal attribute.

Sets the node's srfs[name]->spaceTotal attribute value to 'size'.

CNodeSrfsSpaceAvailPut

```
void CNodeSrfsSpaceAvailPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of an srfs device to update info on.
size	New value to spaceAvail attribute.

Sets the node's srfs[name]->spaceAvail attribute value to 'size'.

CNodeSrfsSpaceReservedPut

```
void CNodeSrfsSpaceReservedPut(CNode *node, char *name, Size size)
```

Args:

node	Pointer to a CNode structure.
name	Name of an srfs device to update info on.
size	New value to spaceReserved attribute.

Sets the node's srfs[name]->spaceReserved attribute value to 'size'.

CNodeSrfsInBwPut

```
void CNodeSrfsInBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of an srfs device to update info on.
bw	New bandwidth value to inBw attribute.

Sets the node's srfs[name]->inBw attribute value to 'bw'.

CNodeSrfsOutBwPut

```
void CNodeSrfsOutBwPut(CNode *node, char *name, int bw)
```

Args:

node	Pointer to a CNode structure.
name	Name of an srfs device to update info on.
bw	New bandwidth value to outBw attribute.

Sets the node's srfs[name]->outBw attribute value to 'bw'.

CNodeCpuPercentIdlePut

```
void CNodeCpuPercentIdlePut(CNode *node, int percent)
```

Args:

node Pointer to a CNode structure.

percent New value to cpuPercentIdle attribute.

Sets the node's cpuPercentIdle attribute value to 'percent'.

CNodeCpuPercentSysPut

```
void CNodeCpuPercentSysPut(CNode *node, int percent)
```

Args:

node Pointer to a CNode structure.

percent New value to cpuPercentSys attribute.

Sets the node's cpuPercentSys attribute value to 'percent'.

CNodeCpuPercentUserPut

```
void CNodeCpuPercentUserPut(CNode *node, int percent)
```

Args:

node Pointer to a CNode structure.

percent New value to cpuPercentUser attribute.

Sets the node's cpuPercentUser attribute value to 'percent'.

CNodeCpuPercentGuestPut

```
void CNodeCpuPercentGuestPut(CNode *node, int percent)
```

Args:

node Pointer to a CNode structure.

percent New value to cpuPercentGuest attribute.

Sets the node's cpuPercentGuest attribute value to 'percent'.

CNodeFree

```
void CNodeFree(CNode *node)
```

Args:

node Pointer to a CNode structure.

Free up all malloc-ed storage associated with 'node'.

CNodeInit

```
void CNodeInit(CNode *node)
```

Args:

node Pointer to a CNode structure.

Initialized the values of the 'node' members to something that are consistent.

CNodePrint

```
void CNodePrint(CNode *node)
```

Args:

node Pointer to a CNode structure.

Print out the values of the 'node' members.

send_queries

```
static int send_queries(CNode *node, char *arch, int typeOfData,
                                     struct CNodeAttrInfo **buf)
```

Args:

node Pointer to a CNode structure.

arch Some unique classification of a node (could be its name or os type)

typeOfData
type of query to send out (i.e. STATIC_RESOURCE or DYNAMIC_RESOURCE)

buf an array of information describing each of the resource queries to be sent out. Description includes name of the corresponding CNode attribute, its type, and a pointer to the function that handles assigning the query result to the CNode attribute.

The algorithm is as follows:

```
 if typeOfData is STATIC_RESOURCE
 then

   foreach of the the known static attributes
   do
       get the corresponding resource query for the attribute based on a
         matching 'arch' value,
       fill the 'buf' array with relevant information,
       send out the query.
       reset the value of the corresponding attribute to some default value
         so that we can tell if if the query fails
       update the numSends count.
   done
 else if typeOfData is DYNAMIC_RESOURCE
   foreach of the the known dynamic attributes
```

do
get the corresponding resource query for the attribute based on a
matching 'arch' value.
fill the 'buf' array with relevant information,
send out the query.
update the numSends count.
done

NOTE: Information about the queries are listed in the array in the order
that they were sent.

Return the number of queries sent.

put_default_val

```
static int put_default_val(CNode *node, char *attrib, int type, void (*putfunc)())
```

Args:

node	Pointer to a CNode structure.
attrib	a CNode attribute name.
type	data type for the attribue
putfunc	The put function that will store the default value.

if 'type' is INT_TYPE, then put a value of -1
if 'type' is SIZE_TYPE, then put a value of -1.0b
if 'type' is FLT_TYPE, then put a value of -1.0
if 'type' is STR_TYPE, then put a value of NULLSTR

recv_responses

```
static int recv_responses(CNode *node, struct CNodeAttrInfo **buf)
```

Args:

node	Pointer to a CNode structure.
buf	an array of information describing each of the resource queries received. Description includes name of the corresponding CNode attribute, its type, and a pointer to the function that handles assigning the query result to the CNode attribute.

foreach of the query results received,
do
update the corresponding 'node' structure by using information provided
in the 'buf' array.
done
Return the number of query results received.

CNodeStateRead

```
void CNodeStateRead(CNode *node, int typeOfData)
```

Args:

node Pointer to a CNode structure.

typeOfData
The type of data to get for 'node' (i.e. STATIC_RESOURCE, DYNAMIC_RESOURCE)

Don't proceed if CNodeQueryMomGet(node) is set to 0.
open a connection to the node's ResMom.
get the node's ResMom name. Use this name to send resource queries that apply only to this name.
get the ResMom's responses to the queries.

If nodetype is CNODE_UNKNOWN, then update the node's state value: CNODE_FREE, CNODE_DOWN,
close connection to the ResMom.

foreach of the query results received,
do
update the corresponding 'node' structure by using information provided in the 'buf' array.
done

SetCNodeInit

```
void SetCNodeInit(SetCNode *scn)
```

Args:

scn Pointer to a Set of CNode structure.

Initializes the 'scn' structure so that point the head of the set and the tail of set are pointing to NOCNODE. Also, initialize the attributes numAvail, numAlloc, numRsvd, numDown to -1.

SetCNodeAdd

```
void SetCNodeAdd(SetCNode *scn, CNode *cn)
```

Args:

scn Pointer to a Set of CNode structure.

cn A new node to add.

Adds 'cn' to the set of CNodes.

SetCNodeFree

```
void SetCNodeFree(SetCNode *scn)
```

Args:

scn Pointer to a Set of CNode structure.

Free up all malloc-storage associated with scn.

SetCNodeFindCNodeByName

```
CNode *SetCNodeFindCNodeByName(SetCNode *scn, char *node_name)
```

Args:

scn Pointer to a Set of CNode structure.

node_name
A node_name to search for.

Return the node in the set of CNode, 'scn', whose name matches 'node_name'.

SetCNodePrint

```
void SetCNodePrint(SetCNode *scn)
```

Args:

scn Pointer to a Set of CNode structure.

Print out the structures associated with 'scn'.

inSetCNode

```
int inSetCNode(CNode *cn, SetCNode *scn)
```

Args:

cn A node to look for.

scn Pointer to a Set of CNode structure.

Returns 1 if 'cn' is a member of the set of nodes, 'scn'; 0 otherwise.

CNodePartition

```
int CNodePartition(struct CNodeSortArgs *A, int p, int r)
```

Args:

A stuff of information needed to reorder the elements of a set of CNodes.

p the "leftmost" element of a set of CNodes.

r the "rightmost" element of a set of CNodes.

This is the Partition() function in the well-known Quicksort() sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

CNodeQuickSort

```
void CNodeQuickSort (struct CNodeSortArgs *A, int p, int r)
```

Args:

A stuff of information needed to reorder the elements of a set of CNodes.

p the "leftmost" element of a set of CNodes.

r the "rightmost" element of a set of CNodes.

This is the Quicksort() function in the well-known quicksort sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

SetCNodeSortInt

```
int SetCNodeSortInt (SetCNode *s, int (*key)(), int order)
```

Args:

s the set of CNodes to reorder.

key the function to apply to each member of the set of CNodes whose int value will be used to reorder the set of CNodes.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of CNodes in a dynamic array, and then run CNodeQuicksort() function on the array, which also rearranges the pointers representing the set of CNodes.

SetCNodeSortStr

```
int SetCNodeSortStr (SetCNode *s, char *(*key)(), int order)
```

Args:

s the set of CNodes to reorder.

key the function to apply to each member of the set of CNodes whose char* value will be used to reorder the set of CNodes.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of CNodes in a dynamic array, and then run CNodeQuicksort() function on the array, which also rearranges the pointers representing the set of CNodes.

SetCNodeSortDateTime

```
int SetCNodeSortDateTime (SetCNode *s, DateTime (*key)(), int order)
```

Args:

- s — the set of CNodes to reorder.
- key — the function to apply to each member of the set of CNodes whose DateTime value will be used to reorder the set of CNodes.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of CNodes in a dynamic array, and then run CNodeQuicksort() function on the array, which also rearranges the pointers representing the set of CNodes.

SetCNodeSortSize

```
int SetCNodeSortSize (SetCNode *s, Size (*key)(), int order)
```

Args:

- s — the set of CNodes to reorder.
- key — the function to apply to each member of the set of CNodes whose Size value will be used to reorder the set of CNodes.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of CNodes in a dynamic array, and then run CNodeQuicksort() function on the array, which also rearranges the pointers representing the set of CNodes.

SetCNodeSortFloat

```
int SetCNodeSortFloat (SetCNode *s, double (*key)(), int order)
```

Args:

- s — the set of CNodes to reorder.
- key — the function to apply to each member of the set of CNodes whose double value will be used to reorder the set of CNodes.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of CNodes in a dynamic array, and then run CNodeQuicksort() function on the array, which also rearranges the pointers representing the set of CNodes.

6.1.4.3.2. File: af_cnodemap.h

This file defines 2 special variables called `static_attrinfo_map` and `dynamic_attrinfo_map` which are arrays of `struct CNodeAttrInfo` recording the various pointers to functions that update attribute values, attributes' types, and names of the respective CNodeGet functions. Update these data structure accordingly if a new CNodeGet() function has been added. Place those functions under static_attrinfo_map if the type of resource being mapped by the CNodeGet() function in question is STATIC in nature; otherwise, place them in dynamic_attrinfo_map.

6.1.4.3.3. File: af_cnodemap.c

nodeAttrCmpNoTag

```
int nodeAttrCmpNoTag(char *attr1, char *attr2)
```

Args:

attr1 1st attribute name to compare.

attr2 2nd attribute name to compare.

Compares 2 attribute names (ignoring tags in vector attributes), and returns 0 if they are of the same name. Note that a no-tag CNodeDiskSpaceReservedGet[] will match a tagged CNodeDiskSpaceReservedGet[$FASTDIR] (example).

parseAttrForTag

```
char *parseAttrForTag(char *attName)
```

Args:

attNameAn attribute name to parse.

Given an attribute name of the form, "name[tag]", return the "tag" part. If no tag, then return NULL.

getAttrType

```
int *getAttrType(char *attName)
```

Args:

attNameAn attribute name.

Given an attribute name, return its type. If 'attName' does not exist, then return -1.

getAttrPutFunc

```
void (*getAttrPutFunc(char *attName))()
```

Args:

attNameAn attribute name.

Given an attribute name, return its CNodePut() function. If 'attName' does not exist, then return NULL.

getStaticAttrAtIndex

```
char *getStaticAttrAtIndex(int index, int *type, void (**putfunc)())
```

Args:

- index An index value to the static_attrinfo_map array.
- type Type of the attribute.
- putfunc CNodePut() function of the attribute described by 'attName'.

Looks into the static_attrinfo_map[] array and return the name, type and putfunc information of the entry at 'index'.

getDynamicAttrAtIndex

```
char *getDynamicAttrAtIndex(int index, int *type, void (**putfunc)())
```

Args:

- index An index value to the dynamic_attrinfo_map array.
- type Type of the attribute.
- putfunc CNodePut() function of the attribute described by 'attName'.

Looks into the dynamic_attrinfo_map[] array and return the name, type and putfunc information of the entry at 'index'.

attrInfoMapPrint

```
void attrInfoMapPrint(void)
```

Prints out the entries of the static_attrinfo_map and dynamic_attrinfo_map.

addRes

```
int addRes(char *archType, char *nodeAttr, char *hostQuery)
```

Args:

- archType A class type of the resource (could be a node name or os type)
- nodeAttr A node attribute (or even CNodeGet function)
- hostQuery A query string to send to a MOM.

Adds a 3-type (archType, nodeAttr, hostQuery) into the internal Res table. If (archType, nodeAttr, ...) is duplicated, then only the hostQuery portion is updated.

findResPtrGivenNodeAttr

```
static int findResPtrGivenNodeAttr(struct Resource **resptrs, char *nodeAttr)
```

Args:

resptrs Ptr to a table of Resource pointers.

nodeAttrA node attribute (or even CNodeGet function)

Returns the index to resptrs that contain 'nodeAttr'. Otherwise, -1 is returned.

getResPtr

```
static struct Resource **getResPtr(char *archType, char *nodeAttr)
```

Args:

archType A class type of a resource (could be a node name or os type)

nodeAttr A node attribute (or even CNodeGet function)

Returns an array of pointers to the internal Resource table, Res containing entries that match (archType, nodeAttr,,) with a non-NULLSTR or non-"" entry for hostQuery_keyword. This will also match any "*" entry for 'archType.

getNodeAttrGivenResPtr

```
char *getNodeAttrGivenResPtr(struct Resource *resptr)
```

Args:

resptr A pointer to a struct Resource entry.

Returns 'nodeAttr' value of the entry pointed to by 'resptr'.

getHostQueryKeywordGivenResPtr

```
char *getHostQueryKeywordGivenResPtr(struct Resource *resptr)
```

Args:

resptr A pointer to a struct Resource entry.

Returns 'hostQuery_keyword' value of the entry pointed to by 'resptr'.

ResPrint

```
void ResPrint(void)
```

Prints out the entries of the internal Resource table, Res.

ResFree

```
void ResFree(void)
```

Free up all malloc-ed storage associated with the internal Resource table Res.

6.1.4.4. Job

The source code found under the *Job* subdirectory contains data structures and functions that are used by the Job abstraction. The files involved are af_job.h and af_job.c. The main data structures used are:

```
  struct job_struct {
    char *jobId;
    char *jobName;
    char *ownerName;
    char *effectiveUserName;  /* username to execute job under */
    char *effectiveGroupName; /* group to execute job under */
    int  state;
    int  priority;
    int  rerunFlag;          /* rerunnable attribute */
    int  interactiveFlag;    /* is job interactive ? */
    DateTime dateTimeCreated;
    char *emailAddr;         /* for notification of job status */
    char *stageinFiles;
    char *stageoutFiles;
    struct IntRes *intResReq;
    struct SizeRes *sizeResReq;
    struct StringRes *stringResReq;
    struct IntRes *intResUse;
    struct SizeRes *sizeResUse;
    struct StringRes *stringResUse;
    void   *server;          /* needed in order to run a job; need to */
                             /* instruct the appropriate server to run */
                             /* the job it owns */
    void    *queue;          /* needed in order to accumulate */
                             /* certain Que resources based on */
                             /* resource value for job. */

    int    refCnt;           /* # of link references to this struct - only */
                             /* used to determine if this job struct should be */
                             /* freed */
  };
  typedef struct job_struct Job;

  struct SetJobElement {
    struct SetJobElement *nextptr;
    struct SetJobElement *first; /* pointer to the first element in Set */
  };

  struct setJob_struct {
    struct SetJobElement  *head;
    struct SetJobElement  *tail; /* non-NULL tail */
  };
  typedef struct setJob_struct SetJob;
```

6.1.4.4.1. File: af_job.c

JobIdGet

```
char *JobIdGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'jobId' attribute value of 'job'.

JobNameGet

```
char *JobNameGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'jobName' attribute value of 'job'.

JobOwnerNameGet

```
char *JobOwnerNameGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'ownerName' attribute value of 'job'.

JobEffectiveUserNameGet

```
char *JobEffectiveUserNameGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'effectiveUserName' attribute value of 'job'.

JobEffectiveGroupNameGet

```
char *JobEffectiveGroupNameGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'effectiveGroupName' attribute value of 'job'.


```
int JobStateGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'state' attribute value of 'job'.


```
int JobPriorityGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'priority' attribute value of 'job'.


```
int JobRerunFlagGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'rerunFlag' attribute value of 'job'.


```
int JobInteractiveFlagGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'interactiveFlag' attribute value of 'job'.


```
DateTime JobDateTimeCreatedGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'dateTimeCreated' attribute value of 'job'.

JobEmailAddrGet

```
char *JobEmailAddrGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'emailAddr' attribute value of 'job'.

JobServerGet

```
void *JobServerGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'server' attribute value of 'job'.

JobRefCntGet

```
int JobRefCntGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'refCnt' attribute value of 'job'.

JobStageinFilesGet

```
char *JobStageinFilesGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'stageinFiles' attribute value of 'job'.

JobStageoutFilesGet

```
char *JobStageoutFilesGet(Job *job)
```

Args:

job A pointer to a Job object.

Returns 'stageoutFiles' attribute value of 'job'.

JobIntResReqGet

```
int JobIntResReqGet(Job *job, char *name)
```

Args:

job A pointer to a Job object.

name A resource name.

Returns 'intResReq->name' attribute value of 'job'.

JobSizeResReqGet

```
Size JobSizeResReqGet(Job *job, char *name)
```

Args:

job A pointer to a Job object.

name A resource name.

Returns 'sizeResReq->name' attribute value of 'job'.

JobStringResReqGet

```
char *JobStringResReqGet(Job *job, char *name)
```

Args:

job A pointer to a Job object.

name A resource name.

Returns 'stringResReq->name' attribute value of 'job'.

JobIntResUseGet

```
int JobIntResUseGet(Job *job, char *name)
```

Args:

job A pointer to a Job object.

name A resource name.

Returns 'intResUse->name' attribute value of 'job'.

JobSizeResUseGet

```
Size JobSizeResUseGet(Job *job, char *name)
```

Args:

job A pointer to a Job object.

name A resource name.

Returns 'sizeResUse->name' attribute value of 'job'.

JobStringResUseGet

```
char *JobStringResUseGet(Job *job, char *name)
```

Args:

job A pointer to a Job object.

name A resource name.

Returns 'stringResUse->name' attribute value of 'job'.

JobIdPut

```
void JobIdPut(Job *job, char *jobId)
```

Args:

job A pointer to a Job object.

jobId A job id.

Set job->jobId attribute value to 'jobId'.

JobNamePut

```
void JobNamePut(Job *job, char *jobName)
```

Args:

job A pointer to a Job object.

jobNameA job name.

Set job->jobName attribute value to 'jobName'.

JobOwnerNamePut

```
void JobOwnerNamePut(Job *job, char *ownerName)
```

Args:

job A pointer to a Job object.

ownerName
A job's ownername.

Set job->jobOwner attribute value to 'ownerName'.

JobEffectiveUserNamePut

```
void JobEffectiveUserNamePut(Job *job, char *euser)
```

Args:

job A pointer to a Job object.

ownerName
A job's effective username.

Set job->effectiveUserName attribute value to 'euser'.

JobEffectiveGroupNamePut

```
void JobEffectiveGroupNamePut(Job *job, char *groupName)
```

Args:

job A pointer to a Job object.

groupName
A job's effective groupname.

Set job->effectiveGroupName attribute value to 'groupName'.

JobStatePut

```
void JobStatePut(Job *job, int state)
```

Args:

job A pointer to a Job object.

state A job's state.

Set job->state attribute value to 'state'.

JobPriorityPut

```
void JobPriorityPut(Job *job, int priority)
```

Args:

job A pointer to a Job object.

priority A job's priority.

Set job->priority attribute value to 'priority'.

JobRerunFlagPut

```
void JobRerunFlagPut(Job *job, int rerunFlag)
```

Args:

job A pointer to a Job object.

rerunFlag
A job's priority.

Set job->rerunFlag attribute value to 'rerunFlag'.

JobInteractiveFlagPut

```
void JobRerunFlagPut(Job *job, int interactiveFlag)
```

Args:

job A pointer to a Job object.

interactiveFlag
Is job interactive?

Set job->interactiveFlag attribute value to 'interactiveFlag'.

JobDateTimeCreatedPut

```
void JobDateTimeCreatedPut(Job *job, DateTime cdate)
```

Args:

job A pointer to a Job object.

interactiveFlag
Is job interactive?

Set job->dateTimeCreated attribute value to 'cdate'.

JobEmailAddrPut

```
void JobEmailAddrPut(Job *job, char *emailAddr)
```

Args:

job A pointer to a Job object.

emailAddr
Email address to notify of job status.

Set job->emailAddr attribute value to 'emailAddr'.

JobServerPut

```
void JobServerPut(Job *job, void *server)
```

Args:

job A pointer to a Job object.
server Server owner of the job.

Set job->server attribute value to 'server'.

JobRefCntPut

```
void JobRefCntPut(Job *job, int refCnt)
```

Args:

job A pointer to a Job object.
refCnt # of link references to the Job struct.

Set job->refCnt attribute value to 'refCnt'.

JobStageinFilesPut

```
void JobStageinFilesPut(Job *job, char *stagein)
```

Args:

job A pointer to a Job object.
stagein The list of files to stagein.

Set job->stageinFiles attribute value to 'stagein'.

JobStageoutFilesPut

```
void JobStageoutFilesPut(Job *job, char *stageout)
```

Args:

job A pointer to a Job object.
stageout The list of files to stageout.

Set job->stageoutFiles attribute value to 'stageout'.

JobIntResReqPut

```
void JobIntResReqPut(Job *job, char *name, int value)
```

Args:

- job A pointer to a Job object.
- name A resource name.
- value A resource value.

Set intResReq->name attribute value of 'job' to 'value'.

JobSizeResReqPut

```
void JobSizeResReqPut(Job *job, char *name, Size value)
```

Args:

- job A pointer to a Job object.
- name A resource name.
- value A resource value.

Set sizeResReq->name attribute value of 'job' to 'value'.

JobStringResReqPut

```
void JobStringResReqPut(Job *job, char *name, char *value)
```

Args:

- job A pointer to a Job object.
- name A resource name.
- value A resource value.

Set stringResReq->name attribute value of 'job' to 'value'.

JobIntResUsePut

```
void JobIntResUsePut(Job *job, char *name, int value)
```

Args:

- job A pointer to a Job object.
- name A resource name.
- value A resource value.

Set intResUse->name attribute value of 'job' to 'value'.

JobSizeResUsePut

```
void JobSizeResUsePut(Job *job, char *name, Size value)
```

Args:

job	A pointer to a Job object.
name	A resource name.
value	A resource value.

Set sizeResUse->name attribute value of 'job' to 'value'.

JobStringResUsePut

```
void JobStringResUsePut(Job *job, char *name, char *value)
```

Args:

job	A pointer to a Job object.
name	A resource name.
value	A resource value.

Set stringResUse->name attribute value of 'job' to 'value'.

JobInit

```
void JobInit(Job *job)
```

Args:

job	A pointer to a Job object.

Initialize the members of the Job object to consistent values.

JobPrint

```
void JobPrint(Job *job)
```

Args:

job	A pointer to a Job object.

Prints out the values to the members of the 'job'.

JobFree

```
void JobFree(Job *job)
```

Args:

job A pointer to a Job object.

Frees up malloc-ed areas associated with 'job'.

SetJobInit

```
void SetJobInit(SetJob *sjob)
```

Args:

sjob A pointer to a set of jobs.

Initializes the set of jobs, 'sjob', adding a NOJOB end of list record, and forcing the head and tail pointers to point to this end record.

SetJobAdd

```
void SetJobAdd(SetJob *sjob, Job *job)
```

Args:

sjob A pointer to a set of jobs.

job A pointer to a Job object.

Adds 'job' to the set of jobs pointed to by 'sjob'.

SetJobUpdateFirst

```
void SetJobUpdateFirst(SetJob *sjob, struct SetJobElement *first)
```

Args:

sjob A pointer to a set of jobs.

first A pointer to a set of jobs element.

Go through each element of 'sjob' and update each one's first attribute value to 'first'.

SetJobRemove

```
void SetJobRemove(SetJob *sjob, Job *job)
```

Args:

sjob A pointer to a set of jobs.

job A pointer to a Job object.

Delete 'job' from the set of jobs, 'sjob'. The 'job' itself is malloc freed if its refCnt is 0.

SetJobFree

```
void SetJobFree(SetJob *sjob)
```

Args:

sjob A pointer to a set of jobs.

Free up all malloc-ed areas associated with 'sjob'.

SetJobPrint

```
void SetJobPrint(struct SetJobElement *sje)
```

Args:

sje A pointer to a set of jobs element.

Prints out the members of 'sje'.

inSetJob

```
int inSetJob(Job *job, struct SetJobElement *sje)
```

Args:

job A pointer to a Job object.

sje A pointer to a set of jobs element.

Returns TRUE or FALSE depending on whether or not 'job' is in 'sje'.

strToJobState

```
int strToJobState(char *val)
```

Args:

val A string containing: "Q", "R", "T", "H", "E", "W", "D"

Returns the following:

string	value
"Q"	QUEUED
"R"	RUNNING
"T"	TRANSIT
"H"	HELD
"E"	EXITING
"W"	WAITING
"D"	DELETED

firstJobPtr

```
void firstJobPtr(struct SetJobElement **sjeptr, struct SetJobElement *first)
```

Args:

- sjeptr pointer to a pointer to a set of Jobs.
- sje A pointer to a Job element.

Updates the *sjeptr to "first", and then continues to reassign the value of *sjeptr to *sjeptr->nextptr until encountering a non-DELETED Job record.

nextJobPtr

```
void nextJobPtr(struct SetJobElement **sjeptr)
```

Args:

- sjeptr pointer to a pointer to a set of Jobs.

Updates the *sjeptr to *sjeptr->nextptr, and then continues to reassign the value until encountering a non-DELETED Job record.

JobPartition

```
int JobPartition(struct JobSortArgs *A, int p, int r)
```

Args:

- A stuff of information needed to reorder the elements of a set of Jobs.
- p the "leftmost" element of a set of Jobs.
- r the "rightmost" element of a set of Jobs.

This is the Partition() function in the well-known Quicksort() sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

JobQuickSort

```
void JobQuickSort (struct JobSortArgs *A, int p, int r)
```

Args:

- A stuff of information needed to reorder the elements of a set of Jobs.
- p the "leftmost" element of a set of Jobs.
- r the "rightmost" element of a set of Jobs.

This is the Quicksort() function in the well-known quicksort sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

SetJobSortInt

```
int SetJobSortInt (struct SetJobElement *sje, int (*key)(), int order)
```

Args:

- s — the set of Jobs to reorder.
- key — the function to apply to each member of the set of Jobs whose int value will be used to reorder the set of Jobs.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of Jobs in a dynamic array, and then run JobQuicksort() function on the array, which also rearranges the pointers representing the set of Jobs.

SetJobSortStr

```
int SetJobSortStr (struct SetJobElement *sje, char *(*key)(), int order)
```

Args:

- s — the set of Jobs to reorder.
- key — the function to apply to each member of the set of Jobs whose char* value will be used to reorder the set of Jobs.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of Jobs in a dynamic array, and then run JobQuicksort() function on the array, which also rearranges the pointers representing the set of Jobs.

SetJobSortDateTime

```
int SetJobSortDateTime (struct SetJobElement *sje, DateTime (*key)(), int order)
```

Args:

- s — the set of Jobs to reorder.
- key — the function to apply to each member of the set of Jobs whose DateTime value will be used to reorder the set of Jobs.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of Jobs in a dynamic array, and then run JobQuicksort() function on the array, which also rearranges the pointers representing the set of Jobs.

SetJobSortSize

```
int SetJobSortSize (struct SetJobElement *sje, Size (*key)(), int order)
```

Args:

s the set of Jobs to reorder.

key the function to apply to each member of the set of Jobs whose Size value will be used to reorder the set of Jobs.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Jobs in a dynamic array, and then run JobQuicksort() function on the array, which also rearranges the pointers representing the set of Jobs.

SetJobSortFloat

```
int SetJobSortFloat (struct SetJobElment *sje, double (*key)(), int order)
```

Args:

s the set of CNodes to reorder.

key the function to apply to each member of the set of Jobs whose double value will be used to reorder the set of Jobs.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Jobs in a dynamic array, and then run JobQuicksort() function on the array, which also rearranges the pointers representing the set of Jobs.

6.1.4.5. Que

The source code found under the *Que* subdirectory contains data structures and functions that are used by the Que abstraction. The files involved are af_que.h and af_que.c. The main data structures used are:

```
  struct que_struct {
    struct que_struct *nexptr; /* to maintain a list of ques */
    char *name;                /* name of a queue */
    int  type;                 /* type of queue: execution or routing */
    int  numJobs;
    int  priority;            /* priority of this queue against all other */
                              /* queues */
    int  maxRunJobs;          /* maximum # of jobs allowed to be selected */
                              /* from this queue */
    int  maxRunJobsPerUser;
    int  maxRunJobsPerGroup;
    int  state;               /* can jobs from this queue be scheduled for */
                              /* execution? */
    struct IntRes *intResAvail;
    struct IntRes *intResAssign;
    struct SizeRes *sizeResAvail;
    struct SizeRes *sizeResAssign;
    struct StringRes *stringResAvail;
    struct StringRes *stringResAssign;
    SetJob jobs;             /* pointer to head of the job */
  };
  typedef struct que_struct Que;

  struct SetQue_type {
    Que *head;
    Que *tail;
```

```
};
typedef struct SetQue_type SetQue;
```

6.1.4.5.1. File: af_que.c

QueNameGet

```
char *QueNameGet(Que *que)
```

Args:
que A pointer to a Que object.
Returns 'name' attribute value of 'que'.

QueTypeGet

```
int QueTypeGet(Que *que)
```

Args:
que A pointer to a Que object.
Returns 'type' attribute value of 'que'.

QueNumJobsGet

```
int QueNumJobsGet(Que *que)
```

Args:
que A pointer to a Que object.
Returns 'numJobs' attribute value of 'que'.

QuePriorityGet

```
int QuePriorityGet(Que *que)
```

Args:
que A pointer to a Que object.
Returns 'priority' attribute value of 'que'.

QueMaxRunJobsGet

```
int QueMaxRunJobsGet(Que *que)
```

Args:

que A pointer to a Que object.

Returns 'maxRunJobs' attribute value of 'que'.

QueMaxRunJobsPerUserGet

```
int QueMaxRunJobsPerUserGet(Que *que)
```

Args:

que A pointer to a Que object.

Returns 'maxRunJobsPerUser' attribute value of 'que'.

QueMaxRunJobsPerGroupGet

```
int QueMaxRunJobsPerGroupGet(Que *que)
```

Args:

que A pointer to a Que object.

Returns 'maxRunJobsPerGroup' attribute value of 'que'.

QueStateGet

```
int QueStateGet(Que *que)
```

Args:

que A pointer to a Que object.

Returns 'state' attribute value of 'que'.

QueIntResAvailGet

```
int QueIntResAvailGet(Que *que, char *name)
```

Args:

que A pointer to a Que object.

name A resource name.

Returns 'intResAvail->name' attribute value of 'que'.

QueIntResAssignGet

```
int QueIntResAssignGet(Que *que, char *name)
```

Args:

que A pointer to a Que object.

name A resource name.

Returns 'intResAssign->name' attribute value of 'que'.

QueSizeResAvailGet

```
Size QueSizeResAvailGet(Que *que, char *name)
```

Args:

que A pointer to a Que object.

name A resource name.

Returns 'sizeResAvail->name' attribute value of 'que'.

QueSizeResAssignGet

```
Size QueSizeResAssignGet(Que *que, char *name)
```

Args:

que A pointer to a Que object.

name A resource name.

Returns 'sizeResAssign->name' attribute value of 'que'.

QueStringResAvailGet

```
char *QueStringResAvailGet(Que *que, char *name)
```

Args:

que A pointer to a Que object.

name A resource name.

Returns 'stringResAvail->name' attribute value of 'que'.

QueStringResAssignGet

```
char *QueStringResAssignGet(Que *que, char *name)
```

Args:

que A pointer to a Que object.

name A resource name.

Returns 'stringResAssign->name' attribute value of 'que'.

QueJobsGet

```
SetJobElement *QueJobsGet(Que *que)
```

Args:

que A pointer to a Que object.

Returns 'jobs.head' attribute value of 'que'.

QueNamePut

```
void QueNamePut(Que *que, char *queue_name)
```

Args:

que A pointer to a Que object.

queue_name
A new queue name.

Sets the 'name' attribute value to 'queue_name'.

QueNumJobsPut

```
void QueNumJobsPut(Que *que, int numJobs)
```

Args:

que A pointer to a Que object.

numJobsThe # of PBS jobs.

Sets the 'numJobs' attribute value to 'numJobs'.

QueMaxRunJobsPut

```
void QueMaxRunJobsPut(Que *que, int maxRunJobs)
```

Args:

que A pointer to a Que object.

maxRunJobs
Some # of PBS jobs.

Sets the 'maxRunJobs' attribute value to 'maxRunJobs'.

QueMaxRunJobsPerUserPut

```
void QueMaxRunJobsPerUserPut(Que *que, int maxRunJobsPerUser)
```

Args:

que A pointer to a Que object.

maxRunJobsPerUser
Some # of PBS jobs.

Sets the 'maxRunJobsPerUser' attribute value to 'maxRunJobsPerUser'.

QueMaxRunJobsPerGroupPut

```
void QueMaxRunJobsPerGroupPut(Que *que, int maxRunJobsPerGroup)
```

Args:

que A pointer to a Que object.

maxRunJobsPerGroup
Some # of PBS jobs.

Sets the 'maxRunJobsPerGroup' attribute value to 'maxRunJobsPerGroup'.

QuePriorityPut

```
void QuePriorityPut(Que *que, int priority)
```

Args:

que A pointer to a Que object.

priority Priority value of queue against all other queues.

Sets the 'priority' attribute value of 'que' to 'priority'.

QueStatePut

```
void QueStatePut(Que *que, int state)
```

Args:

que A pointer to a Que object.

state State of queue.

Sets the 'state' attribute value of 'que' to 'state'.

QueIntResAvailPut

```
void QueIntResAvailPut(Que *que, char *name, int value)
```

Args:

que A pointer to a Que object.

name A resource name.
value New resource value.

Sets the 'intResAvail->name' attribute value of 'que' to 'value'.

QueIntResAssignPut

```
void QueIntResAssignPut(Que *que, char *name, int value)
```

Args:
que A pointer to a Que object.
name A resource name.
value New resource value.

Sets the 'intResAssign->name' attribute value of 'que' to 'value'.

QueSizeResAvailPut

```
void QueSizeResAvailPut(Que *que, char *name, Size value)
```

Args:
que A pointer to a Que object.
name A resource name.
value New resource value.

Sets the 'sizeResAvail->name' attribute value of 'que' to 'value'.

QueSizeResAssignPut

```
void QueSizeResAssignPut(Que *que, char *name, Size value)
```

Args:
que A pointer to a Que object.
name A resource name.
value New resource value.

Sets the 'sizeResAssign->name' attribute value of 'que' to 'value'.

QueStringResAvailPut

```
void QueStringResAvailPut(Que *que, char *name, char *value)
```

Args:
que A pointer to a Que object.

name A resource name.

value New resource value.

Sets the 'stringResAvail->name' attribute value of 'que' to 'value'.

QueStringResAssignPut

```
void QueStringResAssignPut(Que *que, char *name, char *value)
```

Args:

que A pointer to a Que object.

name A resource name.

value New resource value.

Sets the 'stringResAssign->name' attribute value of 'que' to 'value'.

QueInit

```
void QueInit(Que *que)
```

Args:

que A pointer to a Que object.

Initialize the members of 'que' to have consistent values.

QuePrint

```
void QuePrint(Que *que)
```

Args:

que A pointer to a Que object.

Prints out the members of the Que structure.

QueFree

```
void QueFree(Que *que)
```

Args:

que A pointer to a Que object.

Frees up malloc-ed areas associated with 'que'.

QueJobInsert

```
void QueJobInsert(Que *que, Job *job)
```

Args:

que	A pointer to a Que object.
job	A pointer to a Job object.

Insert 'job' into the set of jobs pool of 'que'.

QueJobDelete

```
void QueJobDelete(Que *que, Job *job)
```

Args:

que	A pointer to a Que object.
job	A pointer to a Job object.

Deletes 'job' from the set of jobs pool of 'que'.

intExpr

```
static Job *intExpr(Job *j, Que *q, int (*func)(), Comp operator,
                                int value, Job **maxj, Job **minj)
```

Args:

j	A pointer to a Job object.
que	A pointer to a Que object.
func	A pointer to a function returning an integer value.
operator	A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN.
value	Some integer value.
maxj	Holder of a return max Job value.
minj	Holder of a return min Job value.

Runs a "func(j) operator value" and if it returns TRUE, then return j. Otherwise, the return value is NOJOB. If operator is OP_MAX, or OP_MIN, then run func() on each job in q, saving in maxj the job with the largest return value, or saving in minj the job with the minimum return value.

strExpr

```
static Job *strExpr(Job *j, Que *q, char *(*strfunc)(), Comp operator,
                                char *valuestr, Job **maxj, Job **minj)
```

Args:

j	A pointer to a Job object.

que A pointer to a Que object.

strfunc A pointer to a function returning a string value.

operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN

valuestr Some string value.

maxj Holder of a return max Job value.

minj Holder of a return min Job value.

Runs a "strfunc(j) operator strvalue" and if it returns TRUE, then return j. Otherwise, the return value is NOJOB. If operator is OP_MAX, or OP_MIN, then run strfunc() on each job in q, saving in maxj the job with the lexicographically largest return value, or saving in minj the job with the lexicographically minimum return value.

dateTimeExpr

```
static Job *dateTimeExpr(Job *j, Que *q, DateTime *(*datetfunc)(),
                         Comp operator, DateTime datet, Job **maxj, Job **minj)
```

Args:

j A pointer to a Job object.

que A pointer to a Que object.

datetfunc A pointer to a function returning a DateTime value.

operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN

datet Some DateTime value.

maxj Holder of a return max Job value.

minj Holder of a return min Job value.

Runs a "datetfunc(j) operator datet" and if it returns TRUE, then return j. Otherwise, the return value is NOJOB. If operator is OP_MAX, or OP_MIN, then run datetfunc() on each job in q, saving in maxj the job with the largest return value, or saving in minj the job with the minimum return value.

sizeExpr

```
static Job *sizeExpr(Job *j, Que *q, Size (*sizefunc)(),
                         Comp operator, Size size, Job **maxj, Job **minj)
```

Args:

j A pointer to a Job object.

que A pointer to a Que object.

sizefunc A pointer to a function returning a Size value.

operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN

size Some Size value.

maxj Holder of a return max Job value.

minj Holder of a return min Job value.

Runs a "sizefunc(j) operator size" and if it returns TRUE, then return j. Otherwise, the return value is NOJOB. If operator is OP_MAX, or OP_MIN, then run sizefunc() on each job in q, saving in maxj the job with the largest return value, or saving in minj the job with the minimum return value.

QueJobFindInt

```
Job *QueJobFindInt(Que *que, ...)
```

Args:

que A pointer to a Que object.

... Variable list of arguments, could be: int (*func)(), Comp operator, int value.

This is basically the front end (user interface) to intExpr().

QueJobFindStr

```
Job *QueJobFindStr(Que *que, ...)
```

Args:

que A pointer to a Que object.

... Variable list of arguments, could be: char *(*strfunc)(), Comp operator, char *valuestr.

This is basically the front end (user interface) to strExpr().

QueJobFindDateTime

```
Job *QueJobFindDateTime(Que *que, ...)
```

Args:

que A pointer to a Que object.

... Variable list of arguments, could be DateTime (*datefunc)(), Comp operator, DateTime datet.

This is basically the front end (user interface) to dateTimeExpr().

QueJobFindSize

```
Job *QueJobFindSize(Que *que, ...)
```

Args:

que A pointer to a Que object.

... Variable list of arguments, could be Size (*sizefunc)(), Comp operator, Size size.

This is basically the front end (user interface) to sizeExpr().

QueFilterInt

```
Que *QueFilterInt(Que *que, int (*func)(), Comp operator, int value)
```

Args:

que A pointer to a Que object.

func A pointer to a function returning an integer value.

operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN.

value Some integer value.

This is another front end (user interface) to intExpr(), but this creates a new queue of jobs that satisfy "func(job) operator value" expression for each job in 'que'.

QueFilterStr

```
Que *QueFilterStr(Que *que, char *(*strfunc)(), Comp operator, char *valuestr)
```

Args:

que A pointer to a Que object.

strfunc A pointer to a function returning a string value.

operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN.

valuestr Some string value.

This is another front end (user interface) to strExpr(), but this creates a new queue of jobs that satisfy "strfunc(job) operator valuestr" expression for each job in 'que'.

QueFilterDateTime

```
Que *QueFilterDateTime(Que *que, DateTime (*datefunc)(),
                           Comp operator, DateTime datet)
```

Args:

que A pointer to a Que object.

datefunc A pointer to a function returning a DateTime value.

operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN.

datet Some DateTime value.

This is another front end (user interface) to dateTimeExpr(), but this creates a new queue of jobs that satisfy "datefunc(job) operator datet" expression for each job in 'que'.

QueFilterSize

```
Que *QueFilterSize(Que *que, Size (*sizefunc)(),
                          Comp operator, Size size)
```

Args:

- que A pointer to a Que object.
- sizefunc A pointer to a function returning a Size value.
- operator A compare operator: OP_EQ, OP_NEQ, OP_GT, OP_GE, OP_LE, OP_LT, OP_MAX, OP_MIN.
- size Some Size value.

This is another front end (user interface) to sizeExpr(), but this creates a new queue of jobs that satisfies "sizefunc(job) operator size" expression for each job in 'que'.

SetQueInit

```
void SetQueInit(SetQue *sq)
```

Args:

- sq A pointer to a set of queues object.

Initializes 'sq' so that both head and tail of the list are pointing to NOQUE.

SetQueAdd

```
void SetQueAdd(SetQue *sq, Que *q)
```

Args:

- sq A pointer to a set of queues object.
- q New queue to add to the set.

Adds 'q' to the set of queues, 'sq'. Malloc table is updated since 'q' is a malloc-ed area.

SetQueFree

```
void SetQueFree(SetQue *sq)
```

Args:

- sq A pointer to a set of queues object.

Frees up all storage associated with 'sq' and then re-initializes it.

SetQueFindQueByName

```
Que *SetQueFindQueByName(SetQue *sq, char *queue_name)
```

Args:

sq A pointer to a set of queues object.

queue_name
A name of a queue to search for.

Returns the que in 'sq' whose name is 'queue_name'.

SetQuePrint

```
void SetQuePrint(SetQue *sq)
```

Args:

sq A pointer to a set of queues object.

Prints out the elements in the set of queues, 'sq'.

inSetQue

```
int inSetQue(Que *que, SetQue *sq)
```

Args:

que A queue to search for.

sq A pointer to a set of queues object.

Returns 1 if 'que' is a member of 'sq'; 0 otherwise.

QuePartition

```
int QuePartition(struct QueSortArgs *A, int p, int r)
```

Args:

A stuff of information needed to reorder the elements of a set of Ques.

p the "leftmost" element of a set of Ques.

r the "rightmost" element of a set of Ques.

This is the Partition() function in the well-known Quicksort() sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

QueQuickSort

```
void QueQuickSort (struct QueSortArgs *A, int p, int r)
```

Args:

- A stuff of information needed to reorder the elements of a set of Ques.
- p the "leftmost" element of a set of Ques.
- r the "rightmost" element of a set of Ques.

This is the Quicksort() function in the well-known quicksort sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

SetQueSortInt

```
int SetQueSortInt (SetQue *s, int (*key)(), int order)
```

Args:

- s the set of Ques to reorder.
- key the function to apply to each member of the set of Ques whose int value will be used to reorder the set of Ques.
- order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Ques in a dynamic array, and then run QueQuicksort() function on the array, which also rearranges the pointers representing the set of Ques.

SetQueSortStr

```
int SetQueSortStr (SetQue *s, char *(*key)(), int order)
```

Args:

- s the set of Ques to reorder.
- key the function to apply to each member of the set of Ques whose char* value will be used to reorder the set of Ques.
- order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Ques in a dynamic array, and then run QueQuicksort() function on the array, which also rearranges the pointers representing the set of Ques.

SetQueSortDateTime

```
int SetQueSortDateTime (SetQue *s, DateTime (*key)(), int order)
```

Args:

- s the set of Jobs to reorder.
- key the function to apply to each member of the set of Ques whose DateTime value will be used to reorder the set of Ques.
- order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Ques in a dynamic array, and then run QueQuicksort() function on the array, which also rearranges the pointers representing the set of Ques.

SetQueSortSize

```
int SetQueSortSize (SetQue *s, Size (*key)(), int order)
```

Args:

- s — the set of Ques to reorder.
- key — the function to apply to each member of the set of Ques whose Size value will be used to reorder the set of Ques.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of Ques in a dynamic array, and then run QueQuicksort() function on the array, which also rearranges the pointers representing the set of Ques.

SetQueSortFloat

```
int SetQueSortFloat (SetQue *s, double (*key)(), int order)
```

Args:

- s — the set of Ques to reorder.
- key — the function to apply to each member of the set of Ques whose double value will be used to reorder the set of Ques.
- order — the order of sort: ASCending or DESCending.

Holds the pointers to the set of Ques in a dynamic array, and then run QueQuicksort() function on the array, which also rearranges the pointers representing the set of Ques.

6.1.4.6. Server

The source code found under the *Server* subdirectory contains data structures and functions that are used by the Server abstraction. The files involved are af_server.h and af_server.c. The main data structures used are:

```
  struct server_struct {
    struct server_struct *nexptr; /* to maintain a list of servers */
    char *inetAddr;               /* hostname of the server */
    int  portNumberOneWay;   /* scheduler <-- server */
                             /* if set to 0, use PBS_SCHEDULER_SERVICE_PORT */
    int  portNumberTwoWay;   /* scheduler <-> server */
                             /* if set to 0, use PBS_BATCH_SERVICE_PORT_DIS */
    int  socket;             /* socket file descriptor */
    int  fdOneWay;           /* fd to use when only receiving messages from */
                             /* the Server */
    int  fdTwoWay;           /* fd to use when sending messages to and */
                             /* receiving messages from the Server. -1 if */
                             /* not connected. */
    int  state;
    int  maxRunJobs;         /* on this server */
    int  maxRunJobsPerUser;
    int  maxRunJobsPerGroup;
    char *defQue;           /* server's default queue */
    struct IntRes *intResAvail;
    struct IntRes *intResAssign;
```

```
    struct SizeRes *sizeResAvail;
    struct SizeRes *sizeResAssign;
    struct StringRes *stringResAvail;
    struct StringRes *stringResAssign;
    SetQue queues;         /* queues managed by the server */
    SetCNode nodes;
  };
  typedef struct server_struct Server;

  struct SetServer_type {
    Server *head;
    Server *localhost;
    Server *tail;
  };
  typedef struct SetServer_type SetServer;

  SetServer AllServers;   /* list of Servers known to the system */
```

6.1.4.6.1. File: af_server.h

In this file, the special structures `node_alist, serv_alist, que_alist, job_alist` hold the attribute values that will be queried from the Server. `ServerAttrInfo` hold the mappings for the job attribute names and Job*Put() functions. `accumTable` hold the list of resources that must be accumulated and Job*Put() functions.

6.1.4.6.2. File: af_server.c

pbserror

```
static char *pbserror(void)
```

This returns the message string associated with the current value of the global variable pbs_errno.

socket_to_conn

```
static int socket_to_conn(int sock)
```

Args:

sock A socket number.

Updates some internal table to convert opened 'sock' into a connection.

get_4byte

```
static int get_4byte(int sock, unsigned long *val)
```

Args:

sock A socket to read from.

val The return integer from socket.

Read and return a 4 byte integer from the network. Returns the (unsigned long) integer in *val. The function return is 0 for EOF, +1 for success, or -1 if error.

updateServerJobInfo

```
static void updateServerJobInfo(Job *job, char *name, char *res, char *value)
```

Args:

job the job

name name of an attribute to update for job.

res name of a resource to update for job.

value new value of attribute, resource for job.

This function consults the ServerAttrInfo[] table for updating the appropriate attribute,resource=value for Job.

inAccumTable

```
int inAccumTable(char *resName)
```

Args:

resNamename of a resource.

This function consults the accumTable[] and return TRUE if resName is in the table; otherwise, returns FALSE.

accumRes

```
int accumRes(Job *job)
```

Args:

job Pointer to a job structure.

This functions looks into the resources_required.* resources, and for any resource name that is found in accumTable[], the corresponding values for Server and Queue (those owning the job) are updated.

ServerInetAddrGet

```
char *ServerInetAddrGet(Server *server)
```

Args:

server A pointer to a Server object.
Returns inetAddr attribute value of 'server'.

ServerDefQueGet

```
char *ServerDefQueGet(Server *server)
```

Args:
server A pointer to a Server object.
Returns defQue attribute value of 'server'.

ServerSocketGet

```
int ServerSocketGet(Server *server)
```

Args:
server A pointer to a Server object.
Returns socket attribute value of 'server'.

ServerPortNumberOneWayGet

```
int ServerPortNumberOneWayGet(Server *server)
```

Args:
server A pointer to a Server object.
Returns portNumberOneWay attribute value of 'server'.

ServerPortNumberTwoWayGet

```
int ServerPortNumberTwoWayGet(Server *server)
```

Args:
server A pointer to a Server object.
Returns portNumberTwoWay attribute value of 'server'.

ServerFdTwoWayGet

```
int ServerFdTwoWayGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns fdTwoWay attribute value of 'server'.

ServerFdOneWayGet

```
int ServerFdOneWayGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns fdOneWay attribute value of 'server'.

ServerStateGet

```
int ServerStateGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns state attribute value of 'server'.

ServerMaxRunJobsGet

```
int ServerMaxRunJobsGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns maxRunJobs attribute value of 'server'.

ServerMaxRunJobsPerUserGet

```
int ServerMaxRunJobsPerUserGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns maxRunJobsPerUser attribute value of 'server'.

ServerMaxRunJobsPerGroupGet

```
int ServerMaxRunJobsPerGroupGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns maxRunJobsPerGroup attribute value of 'server'.

ServerQueuesGet

```
int ServerQueuesGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns the pointer to server->queues attribute of 'server'.

ServerJobsGet

```
int ServerJobsGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns the jobs.head value of 'server'.

ServerIntResAvailGet

```
int ServerIntResAvailGet(Server *server, char *name)
```

Args:

server A pointer to a Server object.

name A resource name.

Returns intResAvail->name attribute value of 'server'.

ServerIntResAssignGet

```
int ServerIntResAssignGet(Server *server, char *name)
```

Args:

server A pointer to a Server object.

name A resource name.

Returns intResAssign->name attribute value of 'server'.

ServerSizeResAvailGet

```
Size ServerSizeResAvailGet(Server *server, char *name)
```

Args:

server A pointer to a Server object.

name A resource name.

Returns sizeResAvail->name attribute value of 'server'.

ServerSizeResAssignGet

```
Size ServerSizeResAssignGet(Server *server, char *name)
```

Args:

server A pointer to a Server object.

name A resource name.

Returns sizeResAssign->name attribute value of 'server'.

ServerStringResAvailGet

```
char *ServerStringResAvailGet(Server *server, char *name)
```

Args:

server A pointer to a Server object.

name A resource name.

Returns stringResAvail->name attribute value of 'server'.

ServerStringResAssignGet

```
char *ServerStringResAssignGet(Server *server, char *name)
```

Args:

server A pointer to a Server object.

name A resource name.

Returns stringResAssign->name attribute value of 'server'.

ServerInetAddrPut

```
void ServerInetAddrPut(Server *server, char *server_name)
```

Args:

server A pointer to a Server object.

server_name
Name of the server.

Sets the inetAddr attribute value to 'server_name'.

ServerDefQuePut

```
void ServerDefQuePut(Server *server, char *queue_name)
```

Args:

server A pointer to a Server object.

server_name
Name of the default queue.

Sets the defQue attribute value to 'queue_name'.

ServerPortNumberOneWayPut

```
void ServerPortNumberOneWayPut(Server *server, int port)
```

Args:

server A pointer to a Server object.

port Port number.

Sets the portNumberOneWay attribute value to 'port'.

ServerPortNumberTwoWayPut

```
void ServerPortNumberTwoWayPut(Server *server, int port)
```

Args:

server A pointer to a Server object.

port Port number.

Sets the portNumberTwoWay attribute value to 'port'.

ServerSocketPut

```
void ServerSocketPut(Server *server, int fd)
```

Args:

server A pointer to a Server object.

fd A file descriptor.

Sets the socket attribute value to 'fd'.

ServerFdTwoWayPut

```
void ServerFdTwoWayPut(Server *server, int fd)
```

Args:

server A pointer to a Server object.

fd A file descriptor.

Sets the fdTwoWay attribute value to 'fd'.

ServerFdOneWayPut

```
void ServerFdOneWayPut(Server *server, int fd)
```

Args:

server A pointer to a Server object.

fd A file descriptor.

Sets the fdOneWay attribute value to 'fd'.

ServerStatePut

```
void ServerStatePut(Server *server, int state)
```

Args:

server A pointer to a Server object.

state A server state.

Sets the state attribute value of 'server' to 'state'.

ServerMaxRunJobsPut

```
void ServerMaxRunJobsPut(Server *server, int maxRunJobs)
```

Args:

server A pointer to a Server object.

maxRunJobs
# of jobs.

Sets the maxRunJobs attribute value of 'server' to 'maxRunJobs'.

ServerMaxRunJobsPerUserPut

```
void ServerMaxRunJobsPerUserPut(Server *server, int maxRunJobsPerUser)
```

Args:

server A pointer to a Server object.

maxRunJobsPerUser
# of jobs.

Sets the maxRunJobsPerUser attribute value of 'server' to 'maxRunJobsPerUser'.

ServerMaxRunJobsPerGroupPut

```
void ServerMaxRunJobsPerGroupPut(Server *server, int maxRunJobsPerGroup)
```

Args:

server A pointer to a Server object.

maxRunJobsPerGroup
# of jobs.

Sets the maxRunJobsPerGroup attribute value of 'server' to 'maxRunJobsPerGroup'.

ServerIntResAvailPut

```
void ServerIntResAvailPut(Server *server, char *name, int value)
```

Args:

server A pointer to a Server object.

name Resource name.

value Resource value.

Sets the intResAvail->name attribute value of 'server' to 'value'.

ServerIntResAssignPut

```
void ServerIntResAssignPut(Server *server, char *name, int value)
```

Args:

server A pointer to a Server object.

name Resource name.

value Resource value.

Sets the intResAssign->name attribute value of 'server' to 'value'.

ServerSizeResAvailPut

```
void ServerSizeResAvailPut(Server *server, char *name, Size value)
```

Args:

server A pointer to a Server object.

name Resource name.

value Resource value.

Sets the sizeResAvail->name attribute value of 'server' to 'value'.

ServerSizeResAssignPut

```
void ServerSizeResAssignPut(Server *server, char *name, Size value)
```

Args:

server A pointer to a Server object.

name Resource name.

value Resource value.

Sets the sizeResAssign->name attribute value of 'server' to 'value'.

ServerStringResAvailPut

```
void ServerStringResAvailPut(Server *server, char *name, char *value)
```

Args:

server A pointer to a Server object.

name Resource name.

value Resource value.

Sets the stringResAvail->name attribute value of 'server' to 'value'.

ServerStringResAssignPut

```
void ServerStringResAssignPut(Server *server, char *name, char *value)
```

Args:

server A pointer to a Server object.

name Resource name.

value Resource value.

Sets the stringResAssign->name attribute value of 'server' to 'value'.

ServerPrint

```
void ServerPrint(Server *server)
```

Args:

server A pointer to a Server object.

Prints out values to the Server structure.

ServerInit2

```
static void ServerInit2(Server *server)
```

Args:

server A pointer to a Server object.

Initializes all members of the server structure except inetAddr, the port numbers, socket, and file descriptors.

ServerInit

```
void ServerInit(Server *server)
```

Args:

server A pointer to a Server object.

Initializes all members of the server structure.

ServerOpenInit

```
int ServerOpenInit(Server *server)
```

Args:

server A pointer to a Server object.

The algorithm is as follows:

```
  get network address of 'server'.
  create a new socket.
  bind the socket to a local port.
  listen to the local port for incoming messages/request.
  Update the socket attribute of the 'server'.
  Return 0 if everything's okay; 1 otherwise.
```

ServerOpen

```
int ServerOpen(Server *server)
```

Args:

server A pointer to a Server object.

The algorithm is as follows:

```
get the socket of the 'server' (set in the 'socket' attribute).
If socket is not a valid value (no socket exists),
then
   send a pbs_connect() call using the server name alone.
   Use the file descriptor value obtained from pbs_connect() as the new value
     to fdTwoWay attribute.
else
   accept() a request from the server using the socket.
   Check to make sure that the request came from a trusted host.
   Use the descriptor value returned by accept() call as the new value to
     fdOneWay().
   Call socket_to_conn() to obtain another descriptor to be used as the
     value of the fdTwoWay attribute.

Return 0 if everything's ok; 1 otherwise.
```

ServerRead

```
int ServerRead(Server *server)
```

Args:

server	A pointer to a Server object.

Get a scheduling command from the 'server'.

ServerWriteRead

```
int ServerWriteRead(Server *server, int msg, void *param)
```

Args:

server	A pointer to a Server object.
msg	Type of message to send to the server.
param	Additional parameters to accompany the message to send to the server.

```
If msg is STATNODE, then issue a pbs_statnode() call using the fdTwoWay
    attribute value as file descriptor.
If msg is STATSERV, then issue a pbs_statserver() call using the fdTwoWay
    attribute value as file descriptor.
If msg is STATQUE, then issue a pbs_statque() call using the fdTwoWay
    attribute value as file descriptor.
If msg is STATJOB, then issue a pbs_statjob() call using the fdTwoWay
    attribute value as file descriptor.
```

ServerClose

```
int ServerClose(Server *server)
```

Args:

server A pointer to a Server object.

Issues a pbs_disconnect() on the descriptor given by the fdTwoWay attribute. Returns the value obtained from pbs_disconnect().

ServerCloseFinal

```
void ServerCloseFinal(Server *server)
```

Args:

server A pointer to a Server object.

Get any opened socket for the 'server'. Clean up any remaining request on it. Then close the socket.

getNodesInfo

```
static int getNodesInfo(Server *server)
```

Args:

server A pointer to a Server object.

Issue a ServerWriteRead() using node_alist as param value. Get the results, and fill out the appropriate member of the CNode structure, and add it to the set of nodes known to 'server'. Returns 0 if successful; non-zero otherwise.

getServerInfo

```
static int getServerInfo(Server *server)
```

Args:

server A pointer to a Server object.

Issue a ServerWriteRead() using serv_alist as param value. Get the results and fill out the appropriate member of the Server structure. Returns 0 if successful; non-zero otherwise.

getQueuesInfo

```
static int getQueuesInfo(Server *server)
```

Args:

server A pointer to a Server object.

Issue a ServerWriteRead() using que_alist as param value. Get the results and fill out the appropriate member of the Server structure. Returns 0 if successful; non-zero otherwise.

getJobsInfo

```
static int getJobsInfo(Server *server)
```

Args:

server A pointer to a Server object.

Issue a ServerWriteRead() using job_alist as param value. Get the results and fill out the appropriate member of the Server structure. Returns 0 if successful; non-zero otherwise.

ServerFree2

```
static void ServerFree2(Server *server)
```

Args:

server A pointer to a Server object.

Free up all temporary strings associated with 'server'. Free up malloc-ed areas associated with 'server'. Free up all dynamic strings, and job structures associated with the 'server's queues.

ServerFree

```
static void ServerFree(Server *server)
```

Args:

server A pointer to a Server object.

Free up the entire Server structure, and any malloc-ed areas associated with it.

ServerStateRead

```
void ServerStateRead(Server *server)
```

Args:

server A pointer to a Server object.

Deallocates all queuest and jobs associated with 'server'. If fdTwoWay attribute valid is invalid, then issue a ServerOpen(). Get server, nodes, queues, and jobs info from the 'server'.

AllNodesGet

```
SetCNode *AllNodesGet()
```

Args:

Returns ServerNodesGet(AllServersLocalHostGet))

AllNodesLocalHostGet

```
SetCNode *AllNodesLocalHostGet()
```

Args:

Returns the CNode associated with the local host.

ServerNodesGet

```
SetCNode *ServerNodesGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns the set of nodes managed by 'server'.

ServerNodesAdd

```
CNode *ServerNodesAdd(Server *server, char *name, int port, int queryMom)
```

Args:

server A pointer to a Server object.

name name of a node to add.

port port number of the associated MOM

queryMomflag as to whether or not to query the corresponding MOM.

```
If node with 'name' is already in the set of nodes managed by 'server', then
    no need to add.
Otherwise,
    create a new CNode object, initialize it, propagate any resmom information,
      and add the new oobject to the list of nodes known to the 'server'.
```

ServerNodesHeadGet

```
CNode *ServerNodesHeadGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns the first CNode object in the list of nodes known to 'server'.

ServerNodesTailGet

```
CNode *ServerNodesTailGet(Server *server)
```

Args:

server A pointer to a Server object.

Returns the last CNode object in the list of nodes known to 'server'.

ServerNodesQuery

```
int ServerNodesQuery(Server *server, char *spec)
```

Args:

server A pointer to a Server object.

spec A nodes specification

Calls pbs_rescquery() to issue a request to 'server' to query for availability of resources as specified in 'spec'. After getting the results, this calls ServerNodesNumAvailPut(), ServerNodesNumAllocPut(), ServerNodesNumRsvdPut(), and ServerNodesNumDownPut().

Then, it will return SUCCESS or FAIL depending on the results.

ServerNodesReserve

```
int ServerNodesReserve(Server *server, char *spec, int resId)
```

Args:

server A pointer to a Server object.

spec A nodes specification

resId A handle to the reservation

Calls pbs_rescreserve() to issue a request to 'server' to reserve resources specified in 'spec'. If 'resId' is zero, then this is for a new reservation. Otherwise, it is for an existing or partial reservation.

Then, it will return SUCCESS or FAIL depending on the results.

ServerNodesRelease

```
int ServerNodesRelease(Server *server, int resId)
```

Args:

server A pointer to a Server object.

resId A handle to the reservation

Calls pbs_rescrelease() to ssue a request to 'server' to release resources from a previous reservation session whose handle is 'resId'.

Then, it will return SUCCESS or FAIL depending on the results.

ServerNodesNumAvailGet

```
int ServerNodesNumAvailGet(Server *server)
```

Args:

server A pointer to a Server object.

Return the numAvail attribute value of the nodes attribute of 'server'.

ServerNodesNumAllocGet

```
int ServerNodesNumAllocGet(Server *server)
```

Args:

server A pointer to a Server object.

Return the numAlloc attribute value of the nodes attribute of 'server'.

ServerNodesNumRsvdGet

```
int ServerNodesNumRsvdGet(Server *server)
```

Args:

server A pointer to a Server object.

Return the numRsvd attribute value of the nodes attribute of 'server'.

ServerNodesNumDownGet

```
int ServerNodesNumDownGet(Server *server)
```

Args:

server A pointer to a Server object.

Return the numDown attribute value of the nodes attribute of 'server'.

ServerNodesNumAvailPut

```
void ServerNodesNumAvailPut(Server *server, int numAvail)
```

Args:

server A pointer to a Server object.

numAvail # of nodes available

Update the numAvail value of the nodes attribute of 'server' to 'numAvail'.

ServerNodesNumAllocPut

```
void ServerNodesNumAllocPut(Server *server, int numAlloc)
```

Args:

server A pointer to a Server object.

numAlloc # of nodes allocated

Update the numAlloc value of the nodes attribute of 'server' to 'numAlloc'.

ServerNodesNumRsvdPut

```
void ServerNodesNumRsvdPut(Server *server, int numRsvd)
```

Args:

server A pointer to a Server object.

numRsvd # of nodes reserved

Update the numRsvd value of the nodes attribute of 'server' to 'numRsvd'.

ServerNodesNumDownPut

```
void ServerNodesNumDownPut(Server *server, int numDown)
```

Args:

server A pointer to a Server object.

numDown # of nodes down

Update the numDown value of the nodes attribute of 'server' to 'numDown'.

JobAction

```
int JobAction(Job *job, Action action, void *params)
```

Args:

job Pointer to a job object.

action Action to perform on the job: SYNCRUN, ASYNCRUN, DELETE, RERUN, HOLD, RELEASE, SIGNAL, MODIFYATTR, and MODIFYRES.

params Additional parameters to the action.

Depending on the action specified, issue the appropriate PBS API call:

```
  If SYNCRUN, then pbs_runjob(), update the job's state to RUNNING,
                  accumulate the resources.
  If ASYNCRUN, then pbs_asyrunjob(), update the job's state to RUNNING,
                  accumulate the resources.
  If DELETE, then pbs_deljob(), update the job's state to DELETED.
  If RERUN, then pbs_rerunjob(), update the job's state to QUEUED.
  If HOLD, then pbs_holdjob(), update the job's state to HELD.
  If RELEASE, then pbs_rlsjob(), update the job's state to RELEASE.
  If SIGNAL, then pbs_sigjob(),
  If MODIFYRES or MODIFYATTR, then pbs_alterjob(), and update the value
                                  for the appropriate resource or attribute.
Returns SUCCESS (1) if operation was completed successfully; otherwise,
it return FAIL (0).
```

SetServerInit

```
void SetServerInit(SetServer *ss)
```

Args:

ss A set of server structures.

Initialize the set of servers 'ss' so that both head and tail are pointing to NOSERVER.

SetServerAdd

```
void SetServerAdd(SetServer *ss, Server *s)
```

Args:

ss A set of server structures.

s A pointer to a Server object.

Add Server 's' to the set of servers, 'ss'.

SetServerFree

```
void SetServerFree(SetServer *ss)
```

Args:

ss A set of server structures.

Free up malloc-ed areas associated with 'ss'.

SetServerPrint

```
void SetServerPrint(SetServer *ss)
```

Args:

ss A set of server structures.

Prints out the elements in the set of servers 'ss'.


```
int inSetServer(Server *s, SetServer *ss)
```

Args:

s Pointer to a Server object.

ss A set of server structures.

Returns 1 if 's' is a member of 'ss'; 0 otherwise.


```
int AllServersAdd(char *name, int port)
```

Args:

name A node name.

port Network port number for the new Server.

Creates a new Server(name, port) object, and adds it (if not a duplicate) to the internal SetServer variable, AllServers.


```
void AllServersInit(void)
```

Initializes the internal SetServer variable, AllServers to a consistent value.


```
void AllServersGet(void)
```

Returns a pointer to the internal SetServer variable, AllServers.


```
void AllServersFree(void)
```

Frees up malloc-ed storage associated with internal variable, AllServers.

ServerPartition

```
int ServerPartition(struct ServerSortArgs *A, int p, int r)
```

Args:

- A stuff of information needed to reorder the elements of a set of Servers.
- p the "leftmost" element of a set of Servers.
- r the "rightmost" element of a set of Servers.

This is the ServerPartition() function in the well-known Quicksort() sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

ServerQuickSort

```
void ServerQuickSort (struct ServerSortArgs *A, int p, int r)
```

Args:

- A stuff of information needed to reorder the elements of a set of Servers.
- p the "leftmost" element of a set of Servers.
- r the "rightmost" element of a set of Servers.

This is the Quicksort() function in the well-known quicksort sorting algorithm. (see "Introduction to Algorithms" by Cormen, et al) pp. 153-156).

SetServerSortInt

```
int SetServerSortInt (SetServer *s, int (*key)(), int order)
```

Args:

- s the set of Servers to reorder.
- key the function to apply to each member of the set of Servers whose int value will be used to reorder the set of Servers.
- order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Servers in a dynamic array, and then run ServerQuicksort() function on the array, which also rearranges the pointers representing the set of Servers.

SetServerSortStr

```
int SetServerSortStr (SetServer *s, char *(*key)(), int order)
```

Args:

s the set of Servers to reorder.

key the function to apply to each member of the set of Servers whose char* value will be used to reorder the set of Servers.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Ques in a dynamic array, and then run ServerQuickSort() function on the array, which also rearranges the pointers representing the set of Servers.

SetServerSortDateTime

```
int SetServerSortDateTime (SetServer *s, DateTime (*key)(), int order)
```

Args:

s the set of Servers to reorder.

key the function to apply to each member of the set of Servers whose DateTime value will be used to reorder the set of Servers.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Servers in a dynamic array, and then run ServerQuicksort() function on the array, which also rearranges the pointers representing the set of Servers.

SetServerSortSize

```
int SetServerSortSize (SetQue *s, Size (*key)(), int order)
```

Args:

s the set of Ques to reorder.

key the function to apply to each member of the set of Servers whose Size value will be used to reorder the set of Servers.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Servers in a dynamic array, and then run ServerQuickSort() function on the array, which also rearranges the pointers representing the set of Servers.

SetServerSortFloat

```
int SetServerSortFloat (SetServer *s, double (*key)(), int order)
```

Args:

s the set of Servers to reorder.

key the function to apply to each member of the set of Servers whose double value will be used to reorder the set of Servers.

order the order of sort: ASCending or DESCending.

Holds the pointers to the set of Servers in a dynamic array, and then run ServerQuicksort() function on the array, which also rearranges the pointers representing the set of Servers.

6.1.4.7. System

The source code found under the *System* subdirectory contains data structures and functions that are used in order to build the resulting scheduler daemon. It is under this abstraction where the main() part of the program exists. The files involved are af_config.h, af_config.c, af_system.h, and af_system.c.

6.1.4.7.1. File: af_config.c

This contains functions related to the scheduler configuration file.


```
void badconn(char *msg, struct sockaddr_in saddr)
```

Args:

msg A message to attach to the PBS log file regarding a bad connection.

saddr The bad address that attempted to connect to the scheduler.

Sends a message to the PBS log file regarding a bad connection involving 'saddr'.


```
int addClient(char *name)
```

Args:

name A name to add to the list of okClients.

If not a duplicate, add the host address of 'name' to the list of addresses allowed to connect to the scheduler. The list is maintained via the internal variable, okClients. Returns 0 if successful; non-zero otherwise.


```
int validateClient(void *saddr)
```

Args:

saddr A host address to validate.

Returns 0 if saddr's host address appears on the okClients list; otherwise, return non-zero.


```
void freeClients(void)
```

Free up the malloc-ed storage associated with okClients.

freeConfig

```
void freeConfig(void)
```

Free up the malloc-ed storage associated with various arrays filled as a result of reading the configuration file.

getNextToken

```
static char *getNextToken(char *line)
```

Args:

line If non-NULLSTR, a new line to read. Otherwise, continue from previous call.

Returns the next token flagged by a previous call to strtok(). Returns NULL if no more tokens or if an error has occurred.

readConfig

```
int readConfig(char *file)
```

Args:

file File to read.

Read and process the lines in the configuration file. Valid lines format are:
$clienthost <hostname>
$momhost <hostname> <port>
$node <node_name> <CNodeGet() function name> <hostQuery_keyword>

6.1.4.7.2. File: af_config.c

lock_out

```
static void lock_out(int fds, short op)
```

Args:

fds The padlock.

op Lock type.

Prevents other daemons from accessing the file represented by 'fds'.

die

```
static void die(int sig)
```

Args:

sig The signal number.

Causes the scheduler to exit after shutting down the system and closing PBS log file.

initSchedCycle

```
static void initSchedCycle(void)
```

Get all the static resource values for all the known CNodes.

addDefaults

```
static void addDefaults(void)
```

Loads the okClients, and Res internal variables with some default values.

toolong

```
static void toolong(int sig)
```

Args:

sig A signal number.

Parent re-execs itself, while child process attempts to dump core if no core file exists.

restart

```
static void restart(int sig)
```

Args:

sig A signal number.

The algorithm is as follows:

```
  Save some information about the local Server such as PortNumberOneWay,
    PortNumberTwoWay, socket, and fdOneWay.
  Free up all malloc-ed storage filled in when the configuration file was read.
  Add the local server again to the list of known Servers. The saved values of
    PortNumberOneWay, PortNumberTwoWay, socket, and fdOneWay are reloaded into
    the local Server structure.
  Add default values to the internal variables okClients and Res.
  Re-read the configuration file.
  Initialize a scheduling cycle.
```

getArgs

```
static void getArgs(int argc, char *argv[])
```

Args:

- argc # of arguments.
- argv list of the actual arguments.

This function takes care of reading the command line arguments. See BASL ERS for the format of the commandline.

cdToPrivDir

```
static void cdToPrivDir(void)
```

Checks to make sure that the priv directory is not group and other-writeable before cd-ing to it. Pbs_environment file is also checked for security.

secureEnv

```
static void secureEnv(void)
```

Set up a secure executing environment for the scheduler daemon.

signalHandleSet

```
static void signalHandleSet(void)
```

Sets signal handlers for SIGHUP, SIGALRM, SIGINT, and SIGTERM.

SystemInit

```
static void SystemInit(int argc, char *argv[])
```

Args:

- argc # of arguments.
- argv A list of arguments.

The algorithm for this function is as follows:

```
  Check to make sure that effective user id and user id are set to root (when
    not under DEBUG mode)
  Get the local hostname.
  Initialize the set of servers known to the system. Add the local hostname to
    this set.
```

```
  Get command line arguments (via the supplied argc, argv parmeters).
  go to the privilege directory (sched_priv).
  create a secure executing environment.
  Open the PBS sched log file.
  Get a socket from the local server.
  Add defaults to internal Resource variable Res.
  Initialize the set of nodes known to the system.
  Read the configuration file (if set).
  Attempt to open the lock file. If successful, prevent other daemons from
    accessing the file.
  Kill the parent process, causing the child process to be become stand-alone
    daemon.
  Direct stdout/stderr to some debug file (sched_out or as specified in -p)
  Get current process id of the child.
  Close any stdin of the process.
  Write a message to the sched lock file the process id of the daemon.
  Lock the daemon into memory if PLOCK_DAEMONS variable is set appropriately.
  Set up things for DIS data encoding/decoding.
  Set up the signal handlers.
  Initialize a scheduling cycle.
  Write a message to the PBS sched log file indicating that the daemon has
    started.
```

SystemStateRead

```
void SystemStateRead(void (*sched_main)())
```

Args:

sched_main
: A function to invoke during a scheduling cycle.

The algorithm for this function is as follows:

```
  Get the local server's socket.
  Listen on it for messages. If a message has arrived, then go get it.
  if the message received is one of
     {SCH_SCHEDULE_NEW, SCH_SCHEDULE_TERM, SCH_SCHEDULE_TIME,
      SCH_SCHEDULE_RECYC, SCH_SCHEDULE_CMD, SCH_SCHEDULE_FIRST}, then
    begin
      set up an alarm for 'alarm_time'
      Get data for all servers known to the system.
      Get DYNAMIC_RESOURCE data for all nodes known to the system.
      Call sched_main()
      Then disconnect opened connections (2-way channels) to the servers.
      Reset alarm time.
      listen for the next scheduling  message
    end
  else if message received is one of
     {SCH_CONFIGURE, SCH_RULESET}, then
    begin
      Issue a restart() call which will re-read the configuration file.
      listen for the next scheduling  message
    end
  else if message received is one of
```

```
     {SCH_CONFIGURE, SCH_RULESET}, then
    begin
      Issue a restart() call which will re-read the configuration file.
    end
  else if message received is SCH_QUIT, then
      return from this function.
  else if message received is one of
     {SCH_ERROR, SCH_SCHEDULE_NULL},
      listen for the next scheduling  message
  end
```

SystemCloseServers

```
void SystemCloseServers(void)
```

For all the Servers known the the system, close their fdTwoWay file descriptor.

SystemClose

```
void SystemClose(void)
```

Close all file descriptors associated with the Servers known to the system. Free up all malloc-ed areas filled in when the configuration file was read.

6.2. The Tcl Scheduler

The second provided scheduler is based on the Tcl language developed by John K. Ousterhout. Tcl stands for Tool Control Language.

6.2.1. Tcl Scheduler Overview

The Tcl Scheduler contains a number of functions which act as wrappers for existing PBS library calls. The main() routine opens a logfile, processes the command line arguments, and sets up signal handling. After that, a Tcl interpreter is created and Tcl_CreateCommand() is called for each of the function wrappers. The initialization script is run if it exists and the body script is read into memory. A socket is set up to get connections from the server and a loop is entered to process wakeup calls from the server. Each wakeup contains a command from the server. If the command is *SCH_SCHEDULE_NEW*, *SCH_SCHEDULE_TERM*, *SCH_SCHEDULE_TIME*, *SCH_SCHEDULE_RECYC*, or *SCH_SCHEDULE_CMD* the function Tcl_Eval() is called with the saved body script. If the result from this is not *TCL_OK* an error message is logged and the process aborts.

6.2.2. File: pbs_tclWrap.c

The purpose of the wrapper routines is to check the legality of the parameters passed from a Tcl command and call a library function. The **sched_tcl** man page describes each Tcl function in detail. The following is a list of their names with arguments and any special processing that they need to do.

OpenRM()

```
int OpenRM(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "openrm" and calls the PBS resource monitor library function openrm().

CloseRM()

```
int CloseRM(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "closerm" and calls the PBS resource monitor library function closerm().

DownRM()

```
int DownRM(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "downrm" and calls the PBS resource monitor library function downrm().

ConfigRM()

```
int ConfigRM(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "configrm" and calls the PBS resource monitor library function configrm().

AddREQ()

```
int AddREQ(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "addreq" and calls the PBS resource monitor library function addreq().

AllREQ()

```
int AllREQ(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "allreq" and calls the PBS resource monitor library function allreq().

GetREQ()

```
int GetREQ(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "getreq" and calls the PBS resource monitor library function getreq().

FlushREQ()

```
int FlushREQ(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "flushreq" and calls the PBS resource monitor library function flushreq().

ActiveREQ()

```
int ActiveREQ(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "activereq" and calls the PBS resource monitor library function activereq().

FullResp()

```
int FullResp(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "fullresp" and calls the PBS resource monitor library function fullresp().

attrlist()

```
char* attrlist(struct attrl *ap)
```

This function takes a list of attrl's and creates a Tcl list from them. For each attrl, Tcl_Merge() is called to create a list with two elements. The first is the name and resource if it exists, separated by a colon. The second is the value. Tcl_Merge() is called again to combine all the name/value pairs into one list. The functions PBS_StatServ(), PBS_StatJob(), PBS_SelStat(), and PBS_StatQue() all use this to create their return lists.

PBS_StatServ()

```
int PBS_StatServ(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsstatserv" and calls the PBS interface library function pbs_statserv(). The single batch_status struct which is returned is combined using *Tcl_Merge()* to form a list element with the batch_status struct's name, attribs and text forming the three sublists. The routine *attrlist()* is called to form the second sublist out of the attribs. The temporary storage used to create the list is free'ed.

PBS_StatJob()

```
int PBS_StatJob(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsstatjob" and calls the PBS interface library function pbs_statjob(). The list of batch_status struct's which are returned are looped through with an array of three char *'s being setup to pass to *Tcl_Merge()* to form a list element with the batch_status struct's name, attribs and text forming the three sublists. The routine *attrlist()* is called to form the second sublist out of the attribs. All the above list elements are combined in a final call to *Tcl_Merge()* and the temporary storage used to create the lists is free'ed.

PBS_SelStat()

```
int PBS_SelStat(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsselstat" and calls the PBS interface library function pbs_selstat() with a list of attropl struct's giving the attributes to select. To get only the "runnable" jobs from the server, the attropl's are setup to only return jobs with "queue_type=E" and "job_state=Q". The list of batch_status struct's which are returned are looped through with an array of three char *'s being setup to pass to *Tcl_Merge()* to form a list element with the batch_status struct's name, attribs and text forming the three sublists. The routine *attrlist()* is called to form the second sublist out of the attribs. All the above list elements are combined in a final call to *Tcl_Merge()* and the temporary storage used to create the lists is free'ed.

PBS_StatQue()

```
int PBS_StatQue(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsstatque" and calls the PBS interface library function pbs_statque(). The list of batch_status struct's which are returned are looped through with an array of three char *'s being setup to pass to *Tcl_Merge()* to form a list element with the batch_status struct's name, attribs and text forming the three sublists. The routine *attrlist()* is called to form the second sublist out of the attribs. All the above list elements are combined in a final call to *Tcl_Merge()* and the temporary storage used to create the lists is free'ed.

PBS_RunJob()

```
int PBS_RunJob(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsrunjob" and calls the PBS interface library function pbs_runjob().

PBS_AsyRunJob()

```
int PBS_AsyRunJob(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsasyrunjob" and calls the PBS interface library function pbs_asyrunjob().

PBS_MoveJob()

```
int PBS_MoveJob(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsmovejob". A call to the fuction get_server() is made, followed by a call to the PBS interface library function pbs_movejob().

PBS_DelJob()

```
int PBS_DelJob(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsdeljob" and calls the PBS interface library function pbs_deljob().

PBS_HoldJob()

```
int PBS_HoldJob(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsholdjob" and calls the PBS interface library function pbs_holdjob().

PBS_QueueOp()

```
int PBS_QueueOp(ClientData, Tcl_Interp *, int, char *[], struct attropl *)
```

This function is called by PBS_EnableQueue(), PBS_DisableQueue(), PBS_StartQueue() and PBS_StopQueue(). It is not bound to any Tcl function directly. It calls the PBS interface library function pbs_manager() with the second and third parameters of *MGR_CMD_SET* and *MGR_OBJ_QUEUE* respectively, and a *struct attropl* * supplied by the calling routine depending on what action is to be done.

PBS_EnableQueue()

```
int PBS_EnableQueue(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsqenable" and calls PBS_QueueOp().

PBS_DisableQueue()

```
int PBS_DisableQueue(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsqdisable" and calls PBS_QueueOp().

PBS_StartQueue()

```
int PBS_StartQueue(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsqstart" and calls PBS_QueueOp().

PBS_StopQueue()

```
int PBS_StopQueue(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "pbsqstop" and calls PBS_QueueOp().

PBS_AlterJob()

```
int PBS_AlterJob(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "pbsalterjob" and calls the PBS interface library function pbs_alterjob(). The Tcl function *Tcl_SplitList()* is called to separate each of the attributes to be altered. Then a loop is entered to create a attrl structure for each attribute.

DateTime()

```
int DateTime(ClientData, Tcl_Interp *, int, char *[])
```

This function is bound to the Tcl function "datetime". A switch statement is entered for the number of arguments. The time format requested is determined and the result calculated by using the POSIX *time()*, *mktime()* and *localtime()* functions.

StrFtime()

```
int StrFtime(ClientData, Tcl_Interp *, int, Tcl_Obj *CONST [])
```

This function is bound to the Tcl function "strftime" and calls the POSIX function *strftime()*. It requires two arguments. The first is a format string. The format conventions are the same as those for the POSIX function strftime(). The second argument is POSIX calendar time in seconds.

add_cmds()

```
void add_cmds(Tcl_Interp *interp)
```

Call Tcl_CreateCommand() for each of the new commands. Also call site_cmds() so any site-specific processing will be done.

6.2.3. File: pbs_sched.c

This file contains main() and the loop which reads and processes commands from the server.

start_tcl()

```
void start_tcl(void)
```

The function Tcl_CreateInterp() is called to create a Tcl interpreter. Then, add_cmds() is called to create the additional commands. The initialization script is run if it exists and the body script is read into memory.

restart()

```
void restart(int sig)
```

This is the signal handler for SIGHUP. The Tcl interpreter is deleted by calling Tcl_DeleteInterp(). Then a new one is started by calling start_tcl().

server_command()

```
int server_command(int socket_number)
```

This function waits for a server wakeup and reads the command. A call to **accept()** is made followed by a **read()** to get the four byte command. The command is returned.

6.2.4. File: site_tclWrap.c

This file is provided as a holder for any site-specific code which needs to be included. It contains one routine which is called from add_cmds().

site_cmds()

```
void site_cmds(Tcl_Interp *interp)
```

As delivered, this function just returns. Use it to add commands to Tcl that solve your problems in scheduling better, faster and cheaper!

6.3. The C Scheduler

The third provided scheduler is not a complete program. To use this will require the largest initial effort but will yield the most flexablity and quickest runtime of all the other schedulers. It is recommended that a site use either the Basl or Tcl scheduler to to try out policies and move to use the C scheduler only after having firmly settled on something that will not be changed quickly.

There are two functions that must be provided by the scheduler writter. The first is

```
int schedinit(int argc, char *argv[])
```

The parameters passed are the same as those passes to main from the command line. If this function returns a non-zero value, this is considered a failure and the scheduler exits. The second function is

```
int schedule(int cmd)
```

The parameter is the command from the server. If this function returns a non-zero value, this is considered a failure and the scheduler exits. The global variable

```
int connector
```

must be defined and is setup with the PBS connection handle when *schedule* is called.

If a resource mom connection is to be used in the scheduler, the global variable pbs_rm_port should be used as the default port.

6.3.1. File: pbs_sched.c

The C Scheduler has a provided main routine which processes the command line arguments, and sets up signal handling before going into a loop reading scheduler commands.

toolong()

```
void toolong(int sig)
```

This is the routine setup as the alarm signal handler. This carries over from the Tcl scheduler as a error recovery method if the schedule run takes too long. An error message is logged and the network is shutdown. The process forks and the parent simply re-exec's itself to do a clean startup. The child calls abort() if no core file exists already and exits. File:

restart()
```
void restart(int sig)
```

This is the routine setup as the HUP signal handler. It logs a message and calls schedule() with the argument *SCH_CONFIGURE*.

server_command()

```
int server_command(int socket_number)
```

Call **accept()** to get a new socket from the socket specified by *socket_number*. Then convert the new socket to a connection via *socket_to_conn()*.

6.3.2. FIFO Sample C scheduler

Scheduling policies differ greatly from site to site. They differ to such a degree that it is impossible to guess all the different parts that everyone will want in their scheduler. This sample scheduler was meant as a jumping off point into a useful scheduling policy. It would be useful to know where to change the code. In making this scheduler several assumptions were made which will probably be wrong.

To make the scheduler gather more data from the pbs_server:

1. add the variable to the correct data structure in data_types.h i.e. job_info / queue_info / server_info / node_info

2. edit the appropriate file and change the *query_*_info()* function. These functions will loop though a batch_status structure which is returned from the server. There is a large if/else statement block comparing the current element in the list to information that is wanted. Add a new statement to the end block. All the symbolic constants for the attributes are in src/include/pbs_ifl.h.

3. Add the initialization of the new variable in the *new_*_info()* function, and free it in the *free_*_info()* function. Also add a print statement to *print_*_info()* if you plan to use that for debugging.

To have the scheduler check more/different resources:

There is a variable in globals.c which tells the scheduler which variables to check aginst jobs in the scheduling cycles. Change the array res_to_check.

Format: **{ resource_name, comment_msg, debug_msg }**

- resource_name is the name of a resource how PBS views it. Ex: ncpus, cput, mem

- comment_msg is what the comment of the job will bet set to if there is a insufficient amount of this resource

- debug_msg is what will be logged if there is an insufficient amount of this resource

To add a new sorting method:

1. add a new element to the sort_type enum in constants.h

2. write the compare function used by qsort the prototype: *int func(const void *v1, const void *v2)*

   The compare function should return

   -1: if v1 < v2

   0: if v1 == v2

   1: if v1 > v2

   The current compare functions are in the file *sort.c*.
   NOTE: multi_sort uses the global array, so it will automatically work with the new sort.

3. add to the sorting_info array in globals.c

   Format: **{ sort_type, config_name, cmp_func_ptr }**

   - sort_type: the element in the enum sort_type

   - config_name: string: The name of the sort which is used in the scheduling policy config file

   - cmp_func_ptr: pointer to compare function
     `int (*cmp_func_ptr) (const void*, const void*)`

To add a global sort i.e. one that happens with every sort:

There is one entry point into the sorting compare function. Currently it is sorting on sch_priority and if equal to call the requested sorting function. Modify this function to change the sorting globally.

To change how the scheduler picks the next job candidate to run:

This is decided in the *next_job()* function. Currently there are 3 choices Round Robin, By Queue, or neither. If another choice is added, a bit should be added to the config strucuture and scheduling policy config file should be updated.

To change how the scheduler decides if a job can fit into the system.

All the checks for a job are in check.c *is_ok_to_run_in_queue()* is run once per scheduling cycle for each queue. Any any queue check which needs to be checked but wont change within a scheduling cycle should be added to this fuction. Any check which needs to be checked once for each job should be added to *is_ok_to_run_job()*.

To add to the scheduling config file:

1. add a variable to the config structure and possibily status structure in data_types.h. If what is being added is going to change between prime and non-prime, the status structure needs to be changed. If it will change with prime and non-prime time, make sure to change the init functions (see below).
2. add a symbolic constant in config.h which will hold the name which will appear in the config file. The prefix for the symbolic constant is PARSE.
3. change the parse_config function in parse.c There is an if/else block which checks each config word from the file. This block needs to be expanded. The variable config_name contains the name on the left side of the colen(:) in the config file. If the value is boolean or numeric, the num variable will hold its value. In any case the variable config_value will hold the string value.

   Find a good place in the if/else structure to add your new item. It should be if(!strcmp(config_name, PARSE_symbolic_constant))

   If an error is detected set the variable error to 1, and it will be printed if there is an error.

To have something happen at the start of primetime or nonprimetime

There are two functions in the file prime.c. *Init_non_prime_time()* and *init_prime_time()*.

These functions are called in the beginning of primetime and nonprimetime. Add all the necessary code in those functions. The status structure is updated in theses functions.

To change information gathered by MOM Information is picked up from mom for nodes.

1. edit globals.c and add an element to the array res_to_get. This will cause the scheduler to query mom for the resource.
2. edit data_types.h and add new members to the node_info structure.
3. edit the function *talk_with_mom()* in node_info.c. Near the end of the function there will be an if/else block. This is where the answers from mom are converted into the data for the node.

NOTE: If you are running a single timeshared system, set up a nodes file with your one system and mark it timeshared(:ts).

To change how load balencing is done:

The load balencing policy is done in 2 functions. The first checks to see if there is a timesharing node available to run on, *is_node_available()*. The second finds the best timesharing node to run on, *find_best_node()*.

To change how starving jobs are helped:

Job starvation is done by setting a internal scheduler priority variable sch_priority. This is done in the function *update_starvation()*. It is called in *init_scheduling_cycle()* to make sure it will be updated every cycle. Currently, all which is done is to set the priority to the ammount of times it has waited the max_starve time.

The seconds half of the job starvation code is how to allow jobs to run while there is a starving job. There is a function in *check.c* which only allows the most starving job to run. The function knows which job is most starving, since it is stored in the scheduling cycle status variable, stat.

6.3.2.1. File: globals.c

This file defines the necessary global variables for the scheduler. Most of the variables are constant.

res_to_check
: This is the list of resources the scheduler will check inorder to see if a job can run. The Format: **name, comment_msg, debug_msg**

namethe name of the resource as PBS knows it

comment_msg
: If the job can not run, the comment attribute of the job will be changed to this message.

debug_msg
: If the job can not run, this debug message will be logged

sorting_info
: This variable holds all the information about the different sorts which can be done on the jobs.
The Format:
sort_type, sort_name, cmp_func

sort_type
: element from the enum sort_type

sort_name
: The name of the sort that will appear in the scheduling policy config file i.e. shortest_job_first

cmp_func
: the function pointer to qsort compare function

num_res
: This is the number of elements in the res_to_check array

num_sorts
: This is the number of elements in the sorting_info array

conf This is the global config structure. This holds all the run time config info read in from the scheduling policy config file. This information does not change during the runtime of the scheduler.

stat This is the global scheduling cycle status structure. This holds all the configuration information which changes during the runtime of the scheduler.

6.3.2.2. File: check.c

The functions in this file deal with checking if a job can run on the system at the current time.

The main check functions

is_ok_to_run_in_queue

```
int is_ok_to_run_queue(queue_info *qinfo)
```

Args:

qinfo the structure holding the queue information

Returns:

SUCCESS
if is possible to run jobs in the queue

QUEUE_NOT_STARTED
if the queue is not started

QUEUE_NOT_EXEC
if the queue is not an execution queue

IN_DED_TIME
if it is dedicated time and the queue is not a dedicated time queue

This function is called on each queue to see if jobs in the queue can be run. It checks first to see if the queue's started attribute is set to 'true' and then it checks to see if the queue is a execution queue. It also will check to see if it is currently dedicated time and if the queue is a dedicated time queue or if we are not in dedcated time. If all the conditions are true then jobs can run in the queue.

NOTE: this function gets called once per queue each scheduling cycle, other queue related variables that can change doing the scheduling cycle are checked in *is_ok_to_run_job()*.

is_ok_to_run_job

```
int is_ok_to_run_job(server_info *sinfo, queue_info *qinfo, job_info *jinfo)
```

Args:

sinfo the server the job resides in

qinfo the queue the job resides in

jinfo the job to check

Returns:

SUCCESS
if the job is OK to run

QUEUE_GROUP_LIMIT_REACHED
if the queue's max_group_run limit has been reached

QUEUE_USER_LIMIT_REACHED
if the queue's max_user_run limit has been reached

QUEUE_JOB_LIMIT_REACHED
if queue's max_running limit has been reached

SERVER_GROUP_LIMIT_REACHED
if the server's max_group_run limit has been reached

SERVER_USER_LIMIT_REACHED
if the server's max_user_run limit has been reached

SERVER_JOB_LIMIT_REACHED
if the server's max_running limit has been reached

NOT_QUEUED
if the job is not in the queued state

CROSS_INTO_DED_TIME
if the job would cross a dedicated time boundry

NOT_ENOUGH_NODES_AVAIL
if the there are not enough of the right type of nodes available

return_code
from what *check_avail_resources()*

This function will check the queues max_running, max_user_run, and max_group_run attributes, and the servers max_running, max_group_run, and max_user_run attributes. It will also check if the job is queued. It will check of the job would cross a dedicated time boundry. There is a check to see if there are enough nodes of the right type of nodes available to run the job. Finally, it will it will call *check_avail_resources()* to see if the job is able to run within the server's resources.

Helper Functions

check_avail_resources

```
int check_avail_resources(server_info *sinfo, queue_info *qinfo,
                          job_info *jinfo)
```

Args:

sinfo the server the job resides in

qinfo the queue the job resides in

jinfo the job

Returns:

SUCCESS
if job is within the resources left on the system

index
of the res_to_check array for the resource which was lacking.

This function will check to see if the there are enough resources available to run the job. The resources are specified in the global array res_to_check. If the resource is not found or set to infinity, the check is skipped.

check_server_max_user_run

```
int check_server_max_user_run(server_info *sinfo, char *account)
```

Args:

sinfo the server

account
account name of the owner of the job

Returns:

0 if user is within server's max_user_run limit

SERVER_USER_LIMIT_REACHED
if user is above server's max_user_run limit

This function counts the number of running jobs on the server which are owned by account. If that less then the max_user_run attribute on the server then {TRUE} is returned. If max_user_run is not set, then this function does not count.

check_queue_max_user_run

```
int check_queue_max_user_run(queue_info *qinfo, char *account)
```

Args:

qinfo the queue

account
account name of the owner of the job

Returns:

0
if the user is within queue user run limits

QUEUE_USER_LIMIT_REACHED
if the user is above queue user run limits

This function counts the number of running jobs the user has running in the queue and checks it against the max_user_run of the queue. Nothing is done if max_user_run is not set

check_queue_max_group_run

```
int check_queue_max_group_run(queue_info *qinfo, char *group)
```

Args:

qinfo information about the queue

groupthe group name

Returns:

0
if the group is within their queue group run limits

QUEUE_GROUP_LIMIT_REACHED
if the group is above their queue group run limits

This function counts the number of running jobs the group has in the queue and checks that against the max_group_run of the queue. Nothing is done if max_group_run is not set.

check_server_max_group_run

```
int check_server_max_group_run(server_info *sinfo, char *group)
```

Args:

sinfo the server

groupgroup name

Returns:

0

if the group is within their server group run limits

SERVER_GROUP_LIMIT_REACHED

if the group is above their server group run limits

This function counts the number of running jobs the group has on the server and checks that against the max_group_run of the server. Nothing will be done if max_group_run is not set

dynamic_avail

```
long int dynamic_avail(resource *res)
```

Args:

res the resource

Returns:

the remaining availability of the resource

This function will return the remaining unallocated amount of a resource. The server will return the maximum amount of a resource(resources_max). It will also return the amount which is available for the scheduler to use (resources_available). If the resources_available attribute of the server is set, the amount available at the current time is resources_available - resources_assigned. If it is not set, the maximum is used: resources_max - resources_assigned.

count_by_user

```
int count_by_user(job_info **jobs, char *user)
```

Args:

jobs an array of jobs

user the user name

Returns:

The number of jobs the user owns in the array

This function counts the number of jobs the user owns in the array of jobs, jobs.

count_by_group

```
int count_by_group(job_info **jobs, char *group)
```

Args:

jobs an array of jobs

group the group name

Returns:

The number of jobs the group owns in the array

This function counts the number of jobs the group owns in the array of jobs.

check_ded_time_boundry()

```
int check_ded_time_boundry( job_info *jinfo )
```

Args:

jinfo The job to check

Returns

0: if the job will not cross into dedicated time

CROSS_DED_TIME_BOUNDRY:
if the job will cross either dedicated time boundry (start or finish)

This function will check if the job will cross either dedtime boundry (start or finish). If it is currently dedtime the function will check if the job will complete before dedtime is over. If it is not currently dedtime, the function will check if the job will finish before dedtime starts.

check_ded_time_queue()

```
int check_ded_time_queue( queue_info *qinfo )
```

Args

qinfo The queue to check

Returns

0 if it is dedtime and the queue is a didtime queue or if it is not dedtime and the queue is not a dedtime queue

DED_TIME
otherwis

This function will check if it is an approprate time to run jobs in the queue If it is dedtime only dedtime queues can run jobs. If it is not dedtime make sure no jobs are run out of dedtime queues.

check_nodes()

```
int check_nodes( int pbs_sd, job_info *jinfo )
```

Args

pbs_sd
: the connection descriptor to the pbs server

jinfo
: the job to check

ninfo_arr
: Array of nodes

Returns

0
: if the job can run

NOT_ENOUGH_NODES_AVAIL
: if there are not suficient nodes of the correct type to run the job

SCHD_ERROR
: on error

First, this function will check what type of nodes the job needs. If the job needs cluster nodes, the function will make a *pbs_rescquery()* call to see if there are sufficent nodes to be able to run the job. If the job needs timesharing nodes a call to *is_node_available* will be made and its value returned.

check_node_availability()

```
int check_node_availability( job_info *jinfo, node_info **ninfo_arr )
```

Args

jinfo
: The job to check if there is a node available for

ninfo_arr
: The array if nodes to check if the job will fit

Returns

0
: if the node is available

NO_AVAILABLE_NODE
: if the node is not available

This function is used to find out if a node exists which the job can be run on. Currently the function will check if the arch of the job and the machine match, the memory is not more

then the max physmem of the machine, and that the load avereage wont raise above the max load ave for a node.

check_starvation

```
int check_starvation( job_info *jinfo )
```

Args:

jinfo the current job to check

Returns

0

If jinfo is the starving job or there are no starving jobs or starving jobs are not being helped

JOB_STARVING

if jinfo is not the starving job and there are starving jobs

This function will allow only the starving job to run if there are starving jobs and the scheduler is helping starving jobs.

6.3.2.3. File: fairshare.c

This file contains all the functions dealing with the fairshare algorithm. The functions to create the group_info structures, and to create the resource group tree. Also the functions to collect usage, and to select the next best job to be considered to be run.

add_child

```
void add_child( group_info *ginfo, group_info *parent )
```

Args:

node The group_info to add to the tree

parent

The parent of the group_info to be added to the tree

The function adds ginfo onto the resource group tree. The parent of where the node should be is passed in. It connects the group_info, and sets the resgroup.

add_unknown

```
void add_unknown( group_info *node )
```

Args:

node the node to add

The function will add the node onto the "unknown" group. It also recalculates the fair share percentages for the unknown group since they will have changed.

find_group_info

```
group_info *find_group_info( char *name, group_info *root )
```

Args:

nameThe name of the group_info to find

root the root of the tree

Returns

the found group_info or {NULL}

This is a recursive function which runs through the fair share tree trying to find the group_info whose name is passed in.

find_alloc_ginfo

```
group_info *find_alloc_ginfo( char *name )
```

Args:

namethe name of the group_info to find or allocate

Returns

The group if it is found, or the new group if it is created

This function will either find the specified group_info, or allocate an new group_info with the specified name and add it to the "unknown" group.

new_group_info

```
group_info *new_group_info()
```

Returns

a point to the new group_info struct

Allocate and initialize a group_info structure

parse_group

Args:

```
int parse_group( char *fname )
```

fnamethe name of the file to parse

Returns

success/failure

This function opens and parses the the resource_group file. It will take each line of the file and either add a new group_info to the tree or print an error. The function does rudimentary error checking.

free_group_tree

```
void free_group_tree( group_info *root )
```

Args:

root The root of the resource group tree

This is a recursive function which will free all the group_info structures in the resource group tree.

preload_tree

```
int preload_tree()
```

Returns:

success/failure

This function loads the "root" group and the "unknown" group into the resource group tree. This function should be called after parse_config(because it uses the unknown_shares value from the config file.

count_shares

```
int count_shares( group_info *grp )
```

Args:

grp

Returns

the number of shares in the group

This function will count the number of shares in a resource group. It will go down a sibling chain and count the shares.

calc_fair_share_perc

```
int calc_fair_share_perc( group_info *root, int shares )
```

Args:

root The root of the current subtree

shares

The total number of shares in the group or UNSPECIFIED if the value is unknown

Returns

success/failure

This function recurses down the resource group tree calculating the percentange of the machine a user gets. A user is defined as a leaf of the tree. The function that is used: parent_percentage * (shares/total_group_shares). The percentage of the "root" group is 1.0. When a child link is taken to a new group is taken, the shares of that group is counted.

test_perc

```
float test_perc( group_info *root )
```

Args:

root The root of the current subtree

Returns:

total percentage (hopefully 1.0)

This is a debugging function to test the percentages calculated from the resource group tree. It recursivly traverses the tree and adds the percentages for the leaves (users). The target number is 1.0.

update_usage_on_run

```
void update_usage_on_run( job_info *jinfo )
```

Args:

jinfo The job which was just run

This function will add the entire jobs usage to the job temporarly. It will last for this cycle only.

calculate_usage_value

```
long calculate_usage_value( resource_req *resreq )
```

Args:

resreqThe resources to calculate the usage value from

Returns

The calculated value

This function takes all the usage into account which is stored in the resource_req struct passed in. This function is to be modified how the scheduling policy wants to collect the usage. Currently it only accumulates cput. An example of another method would be to accumute cput * 100000 + mem. This would make cput much more important them memory usage, but memory usage would come into effect also.

decay_fairshare_tree

```
void decay_fairshare_tree( group_info *root )
```

Args:

root the root of the current subtree

This function decays the resource group tree. Since the algorighm calls for a half life, the function decays the information by 50%. If the usage decays to zero, it is reset to the default value of one. The function is recursive. The tree gets decayed in a post order traversal.

extract_fairshare

```
job_info *extract_fairshare( job_info **jobs )
```

Args:

jobs The array of jobs to extract from

Returns:

the job with the max fairshare value

This is a extract max function for the fair share algorithm. The function extracts the job with the max value of the function percentage / usage. The value which is returned will be first job of the user with the max value. Usage defaults to one and will not decay below that, so no division by zero error can happen. The function runs in O(n) time.

print_fairshare

```
void print_fairshare( group_info *root )
```

Args:

root The root of the current subtree

This function will print out the fair share tree in a preorder traversal.

write_usage

```
int write_usage()
```

Returns:

success/failure

This function opens the usage file and calls *rec_write_usage()* to write out the resource group tree.

rec_write_usage

```
void rec_write_usage( group_info *root, FILE *fp )
```

Args:

root The root of the current subtree

fp file pointer of the usage file

This function copies the usage and name of a group_info struct into a group_node_usage struct. The smaller struct is written out to the file. Nodes with usage equal to one are not written out, since the default value for usage is one. The function writes out in a preorder traversal.

read_usage

```
int read_usage( )
```

Returns
success/failure

This function reads in the usage information into the usage file. It will find the correct group_info and assign the usage information into the group_info. *find_alloc_ginfo()* is called to find the group_info, so if the user is not in the tree already, a group_info will be added to the "unknown" group.

6.3.2.4. File: job_info.c

This file contains all the functions which have to deal with jobs. The functions which create the job_info structures, and the resource_req structures

query_jobs

```
job_info **query_jobs(int pbs_sd, queue_info *qinfo)
```

Args:

pbs_sd
the connection descriptor to the pbs server

qinfo information about the queue

Returns:
a {NULL} terminated array of jobs that reside in the queue

This function will query all the jobs from the server that reside in the queue. It will then count the jobs so the correct ammount of space can be allocated. After allocation is done, it will call query_job_info on each of the batch_status structs that the server returned. It puts the sentinel value {NULL} at the end of the array. The group_info in the resource group tree is found and assigned into the job structure. Finally, frees up the batch_status list.

query_job_info

```
job_info *query_job_info(struct batch_status *job, queue_info *queue)
```

Args:

job the job information returned from the server

queue
information about the queue the job resides in

Returns:
A pointer to a job_info struct of the processed data about the jobs

This function collects the data out of the linked list of values in the batch_status structure and puts that data into a job_info structure. It checks for {ATTR_p} (priority), {ATTR_qtime} (time job was queued), {ATTR_state} (state of job), {ATTR_comment} (job comment), {ATTR_euser} (user-name of owner), {ATTR_egroup} (group name of owner), {ATTR_exechost} (host the job is executing on), {ATTR_l} (resource requested), and {ATTR_used} (resource used)

new_job_info

```
job_info *new_job_info()
```

Returns:
Pointer to newly allocated and initialized job_info struct

This function allocates and initializes a job_info struct

new_resource_req

```
resource_req *new_resource_req()
```

Returns:
Pointer to newly allocated and initialized resource_req struct

This function allocates and initializes a resource_req struct

find_alloc_resource_req

```
resource_req *find_alloc_resource_req(char *name, resource_req *reqlist)
```

Args:

name
name of resource to look for

reqlist
resource_req list to look in

Returns:
Pointer to found or newly allocated *resource_req*

This function will attempt to find the *resource_req* struct with the name passed in. If it is found, it is returned. If it can not be found, a new *resource_req* is allocated and the name field is assigned.

free_job_info

```
void free_job_info(job_info *jinfo)
```

Args:

jinfo the job_info structure to free

This function will free all the memory used by a job_info struct.

free_jobs

```
void free_jobs(job_info **jarr)
```

Args:

jarr array of jobs to free

This function calls free_job_info all each element in the array. Finally, it will free the array itself

find_resource_req

```
resource_req *find_resource_req(resource_req *reqlist, const char *name)
```

Args:

reqlist
the *resource_req* linked list to search through

namethe name to look for

Returns:

Found resource_req or {NULL} if the resource can not be found

This function will search through the resource_req list looking for a *resource_req* with the specified name.

free_resource_req_list

```
void free_resource_req_list( resource_req *list )
```

Args:

list the resource_req list to free

Frees up memory used by a resource_req linked list.

print_job_info

```
void print_job_info( ob_info *jinfo, char brief)
```

Args:

jinfo job_info to print

brief boolean

This function will print out the job info to stdout. It is meant for debugging purposes. If the brief flag is true, only the name of the job is printed.

set_state

```
void set_state(char *state, job_info *jinfo)
```

Args:

state the state of the job

jinfo job information which needs one of the state bits set

Returns:

jinfo is passed in by reference

This function will set one of the following state bits: is_queued, is_running, is_held, is_transit, is_exiting, is_waiting.

update_job_on_run

```
void update_job_on_run(int pbs_sd, job_info *jinfo)
```

Args:

pbs_sd
connection descriptor to the pbs server

jinfo the job which was just run

This function updates the state bit from is_queued to is_running.

update_job_comment

```
int update_job_comment(int pbs_sd, job_info *jinfo, char *comment)
```

Args:

pbs_sd
connection descriptor to the pbs server

jinfo job to update

comment
the comment string

Returns:

Success or Failure. Use pbs_errno for more information

The function first checks if the comment of the job is the same of what it is being set to. If it is, the comment will not be updated. If the comment is not the same, the old space for the comment in jinfoi is freed in order to update it. Lastly pbs_alterjob is called to update the comment on the server

translate_job_fail_code()

```
void translate_job_fail_code( int fail_code, char *comment_msg, char *log_msg )
```

Args:

fail_code
: the return code from the check functions

comment
: string passed by reference

This function will translate the failure code from *is_ok_to_run_job()* to both a comment message and a log error message. They are copied into buffers supplied by the caster. Any code which is less then {RET_BASE} is considered to be an index into the res_to_check array. The default action is the clear the comment message. The symbolic constants are defined in config.h

update_jobs_cant_run

```
void update_jobs_cant_run(int pbs_sd, job_info **jinfo_arr, job_info *start,
                     char *comment, int start_where)
```

Args:

pbs_sd
: connection descriptor to the pbs server

jinfo_arr
: array of jobs to update comments for

start the job to start updating comments or NULL: start at the front of the array

comment
: the comment to update the jobs

start_where
: where to start relative to the job start

Returns:

nothing - updates comments of jobs in jinfo_arr

This function will update all the comments in a job_info array. It will start before, at, or after the job in start depending on the start_where parameter. The function will also set the job_can_not_run bit.

job_filter

```
job_info **job_filter(job_info** jobs, int size, int (*filter_func) (job_info*, void*), vo
```

Args:

jobs an array of jobs

size size of the array

filter_func
pointer to a function that will do filtering

arg extra arg to pass to filter_func

Returns:
Pointer to the head of the newly allocated filtered array

This function will filter through the jobs in an array. It will call filter_func on each job in the array. If the function returns non-zero, the job is kept, if it returns 0, the job is not kept. A new array is allocated and is up to the user to free.

NOTE: Only an array of pointers has been allocated. The jobs are not copied.

6.3.2.5. File: misc.c

This file contains miscellaneous functions which are used everywhere in the scheduler

string_dup

```
char *string_dup(char *str)
```

Args:

str the string to duplicate

Returns:
Pointer to a newly allocated copy of str

This function will allocate a new string and copy the parameter into the newly allocated space.

NOTE: This function was used instead of strdup() because it is not in the POSIX.1 standard

log

```
void log( int event, int class, char *name, char *text)
```

Args:
IP event
the event type

class
: The event class of the log

name
: the name of the object

text
: the text of the log message

This function will first check if the event type is being filtered. If it isn't log the record using *log_record()*.

res_to_num

```
long int res_to_num(char *res_str)
```

Args:

res_str
: the string returned by the server as a resource

Returns:

: The numeric resource in kilobytes or kilowords for memory, or in seconds for time

This function will convert a string in the form of HH:MM:SS into a number corresponding to the total number of seconds. It will also convert a memory string into the corresponding number of kilobytes. Symbolic constants {MEGATOKILO}, {GIGATOKILO}, and {TERATOKILO} are used in the conversions. The constant {SIZE_OF_WORD} is used in converting words to bytes.

skip_line

```
int skip_line(char *line)
```

Args:

line
: the line from a config file

Returns:

1
: skip the line

0
: parse the line

This function will return 1 if the line is a comment(starting with # or *) or a blank line of only white space.

6.3.2.6. File: parse.c

This file contains functions which read in and parse the scheduling policy configuration file.

parse_config

```
int parse_config(char *fname)
```

Args:

fnamea string containing the filename

Returns:

success/failure and the global conf will be assigned

This function will open the scheduling policy config file and parse it. The global config variable conf will be assigned with the correct values parsed from the file. The format of the file is:

name : value

The scanner is a little lax on scanning. It will skip over white space and the ':'. The parser has rudimentary error detection and recovery. If an error is detected, a message is printed to stderr and the line is skipped.

init_config

```
int init_config()
```

Returns:

success/failure

This function will initialize the global config structure, conf.

reinit_config

```
int reinit_config()
```

Returns:

success/failure

Frees up memory used by the conf config structure and frees the resource group tree. *init_config()* is called to do the initialization.

6.3.2.7. File: queue_info.c

This file contains the functions to create and handle the queue_info structures.

query_queues

```
queue_info **query_queues(int pbs_sd, server_info *sinfo)
```

Args:

pbs_sd
connection descriptor to the pbs server

sinfo information about the server

Returns:

Pointer to an array of queue_info structures

This function does a pbs_statque() to get the information about all the queues on the server. It will then count the queues and allocate an array of pointers to point at queues. Next, it will call *query_queue_info()* on each queue and assign them into that array. The jobs are queried from the server by *query_jobs()*. The function *is_ok_to_run_queue()* is called on each job and is_ok_to_run is set. If it is not OK to run, all the job comments for the jobs in the queue are changed. The job states are counted in the queues. Finally, the list of running jobs is created and general cleanup is done.

query_queue_info

```
queue_info *query_queue_info(struct batch_status *queue, server_info *sinfo)
```

Args:

queuethe batch_status struct returned from the server

sinfo information about the server the queue resides in

Returns:

Pointer to newly allocated and assigned queue_info struct or {NULL} on error

This function takes information out the the linked list in the batch_status struct and puts it into a queue_info struct. The following attributes are converted: {ATTR_start} (started), {ATTR_maxrun} (max_running), {ATTR_maxuserrun} (max_user_run), {ATTR_maxgrprun} (max_group_run), {ATTR_p} (priority), and {ATTR_qtype} (queue_type)

new_queue_info

```
queue_info *new_queue_info()
```

Returns:

Pointer to a newly allocated and initialized queue_info

This function allocates a new queue_info struct and initializes it.

print_queue_info

```
void print_queue_info(queue_info *qinfo, char brief, char deep)
```

Args:

qinfo the queue to print info about

brief only print queue name

deep print info about jobs in queue also

This function will print out a queue_info structure. It is mainly used for debugging. If brief is true, then only the name of the queue is printed. If deep is true, all the jobs in the queue will be printed. brief is passed to *print_job_info()*

update_queue_on_run

```
void update_queue_on_run(queue_info *qinfo, job_info *jinfo)
```

Args:

jinfo the job which was run

qinfo the queue jinfo is in

This function updates the state counts in the queue

free_queues

```
void free_queues(queue_info **qarr, char free_jobs_too)
```

Args:

qarr an array of queues

free_jobs_too
free the jobs in the queues also

This function will call free_queue_info on each queue in the array and then finally free the array. If free_jobs_too is true, free_jobs is called on the job arrays within the queues.

free_queue_info

```
void free_queue_info(queue_info *qinfo)
```

Args:

qinfo pointer to a queue_info structure to free

This function will free all the memory used by a queue_info struct

6.3.2.8. File: server_info.c

This file contains all the functions to create, handle, and free the server_info and the resource structures.

query_server

```
server_info *query_server(int pbs_sd)
```

Args:

pbs_sd
connection descriptor to the pbs server

Returns:

A newly allocated server_info struct

This function calls pbs_statserver() to get batch_status struct about the server. It calls *query_server_info()* to collect all the server information. The nodes are querried by a call to *query_nodes()*. It calls *query_queues()* to get all the info about the queues (which gets the info about the jobs). It then counts the number of queues and collects all the state counts. It will then allocate a *job_info* array and copy into it pointers to all the jobs. Finally it will set running_jobs by filtering out all but running jobs. Timesharing nodes are also set in a simular way.

query_server_info

```
server_info *query_server_info(struct batch_status *server)
```

Args:

server
batch_status struct returned from the server

Returns:

pointer to newly allocated and assigned server_info struct

This function will allocate a new server_info struct. It will then fill it with the information from the linked list within server. It checks the following: {ATTR_dfltque} (default_queue), {ATTR_maxrun} (max_running), {ATTR_maxuserrun} (max_user_run), {ATTR_maxgrprun} (max_group_run), {ATTR_rescavail} (resources_available), {ATTR_rescmax} (resources_max), {ATTR_rescassn} (resources_assigned) It will combine the resources_available, resources_max, and resources_assigned into one resource structure

find_alloc_resource

```
resource *find_alloc_resource(resource *resplist, char *name)
```

Args:

resplist
resource list to search

nameresource name to search for in the list

Returns:

pointer to found resource a newly allocated resource

This function will search through the the resplist. If it find the resource, it returns it. If it is not found, then a new resource is allocated and added to the resplist. The name field is also set.

find_resource

```
resource *find_resource(resource *reslist, const char *name)
```

Args:

reslist
resource list to search through

namethe name to search for

Returns:

pointer to found resource or {NULL}

This function searches through the reslist for the resource specified.

free_server_info

```
void free_server_info(server_info *sinfo)
```

Args:

sinfo the server to free

This function frees all the memory associated with a server_info structure

new_server_info

```
server_info *new_server_info( )
```

Returns:

newly allocated and initialized server_info struct

This function allocates and initializes a new server_info struct

new_resource

```
resource *new_resource()
```

Returns:

newly allocated and initialized resource struct

This function allocates and initializes a new resource struct

print_server_info

```
void print_server_info(server_info *sinfo, char brief)
```

Args:

sinfo information about the server

brief if true only print the server name

This function will print all the fields in a server_info struct. If brief is true, it will only print the name.

free_server

```
void free_server(server_info *sinfo, int free_queues_too)
```

Args:

sinfo the server to free

free_queues_too
if true, will call *free_queues()* on the queues

This function will call *free_server_info()* to free the server, and if free_queues_too is true, will call *free_queues()* to free the queues and jobs.

update_server_on_run

```
void update_server_on_run(server_info *sinfo, queue_info *qinfo, job_info *jinfo)
```

Args:

sinfo server to update

qinfo queue the job was in

jinfo the job which was run

This function updates the information in a *server_info* structure when one of its jobs has been run. First the function will update the running and queued counts, and then update the resources that were assigned to the job.

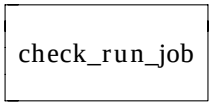
set_jobs

```
void set_jobs(server_info *sinfo)
```

Args:

sinfo
the server

This function will create an array of all the jobs on the server from the arrays contined in the queues. The array will be a list of pointers to the jobs. The jobs themselves are not copied.




```
int check_run_job(job_info *job, void *arg)
```

Args:

job the job to check
arg optional argument

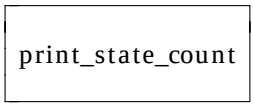
Returns:

1 if the job is running
0 if the job is not running

This function is used by job_filter to keep only running jobs. i.e. return 1 if the job is running.

6.3.2.9. File: state_count.c

This file contains all the functions to handle state_count structures




```
void print_state_count(state_count *sc)
```

Args:

sc state_count to print

This function prints all the fields of a state_count struct. It is mainly used in debugging.


```
void init_state_count(state_count *sc)
```

This function initializes the state count passed in as a parameter

count_states

```
void count_states(job_info **jobs, state_count *sc)
```

Args:

jobs array of jobs

sc state_count passed in as reference

Returns:

sc is passed in by reference

This function will loop through the jobs in the array and count the amount in each state. Then total is set by adding the counts together.

total_states

```
void total_states(state_count *sc1, state_count *sc2)
```

Args:

sc1 state_count struct which gets accumulated into

sc2 state_count which gets added into sc1

Returns:

sc1 is passed by reference

Basically this function does sc1 += sc2; all the fields of sc1 are added with the like field in sc2 and stored in sc1

6.3.2.10. File: fifo.c

This file contains the most important functions in the scheduler. The two main functions which are called for by pbs_sched.c, *schedule()* and *schedinit(). Schedule()* will handle the scheduling command, and call *scheduling_cycle()* to handle a normal cycle. It calls the rest of the functions in order to run the jobs.

schedinit

```
int schedinit(int argc, char *argv[])
```

Args:

argc number of arguments passed into the program on the command line.

argv the arguments passed into the program on the command line.

Returns:

success/failure

This function calls several functions to parse the config files to set up the scheduler for operation.

init_scheduling_cycle

```
int init_scheduling_cycle(server_info *sinfo)
```

Args:

sinfothe server/queue/job info structure

Returns:

success/failure

This function takes care of things that need to before happen every scheduling cycle. If fair-share is turned on, it will collect the usage information by finding the difference between the current resources_used minus the last cycles resources_used. A check to see if it is time to decay is done. It is possible it should have happened in the past, so the last_decay variable will be set to when it should have happened. A check to see if it is time to sync the usage happens also. If the queues need to be sorted, they are sorted by priority, and the jobs are sorted if a sort was selected. It also calls *next_job(1)* to initialize the scheduling policy.

schedule

```
int schedule(int cmd, int sd)
```

Args:

cmd The reason why schedule was called

sd connection descriptor to the pbs server

Returns:

success/failure

This is the function which gets called to start a scheduling cycle. A switch will be done on cmd to see what needs to be done. {SCH_ERROR}, {SCH_SCHEDULE_NULL}, {SCH_RULESET}, {SCH_SCHEDULE_RECYC} are ignored. {SCH_SCHEDULE_RECYC} is ignored because it is meant for a type of scheduler which will only run one job at a time. The server will send a recycle command to the scheduler if only one job is run. There is no reason to run another scheduling cycle if this occurs.

{SCH_SCHEDULE_NEW}, {SCH_SCHEDULE_TERM}, {SCH_SCHEDULE_FIRST}, {SCH_SCHEDULE_CMD}, {SCH_SCHEDULE_TIME} will cause a scheduling cycle to be run. The function *scheduling_cycle()* is called.

{SCH_CONFIGURE} will cause the scheduler to reinitialize its self. The usage information will be written to disk. The config structure will be reinitialized, and the config files will be reread. Lastly the usage info is read back from disk.

{SCH_QUIT} returns 1 from schedule() which will cause the scheduler to exit nicely.

by default return zero which will cause the scheduler to wait for its next cycle to be started.

update_cycle_status

```
void update_cycle_status()
```

This function will update all the status bits in the beginning of every scheduling cycle. It checks for dedicated time, chage in primetime, and sets the status them.

scheduling_cycle

```
int scheduling_cycle( int sd )
```

Args:

sd the connection descriptor to the PBS server

This is the main function which controls the scheduling cycle. It will first call *query_server()* to set up the server/queue/job info structure. Then it will call *schedule_init()* to initialize the scheduling cycle. Finally it gets into the main loop. This loop is controlled by successive calls to *next_job()*. Once *next_job()* returns the next job to run, the function will call *is_ok_to_run_job()* to see if it is within server and queue limits to run. If it can be run, the job will be passed to *run_update_job()* to run the job and to update the internal information (server/queues/jobs). Finally the running jobs are saved and the server / queue / job structure is freed up.

update_last_running

```
int update_last_running( server_info *sinfo )
```

Args:

sinfo the server info

Returns

success/failure

This function frees up the jobs pointed to by the global variable last_running, and it will create a new array from the current running jobs.

run_update_job

```
int run_update_job(int pbs_sd, server_info *sinfo, queue_info *qinfo, job_info *jinfo)
```

Args:

pbs_sd
connection descriptor to the pbs server

sinfo information about the server the job resides in

qinfo information about the queue the job resides in

jinfo the job which needs to run

Returns:
success/failure - see pbs_errno for more details

This function will first run the job and then call the necessary update functions to update the information kept about the jobs in this scheduling cycle. This is done so the server does not need to be consulted every time a job is run. If load balencing is on, the function will call *find_best_node()* to find the best node to run the job on. If *pbs_runjob()* fails, the job comment will be updated to the PBS error message.

next_job

```
job_info *next_job(server_info *sinfo, int init)
```

Args:
sinfo the server to find the next job to run

init Whether or not to initialize

Returns:
The next job to run

This is the main function which controls the scheduling policy. It finds the next job to be considered for running. There are currently three deciding places in this function, whether the jobs should be run round robin, by queue, or just in server order. Several static variables help out. The variables are last_job, last_queue, and cjobs.

If the jobs are to be run in round robin order, the init section will allocate an array of an array of jobs, cjobs. This will be used to cycle through the queues. If strict fifo is set and a job could not run, that queue in the array, cjobs, will be set to {NULL} to insure no more jobs will run from that queue. If fairshare is turned on, instead of picking the next job in the queue to run, *extract_fairshare()* is called to find the next job to run.

If the jobs will be running by queue, the variable last_job and last_queue are used to index into the sinfo -> queues[] -> jobs[] arrays. If fairshare is turned on, sinfo -> queues[] -> jobs[] is passed into *exract_fairshare()* for the next job to be picked. Lastly if the jobs are to be run in queue order, the sinfo -> jobs[] array is used along with the last_job variable. If fairshare is turned on, "sinfo -> jobs[]" is passed into the e If fairshare is turned on, "sinfo -> jobs[]" is passed into the *extract_fairshare()* function for the next job to be found.

update_starvation()

```
job_info *update_starvation( job_info **jobs );
```

Args:

jobs The jobs to update their sch_priority

Returns

The most starving job

This function will go through all the jobs and set their sch_priority. It will be set to *qtime / max_starve*. Which means every time the job waits a max_starve period of time, thier sch_priority goes up by one.

6.3.2.11. File: prime.c

This file contains all the functions dealing with primetime.

is_prime_time

```
enum prime_time is_prime_time( )
```

Returns

PRIME

if it is primetime

NONPRIME

if it is non-primetime

This function checks to see if it is primetime or not. It uses the information in the global config struct. It will first check if it is a holiday. Holidays are nonprimetime. It will then call *check_prime()* to see whether or not it is primetime.

check_prime

```
enum prime_time check_prime( enum days d, struct tm *t )
```

Args:

d the day to check: SATURDAY SUNDAY or WEEKDAY

t the current time in a struct tm

Returns

PRIME or NONPRIME

The function will return PRIME or NONPRIME depending on the status of primetime The function first checks for all or none status. It will nextly check if primeime does not cross a day boundry (i.e. primetime is 0700-1800). It will then check if primetime is less then one hour. Finally a check if primetime crosses a day boundry(i.e. 2200-0400).

is_holiday

```
int is_holiday( int jdate )
```

Args:

jdate the julien date

Returns:

True if it is a holiday

This function looks though the holiday list to see if today is a holiday

parse_holidays

```
int parse_holidays( char *fname )
```

Args

fnamethe name of the file to parse

Returns

success/failure

This function will read in and parse the holidays file. It will first check for the first word to be 4 numbers. This will be the year for prime/nonprime. Next it will check for YEAR, to set teh current year. It will then check for weekday/saturday/sunday to set primetime/nonprime (different format the above). Finally it will read in the holidays. The parser will ignore the string "HOLDAYFILE_VERSION1" It is part of the spec for the UNICOS 8 holidays format.

load_day

```
int load_day( enum days d, enum prime_time pr, char *tok )
```

Args:

d the day to load

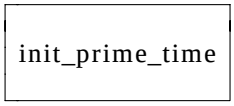
pr PRIME/NONPRIME

tok the time or "all" or "none"

Returns:

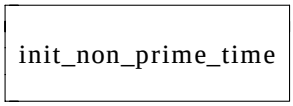
success(0) / failure(-1)

This function will set a primetime or nonprimetime values for a day.




```
void init_prime_time( )
```

This function is called at the beginning of prime time. currently it only changes the scheduling policy bits and the sort to the primetime values.

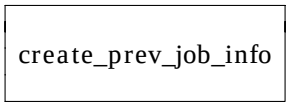



```
void init_non_prime_time( )
```

This function is called at the beginning of non prime time. Currently it only changes the policy bits and the sort to the nonprimetime value.

6.3.2.12. File: prev_job_info

This file contains all the functions for creating and destorying prev_job_info structs.




```
prev_job_info *create_prev_job_info( job_info **jinfo_arr, int size )
```

Args:

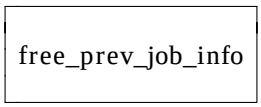
jinfo_arr
array of jobs

size size of the array or {UNSPECIFIED} if unknown

Returns:

newly created and filled prev_job_info array

This function will allocate a new prev_job_info array and fill it with the jobs in jinfo_arr. If size is set to {UNSPECIFIED} the jobs will be counted. The name, resused, and account fields in jinfo_arr will be cleared so they will not be freed at the end of the scheduling cycle.




```
void free_prev_job_info( prev_job_info *pjinfo )
```

Args:

pjinfojob to free

This function frees all the memory used by a prev_job_info struct. Note that it does not free the structure its self. That is part of an arary and will be freed later.

free_pjobs

```
void free_pjobs( prev_job_info *pjinfo_arr, int size )
```

Args:

pjinfo_arr
the array to free

size the size of the array

This function calls *free_prev_job_info()* on every job in pjinfo_arr and then frees the array.

6.3.2.13. File: dedtime.c

This file has all the functions which are specific to dedicated time support

parse_ded_file

```
void parse_ded_file( char *filename )
```

Args:

filename
The name of the file to parse

This function will parse a dedicated time file in the format of
MM/DD/YYYY HH:MM MM/DD/YYYY HH:MM
If the two digit format is used(which is the wrong format) and the year 2000 is shortened to 00, it is smart enough to turn that into the correct date. It does this by checking if it is smaller then some year in the past (90... why 90? why not?), and adding 100 to it. Note this will break if the year 2100 is shortened to 00.

The function will use mktime to turn the date into a UNIX time_t and store it in the global config data structure. Finally it will sort the dedicated times. Zero is a non valid dedtime, it is sorted to the end of the array.

is_ded_time

Returns

1 if it is dedicated time

0 if it is not dedicated time

This function checks if it is dedicated time.

6.3.2.14. File: node_info.c

This file contains all the functions which create, handle, and free node_info structures. It also contains some functions to handle load balencing.

query_nodes()

```
node_info **query_nodes( int pbs_sd, server_info *sinfo )
```

Args:

pbs_sd
: communication descriptor to the pbs server

sinfo
: The server the nodes are associated with

Returns
: An array of nodes which are associated with the server

This function will call *pbs_statnode()* and then convert the batch_status which is returned into an array of nodes. It does this by looping through the linked list returned by *pbs_statnode()* and counting the elements. It will use that count to allocate the array for the nodes. It will then loop through the linked list a second time calling *query_node_info()* on each element. Also, it will call *talk_with_mom()* to get all the information from the resource monitor on the node.

query_node_info()

```
node_info *query_node_info( struct batch_status *node, server_info *sinfo )
```

Args:

node
: The batch_status node returned from the pbs server

sinfo
: The server the node is associated with

Returns
: a node_info with all the information from the batch_status

This function will loop through the attributes in the batch_status struct and set the apporprite values in the node_info.

new_node_info()

```
node_info *new_node_info()
```

Returns
New node_info struct

This function will create a new node_info struct and initialize the values

free_nodes()

```
void free_nodes( node_info **ninfo_arr )
```

Args
ninfo_arr
The array of nodes to free

Call *free_node_info()* on every member of the array and then free the array itself.

free_node_info()

```
void free_node_info( node_info *ninfo )
```

Args
ninfo
The node to free

Free all the memory used bu a node_info structure

set_node_type()

```
int set_node_type( node_info *ninfo, char *ntype )
```

Args:

ninfo
The node to set type

ntype
The node type

Returns
non-zero on error

This function will set one of the nodes type fields (is_timeshare or is_cluster).

set_node_state

```
int set_node_state( node_info *ninfo, char *state )
```

Args:

ninfo
The node to set state

state
The State

Returns
non-zero on error

This function will set the state bits on the node by breaking the state string by commas and then setting the correct state bit for each state listen in the string.

talk_with_mom()

```
int talk_with_mom( node_info *ninfo )
```

Args:

ninfo
The node to get information from its mom

Returns
non-zero on error

This function will connect to the nodes mom and get the resources that are defined in the global res_to_get. The information is then processed and converted into the correct types and assigned to the node.

node_filter

```
node_info **node_filter( node_info **nodes, int size,
             int (*filter_func) (node_info*, void*), void *arg )
```

Args:

nodes
: the array of nodes to filter

size
: the number of nodes in the array

filter_func
: a pointer to a function which will be used to filter the nodes

arg
: an optional arg to be passed to the filter_func

Returns
: filtered array

This function will call the filter function each element in the array. If the filter function returns a non-zero value, the element is included in the new array. The array is initially allocated to the entire size of the original array, and then realloced to the final size after the new array is complete.

is_node_timeshared

```
int is_node_timeshared( node_info *node, void *arg )
```

Args

node
: the node to check if it is timeshared or not

arg
: Unused

Returns

1 if the node is timeshared

2 If the node is not timeshared

This function checks if a node is timeshared or not. It is used in conjuction with node_filter

find_best_node

```
node_info *find_best_node( job_info *jinfo, node_info **ninfo_arr )
```

Args

jinfo
: The job to find the best node for

ninfo_arr
: The array of nodes to find the best node for the job

Returns
: the best node to run the job on

This function will search through all the nodes to find a node which the job has requested the same arch and the node has enough memory. It will find the first node which the added load will not raise the node above the ideal load level. If no such node exists, then find the first node which the added load will not raise it above its max load. The best node is returned.

find_node_info()

```
node_info *find_node_info( char *nodename, node_info **ninfo_arr )
```

Args

nodename
: the node to find

ninfo_arr
: the array of nodes to look in

Returns
: the found node or NULL if not found

Look in the node array and see if the node exists. If it does, return it, if it doesnt, return NULL

[This page is blank.]

7. Resource Monitor

The Resource Monitor is an adjunct to the Job Scheduler. The Resource Monitor daemon provides the scheduler with information about resources on the local system.

7.1. Resource Monitor Overview

The Resource Monitor is part of **pbs_mom**. It listens for input on a specified socket, and responds with a list of resource names and values. The resource monitor can respond to requests from many process, but the socket used is privileged so only a root process can connect.

> Note that pbs_mom no longer deals with allocation of execution nodes. That function has been moved to pbs_server as part of the full parallel awareness features introducted in release 1.1.12.

7.2. Packaging

This chapter of the IDS only discusses the parts of pbs_mom which retain to the Resource Monitor function. The other pieces of pbs_mom are related to job execution. These are discussed in the following chapter entitled **MOM - Machine Oriented Miniserver**.

7.3. Program: pbs_mom

The Resource Monitor portion of pbs_mom consists of an initialization section and shares a single main loop. During the initialization phase, pbs_mom processes the input line and calls *init_network()* to begin listening for clients. The main loop consists of waiting for a message from a client by calling *wait_request()* which will read the input and call a routine to process the request. This routine will obtain the required resource values, then send the information back to the client.

The Resource Monitor may also respond to a reconfiguration command by reading a specified resource file.

7.3.1. Configuration File

The configuration file provides a means to add resource names to the Resource Monitor and also cause functions to be called. This is described in the **pbs_mom** man page.

7.3.2. External Interfaces

The Resource Monitor communicates with the Job Scheduler using the Reliable Packet Protocol (RPP) routines in the PBS net library. Communication from the scheduler to the resource monitor consists of a list of resource names. The resource monitor responds with a list of name/value pairs.

- All information is passed as strings.
- All numeric values are in decimal.
- Time values are in seconds.
- Size (memory/disk) values are in kilobytes with the “kb” appended.

7.3.2.1. Scheduler to Resource Monitor communication

Scheduler to Resource Monitor messages consist of a header, followed by a message body. The format of the message is:

> header, containing command:
> {RM_CMD_CLOSE}, {RM_CMD_REQUEST}, {RM_CMD_CONFIG} or {RM_CMD_SHUTDOWN}

command body

The body of the message has a different usage for each command. For the RM_CMD_CLOSE and RM_CMD_SHUTDOWN commands, the body is ignored and should be zero length.

For the RM_CMD_REQUEST command, the body consists of a number of strings listing resource requests. Each string has the following format:

`name[qualifier=value][qualifier=value]...`

The qualifier/value pairs are enclosed in square brackets and are optional.

For the RM_CMD_CONFIG command, the body should have a single string containing the full path name of a configuration file to read.

7.3.2.2. Resource Monitor to Scheduler communication

Resource Monitor to Scheduler messages consist of a header, followed by a message body. The format of the message is:

header, containing result: [RM_RSP_OK] or [RM_RSP_ERROR]

response body

If the command received was RM_CMD_CLOSE, no response will be returned. If the command received was RM_CMD_REQUEST, the response body will consist of the same list of resources which was sent in the command body with each one followed by an equal sign (=) and a value. Each line in the response body has the form `resource=value`. If no value can be returned, the character following the equal sign is a question mark (?) followed by a space and an error number: [RM_ERR_UNKNOWN], [RM_ERR_BADPARAM], [RM_ERR_NOPARAM], [RM_ERR_EXIST], or [RM_ERR_SYSTEM].

If the value is a single entity, the character following the equal sign will not be a space. If the value is a list, the character following the equal sign will be a space and each list entity will be separated from the next with another space.

If any other command was received, the response body will be zero length.

7.3.2.3. Communication Library

To simplify communication with the resource monitor, a Resource Monitor (RM) library has been provided to handle the details of the protocol described above. The Reliable Packet Protocol (RPP) and Data Is Strings (DIS) librarys are used as well.

7.3.2.4. Signal Handling

The Resource Monitor, pbs_mom, can be commanded to re-read the configuration file which was last read by sending it a SIGHUP signal. If no configuration file has ever been read, no action will take place. An orderly shutdown of the Resource Monitor, pbs_mom, will take place if a SIGINT or SIGTERM signal is received. Several other signals may be defined that also cause an orderly shutdown. These are SIGXCPU, SIGXFSZ, SIGCPULIM, SIGSHUTDN, and SIGINFO.

7.3.3. File: resmon.h

This file defines several structures which will be used throughout the code as well as some constant values such as error codes.

The *rm_attribute* structure is used to pass name/value pairs from square bracket enclosed strings in a request to lower level routines in a convenient form.

```
struct  rm_attribute {
    char*a_qualifier;
    char*a_value;
};
```

The field *a_qualifier* points to the name to the left of the equal sign. For example, the string `[proc=1234]` could be sent as a qualifier for the *mem* request. Here, *a_qualifier* would point to the string `proc` and *a_value* would point to `1234`.

The *config* structure is used to save a name to be used as a key for searching and a value or function call to provide an "answer" for the name in question.

```
typedef char*(*confunc) _A((struct rm_attribute *));
struct  config {
    char*c_name;
    union   {
        confunc c_func;
        char*c_value;
    } c_u;
};
```

For example, suppose the name `Informix` is found in the config file followed by the value 4.10.UD2 for the version. In this case, *c_name* would point to `Informix` and *c_value* would point to `4.10.UD2`. In the case of a name that will have a routine provide a value, the field *c_func* is used to provide a pointer to the function.

7.3.4. File: mom_main.c

This file contains the routines needed for communication and processing an array of configuration elements (names and values).

main()

```
main(int argc, char **argv)
```

Description:

Process command line arguments, and call *read_config()* to read any config files specified. Set up to ignore or catch signals. Call *dep_initialize()* to perform initialization processing based on machine type. Initialize the network communications by calling *init_network()* in the PBS net library. Enter an infinite processing loop which calls *wait_request()* with *get_request()* given as the routine to call to handle a request. Each time a network event or timeout takes place, the routine *end_proc()* is called to do periodic processing. The only machine that takes advantage of this feature right now is the C90. Others just have a stub.

read_config()

```
int read_config(char *file)
```

Returns:

0 on success or 1 on failure.

Description:

If the value for the parameter *file* is not NULL, save the string it points to as the last seen configuration filename. If *file* is NULL, use the previously saved configuration filename. Open and read the configuration file. Save the names and values in a linked list so we can count the number of entries and allocate an array to hold them. If the name

starts with a dollar sign ($), this is an entry which should be found in an internal table and result in a function call. After reading the file, create an array, copy the list elements to the array and free the list.

addclient()

```
int addclient(char *name)
```

Args:*name* is the hostname to be added to the list of hosts which will be allowed to make requests of Mom. The routine **gethostbyname()** is called and the IP address of the host is stored in the array *okclients*.

setlogevent()

```
static u_long setlogevent(char *value)
```

Args:*value* in either decimal or hex to which the log event mask is set.

Sets the external long integer *log_event_mask* to the value. Returns 0 if an error in the value such as an illegal character; returns 1 if ok.

restricted()

```
static  u_long restricted(char *name)
```

Args:*name* is the name of a host.

The named host is allowed to query internal or static resources, but not any that require the execution of a script. This was provied to allow xpbsmon to obtain information about nodes in a cluster. The name is added to the *maskclient* array. A connecting host is check against this array in *bad_restrict()*.

cputmult()

```
static u_long cputmult(char *value)
```

Args:*value*is

The multipler is used to adjust the measured/charged cput against a faster or slower base system.

wallmult()

```
static u_long wallmult(char *value)
```

Args:*value*is

The multipler is used to adjust the measured/charged wall time against a faster or slower base system.

usecp()

```
static u_long usecp(char *value)
```

value is a string containing two tokens seperateh by white space.

This routine parses the $usecp config file entry. Value is broken into the two tokens. The first token is of the form `hostname:/file/path`. The second token is `/alternate/path`. The host name is seperated from the /file/path and the (now) three parts are stored in an array of structures. This array is used by *told_to_cp()* on behalf of *local_or_remote()* to determine if /bin/cp or rcp should be used to copy files.

rm_search()

```
struct config *rm_search(struct config *where, char *what)
```

Args:The *where* pointer is the beginning of an array of config structures which are to be searched. The *what* pointer is a character string which is the name to search for.

Description:

Enter a loop to check each config entry in where. If one is found with a name field that matches *what*, return that entry. If no match is found, return a NULL pointer.

dependent()

```
char *dependent(char *resource, struct rm_attribute *attr)
```

Args:The *resource* character array is the name of the resource to search for. The *attr* pointer specifies a qualifier/value pair in an rm_attribute structure.

Description:

This is the routine which will report back values for resources. The *search()* routine is used to search the array *dependent_config* of type *struct config* contained in the dependent code. If the search returns a match, the function in the dependent code pointed to by the matching entry in the array is called with *attr* as the parameter.

Return:

A string with the value returned from the dependent function. If no match was found, return a NULL.

initialize()

```
void initialize();
```

Description:
Setup the *common_config* array with the entries for "avail", "reserve", "totpool" and "usepool". Then call *read_nodes()* and *dep_initialize()*.

cleanup()

```
void cleanup();
```

Description:
Free all the memory for the node list and call *dep_cleanup()*.

get_request()

```
void get_request(int fd);
```

Description:
Read the socket to get a request. Check to see if there is any previously saved input from this socket. If there is, add the buffer just read to the saved input. Check to see if this input has an end of packet mark. If not, return to wait to complete the packet. If so, format the reply and write it back. If the request is for a resource list, increment the counter *reqnum* so the dependent routine can tell which "packet number" it is processing. Next, call *getattr()* to read the first parameter, if any. If any other parameters are needed by the dependent routine, it can call *getattr()* with a NULL pointer argument.

getattr()

```
struct rm_attribute *getattr(char *str);
```

Description:
Get an rm_attribute structure from a string. Remember the *str* character pointer in a static variable. If a NULL pointer is passed for the string, use the previously remembered pointer. If the rm_attribute name is "tag:", continue to the next attribute. This allows the use of a special attribute which will be ignored. For this feature to work correctly, the "tag:" attribute must be the first one on the line for a request. This is because *getattr()* saves the strings for the name and value in static strings which will be over written by subsequent qualifiers.

arch()

```
char *arch(struct rm_attribute *attrib)
```

Description:
Return the PBS_MACH string defined in "local.mk".

conf_res()

```
char *conf_res(char *s, struct rm_attribute *attr)
```

Description:

Return a value for a resource from the configuration file. If a match is found in *get_request()* for a resource read from the configuration file, this routine is called to generate the reply. The parameter *s* is the value for the resource in the config file. The parameter *attr* is the pointer to the first attribute from the request given by *getattr()*. If *s[0]* is an exclamation mark (!), this is a shell escape resource. If not, then *attr* must be NULL or an error occurs. This is because a static resource has a fixed value and cannot be modified by an attribute.

If the resource is a shell escape, enter a loop to save all the attributes sent with the query. Then enter a loop to scan the command string passed in **s*. Every time a percent character (%) is found, check to see if a parameter substitution should take place by looking to see if a token follows the percent sign and matches one of the saved attribute names. If so, copy the attribute value into the output string, otherwise, just copy the current character. When the scan is done, check to see if any attributes were not used. If so, return an error. Otherwise, call **popen()** to run a shell with the command generated from the request. Return the first line read from standard out from this command.

7.3.5. File: sunos4/mom_mach.c

This is the code used to report values from a sun workstation. It will be used as an example to make it possible to write code to be used on another type of machine.

dep_initialize()

```
void dep_initialize()
```

Args:None.

Description:

This is one of the external entry points what will be the same name for code written for every type of machine. All the steps required to prepare the dependent section of code should be executed here. In this case, use **kvm_open()** and **kvm_nlist()** system calls to open the kernel for use.

Returns:

Nothing.

getprocs()

```
int getprocs()
```

Args:None.

Description:

This routine fills in an array with the process table of the running system. It first checks to see if the information has already been retrieved by comparing reqnum for equality to a static counter it keeps. If it is equal, the information has already been retrieved and no

further work needs to be done. Counter roll over is not a problem since the comparison is for equality. If it needs to refresh the information, it frees the old array, then reads the kernel to get the current number of process. A new array is allocated to hold the table and the kernel is read to get the information.

Returns:
The number of processes in the table. Zero is returned if an error occurs.

cput()

```
char *cput(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:
Check *attrib* to see is the attributes are okay. There needs to be one attribute with a qualifier of "job" or "proc" with a value that is an integer greater than zero. Call *cput_job()* if the qualifier is "job". Call *cput_proc()* if the qualifier is "proc".

Returns:
A character string giving the formatted response for the cpu time in seconds or NULL pointer if an error occurred.

cput_job()

```
char *cput_job(int jobid)
```

Args:The parameter *jobid* is used to identify a "job". On the sun, it will be compared with the process group of a process. If a match is found, it is considered part of the same job.

Description:
Call *getprocs()* to get the process table. Loop over each process entry and see if it is a member of the job identified by *jobid*. If it is, sum the cpu time used by the process into a counter.

Returns:
A character string giving the number of seconds calculated for the cpu time used by the job or a NULL pointer if an error occurred.

cput_proc()

```
char *cput_proc(pid_t pid)
```

Args:The parameter *pid* gives the pid of the process of interest.

Description:
Call **kvm_getproc()** to get the process with the pid we are looking for.

Returns:
A character string giving the number of seconds calculated for the cpu time used by the process or a NULL pointer if an error occurred.

mem()

```
char *mem(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* to see is the attributes are okay. There needs to be one attribute with a qualifier of "job" or "proc" with a value that is an integer greater than zero. Call *mem_job()* if the qualifier is "job". Call *mem_proc()* if the qualifier is "proc".

Returns:

A character string giving the formatted response for the memory used in bytes or a NULL pointer if an error occurred.

mem_job()

```
char *mem_job(int jobid)
```

Args:The parameter *jobid* is used to identify a "job". On the sun, it will be compared with the process group of a process. If a match is found, it is considered part of the same job.

Description:

Call *getprocs()* to get the process table. Loop over each process entry and see if it is a member of the job identified by *jobid*. If it is, sum the memory used by the process into a counter.

Returns:

A character string giving the number of bytes calculated for the memory used by the job or a NULL pointer if an error occurred.

mem_proc()

```
char *mem_proc(pid_t pid)
```

Args:The parameter *pid* gives the pid of the process of interest.

Description:

Call **kvm_getproc()** to get the process with the pid we are looking for.

Returns:

A character string giving the number of bytes calculated for the memory used by the process or a NULL pointer if an error occurred.

jobs()

```
char *jobs(struct rm_attribute *attrib)
```

Description:

Check to make sure there are no attributes. If so, call *getprocs()* and loop through the

process list skipping those owned by root. For each process job id, check an array of saved job id's to see if it has been encountered before. If not, add the current job id to the array of saved job id's.

Returns:

A string with a space separated list of job id's of all the processes in the system, or a NULL pointer if an error occurred.

pids()

```
char *pids(struct rm_attribute *attrib)
```

Description:

Check to make sure there is only one attribute with a qualifier of "job" and a value greater than zero. If so, call *getprocs()* and search through the process list looking for members of the job specified.

Returns:

A string with a space separated list of pid's of all the processes found to be a part of the job or a NULL pointer if an error occurred.

getanon()

```
int getanon(char *id)
```

Args:The character string *id* is used in logging to identify which routine made the call.

Description:

The kernel maintains an area of general information called anoninfo which is retrieved by this routine. As usual, the counter reqnum is compared to a static counter to see if the information is already in hand.

Returns:

0 if all is well, 1 if an error occurred.

totmem()

```
char *totmem(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check to make sure no attributes have been passed. Then, call *getanon()* to fill in the anoninfo structure which contains the total memory size of the machine.

Returns:

A character string with the total memory in bytes or a NULL pointer if an error occurred.

availmem()

```
char *availmem(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check to make sure no attributes have been passed. Then, call *getanon()* to fill in the anoninfo structure which contains the available memory of the machine.

Returns:

A character string with the available memory in bytes or a NULL pointer if an error occurred.

physmem()

```
char *physmem(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check to make sure no attributes have been passed. Then, read the kernel to get the physical memory size of the machine.

Returns:

A character string with the physical memory in bytes or a NULL pointer if an error occurred.

size()

```
char *size(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* to make sure only one attribute was passed. If so, check the qualifier to be sure it is one we understand: if it is "file", call *size_file()*. If it is "fs", call *size_fs()*.

Returns:

A character string pointer which is returned from the function called, or a NULL pointer if an error occurred.

size_fs()

```
char *size_fs(char *param)
```

Args:The parameter is a character string which specifies the path to check.

Description:

Use **statfs()** to get the information about the path specified.

Returns:

A character string with the file system space available in bytes or a NULL pointer if an error occurred.

size_file()

```
char *size_fs(char *param)
```

Args:The parameter is a character string which specifies the path to check.

Description:

Use **stat()** to get the information about the path specified.

Returns:

A character string with the file size in bytes or a NULL pointer if an error occurred.

idletime()

```
char *idletime(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* to make sure no attributes were passed. Use **opendir()** and **readdir()** to read the /dev directory and **stat()** devices which begin with "tty". Maintain a time value with the maximum access time for each device tested. After checking the "tty" devices, perform the same test on "/dev/kbd" and "/dev/mouse".

Returns:

A character string containing the difference between the current time and the maximum access time from all devices tested. This value is reported in seconds. If an error occurred, return a NULL pointer.

walltime()

```
char *walltime(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* to see is the attributes are okay. There needs to be one attribute with a qualifier of "proc" or "job" with a value that is an integer greater than zero. Call *getprocs* and search though the list of processes for the process or job as specified by the attribute. Call **kvm_getu()** to get the user structure for each process. Check to see if the start time is less than any other process encountered. If so, save the start time of the process being checked.

Returns:

A character string containing the difference between the current time and the smallest start time found. This value is reported in seconds. If an error occurred, return a NULL pointer.

loadave()

```
char *loadave(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* to make sure no attributes were passed. Use **kvm_read()** to get the load average reported by the kernel.

Returns:

A character string containing the load average of the system. If an error occurred, return a NULL pointer.

quota()

```
char *quota(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* for an attribute with a name of "type". This attribute must have a value of one of "harddata", "softdata", "currdata", "hardfile", "softfile", "currfile", "timedata", or "timefile". The next attribute must have the name "dir" with the value being a directory name. This directory specifies the file system to check for quota information. The last attribute must have the name "user" with a value giving a user name or an integer specifying a uid. The system call **quotactl()** is used to get quota information for the user in the specified directory. The type "harddata" returns the hard limit for data storage in characters. The type "softdata" returns the warning limit for data storage in characters. The type "currdata" returns the current usage of data storage in characters. The type "hardfile" returns the hard limit for the number of files. The type "softfile" returns the warning limit for the number of files. The type "currfile" returns the current number of files. The type "timedata" returns the number of seconds that a user has left in the grace period for excessive disk use, or zero if the grace period is not active. The type "timefile" returns the number of seconds that a user has left in the grace period for having an excessive number of files, or zero if the grace period is not active.

dep_cleanup()

```
void dep_cleanup()
```

Args:None.

Description:

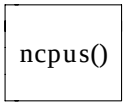
This is another external entry point to the dependent code. Here is where all cleanup operations take place that are specific to the machine of interest. In the case of a sun, all that is needed is to close the kernel device.

Returns:

Nothing.

7.3.6. File: irix5/mom_mach.c

This is the code used to report values from a Silicon Graphics machine. It is very similar to the code for the Sun except SGI IRIX can report the number of cpu's on a host. Also, the methods for getting the information about processes and jobs center around the process file system rather than reading the kernel structures directly. This requires the use of version 5 or later release of IRIX.




```
char *ncpus(struct rm_attribute *attrib)
```

Description:

Since no attributes are legal for this request, check to make sure *attrib* is NULL. Then call **sysmp()** with the parameter `MP_NAPROCS`. Return the value from this call formatted as a decimal number.

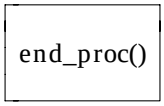
7.3.7. File: solaris5/mom_mach.c

This is the code used to report values from a Sun Solaris machine. It is very similar to the code for IRIX5 except Solaris cannot report any quota information or virtual memory size. As with IRIX5, the methods for getting the information about processes and jobs center around the process file system rather than reading the kernel structures directly.

7.3.8. File: unicos8/mom_mach.c

This file contains the code for the Cray C-90. It has several routines dealing with swap space and is the only machine that uses the "periodic processing" capability of the resource monitor. Another difference for the cray is with the *quota()* routine. It is much more complex then any other machine and can optionally support the Session Reservable File System (SRFS).

This file also provides functions to read the file `/etc/tmpdir.conf` to get the temporary directory names the administrator has set up for SRFS. These currently must be `$TMPDIR`, `$BIGDIR`, `$FASTDIR` and `$WRKDIR`.




```
void *end_proc()
```

Description:

The global variable *last_time* is used to keep track of the last time processing took place. A call to **rtclock()** is made to get the current time in clock ticks. This is compared to last_time to see if the network woke us up before it was time to do something. If so, calculate the value of *wait_time* such that the next wakeup will be timed correctly. This variable is used by the **select()** call in *wait_request()* as a timeout value. If it is time to do something, set *wait_time* to the value of SAMPLE_DELTA which is a define'ed number. Call **tabinfo()** and **tabread()** to get the PWS processor data and the SINFO system data. Set *last_time* to the current time. Calculate the cpu time percentages, filter them and store them in the global variables *cpu_idle*, *cpu_guest*, *cpu_unix*, *cpu_sysw* and *cpu_user*. The method of filtering is to calculate the new value from the current data and the old value as follows:

```
cpu_idle = a * current  +  (1-a) * old
```

The value of a must fall in the range [0-1]. I picked 0.75. Next, calculate the average swap rate since the last call of this routine. Use the same filter operation as above and store the result in the global variable *swap_rate*.

quota()

```
char *quota(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check *attrib* for an attribute with a name of "type". This attribute can have one of the standard quota "type" values given for the other machines. These are "harddata", "softdata", "currdata", "hardfile", "softfile", "currfile", "timedata", or "timefile". The cray supports several others as well. The additional types only operate if the resource monitor is compiled with the symbol SRFS. They are "snap_avail", "ares_avail", "res_total", "soft_res", "delta" and "reserve". The next attribute must have the name "dir" and a value of a directory name or "variable" directory name. These are described below. If the type attribute is one of the SRFS values, there can be no other attributes. If the type attribute is one of the standard values, there must be one more attribute. It can have a name of "user", "group" or "account" and a value of a name or id number. Depending on what the name is, the value is looked up to see if it is valid. This is then used to retrieve the quota information. The standard types have the same meaning as the other machines. The meanings of the SRFS types are taken from the UNICOS header file /usr/include/sys/srfs.h:

```
int   snap_avail; /* number of currently available blocks if a snap     */
                  /* was taken of the system                            */
int   ares_avail; /* snap_avail less unused reserved blocks             */
                  /* the number of blocks available for reservation the */
                  /* sum of ares_avail, delta, and reserved             */
int   res_total;  /* total number of reserved blocks                    */
int   soft_res;   /* set to TRUE if soft reservation is allowed         */
long  delta;      /* over/under subscription delta                      */
long  reserve;    /* buffer for root demanded allocations on SRFS       */
```

The type "soft_res" will return "true" or "false". The values for the rest are converted from blocks to characters.

srfs_reserve()

```
char *srfs_reserve(struct rm_attribute *attrib)
```

Description:

The attributes (there must be at least one) names are passed to *var_value()* to see if the name exists as a defined temp directory. If so, the value is converted to a number and used as a parameter to the system call *quotactl()*. This call is done with the command set to **SRFS_RESERVE** so that a "srfs_assist" mode reservation can be done.

swapused()

```
char *swapused(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Call **tabinfo()** and **tabread()** to get the swapper information. Calculate the number of characters used in the swap areas and format this number in decimal.

cpuidle()

```
char *cpuidle(struct rm_attribute *attrib)
```

Args:The parameter *attrib* is a pointer to the attribute structure returned by *getattr()*.

Description:

Check the global variable *last_time*. If it is zero, return with a system error. This would mean the calls to get the cpu usage information in *end_proc()* had failed and there was nothing to return. Otherwise, format the variable *cpu_idle* and return a pointer to *ret_string*.

var_init()

```
void var_init()
```

Description:

Open and read `/etc/tmpdir.conf`. For each line, ignore it if it begins with a hash mark (#) character. Otherwise, save the temporary directory name and path.

var_cleanup()

```
void var_cleanup()
```

Description:

Free space allocated in *var_init()* for the names and paths.

var_value()

```
char *var_value(char *name)
```

Description:

Search the saved directory names for *name*. Return the value or NULL if it is not found.

7.3.9. File: aix4/mom_mach.c

This is the code used to report values from an IBM 590 workstation running AIX 4.

dep_initialize()

```
void dep_initialize()
```

Args:None.

Description:

Call *open()* to get access to "/dev/kmem' and call *knlist()* to get the name list. The two structures in the kernel we need access to are "vmker" which has memory information, and "avenrun" which has the load averages.

getproctab()

```
int getproctab()
```

Args:None.

Description:

This routine retrieves a table of procsinfo entries. A library call to *getprocs()* is made in a loop which terminates when no more procsinfo entries are available. If there are more entries to retrieve, a call to **realloc()** is made to expand the table size. This table is retained so only the first call should result in several passes though the loop.

7.3.10. File: sp2/mom_mach.c

This is the code used to report values from an IBM SP-2 parallel computer. It is based on the code for the IBM 590 but there are fewer supported functions for this machine because the parallel "resource manager" does not support some important functionality. It cannot return the cpu time or memory utilization for a job's parallel node usage.

getjobstat()

```
int getjobstat()
```

Args:None.

Description:

This routine will get a table of JM_JOB_STATUS entries which give information about jobs running on the nodes. Check *reqnum* to see if data needs to be retrieved. If so, call the library function *jm_connect_ub()* to get access to the "resource manager". Call *jmq_jobs_status()* to retrieve the table of information. Then call *jm_disconnect()* to terminate communication with the resource manager.

This file also contains a function specific to the IBM SP-2 which will remove from consideration any node which is shown by the IBM Job Manager to be busy. This function is called from *nodes_inuse()*.

dep_inuse()

```
int dep_inuse()
```

Description:

Call *getjobstat()* to gather information from the job manager. Then loop thru the node list checking for a JM_JOB_STATUS entry from the job manager that contains a matching node. If one is found, mark the inuse flag true for the node.

8. MOM - Machine-Oriented Miniserver

8.1. Machine-Oriented Miniserver Overview

The purpose of the Machine-Oriented Miniserver (MOM) daemon is to create executing batch jobs, watch over and control their execution and report on their demise to the Batch Server which issued the job to MOM. One MOM exists on each machine under the Batch Server's control. Though the Batch Server maintains responsibility for each batch job it executes, MOM takes care of the housekeeping details required to actually initiate, monitor and clean up after batch jobs. A running job has one or more running tasks. MOM is the parent of each task which runs on its machine. Unlike other batch systems, there is only one MOM for all running tasks, there is no per-task shepherd process.

The MOM daemon has been combined with the Resource Monitor in an effort to consolidate code. The functions will be described separately. MOM also acts as a Task Manager for each job she controls.

The Batch Server uses essentially the same protocol to talk with MOM as the Batch Server's clients use to talk with it. MOM uses restricted interpretations of some of the Batch protocol and makes use of a few special messages. The protocol between MOM and the Batch Server, though, is essentially the same as between the Batch Server and any of its clients. MOM acts as a client of the Batch Server for only one message type. Its purpose is to announce the demise of a batch job.

A Batch Server which is responsible for a simple uniprocessor machine will work with only one MOM. Batch Servers for multiprocessors or for extended families of workstations may deal with multiple MOMs.

8.1.1. MOM's Interpretation of PBS Protocol

MOM interprets a few PBS Protocol messages exactly as does the Batch Server. There are several other PBS Protocol messages which MOM interprets in a more restrictive way than specified in the ERS. Finally, there are a few PBS Protocol messages which are unique to communication with MOM. The desired effect of these PBS Protocol interpretations is to simplify MOM at the expense of requiring a more sophisticated Batch Server. Since there will only be one Batch Server per workstation cluster or distributed memory multiprocessor, and there will be many MOMs, simplifying MOM seems to be a good idea.

8.1.1.1. Unchanged PBS Protocol Messages

The following PBS Protocol messages are interpreted by MOM exactly as specified in the ERS explanation of the protocol.

- Message Job
- Signal Job
- Status Job

8.1.1.2. Re-interpreted PBS Protocol Messages

The following paragraphs cover MOM's restricted PBS Protocol interpretations.

8.1.1.2.1. Modify Job

Of all the aspects of a batch job which can be modified by the Modify Job message, MOM only supports reducing the limits of a running job, and that only if it is possible for the resident machine. Any other attempted modification will result in an error response.

8.1.1.2.2. Delete Job

If the designated job has been checkpointed and if MOM has the checkpoint file, MOM will honor a Delete Job message by deleting the file. This situation will only arise if the Batch Server is configured to let MOM keep checkpoint files. The alternative Batch Server configuration will ask MOM to send it any checkpoint files. If the designated job is running or is unknown to MOM, an error results.

8.1.1.2.3. Hold Job

If the designated job is running and if checkpoint is supported on the resident machine, MOM will checkpoint the job. The checkpoint file may later be sent back to the Batch Server or it may be left in place, at the whim of the Batch Server. If the designated job is not running, or if the target job cannot be checkpointed by the resident machine, an error results.

8.1.1.2.4. Queue Job

Rather than put a job received through the Queue Job protocol into a queue, MOM puts it into execution. If a corresponding checkpoint file exists, the job is actually restarted.

8.1.1.2.5. Server Shutdown

If any job is still running, MOM reports an error. Otherwise, she exits. It is the responsibility of the Batch Server to send MOM a Signal Job message or a Hold Job message for each running job before sending the Server Shutdown message.

8.1.1.3. Unused PBS Protocol Messages

Much of the PBS Protocol has no meaning to MOM. Any of the following messages, if received by MOM, will result in an error response.

- Manager
- Move Job
- Rerun Job
- Run Job
- Select Jobs
- Status Queue
- Status Server
- Locate Job
- Track Job
- Pull Job
- Register Job

8.1.1.4. MOM-specific PBS Protocol Messages

The following PBS Protocol messages are used exclusively by the Batch Server while acting as a client of MOM.

8.1.1.4.1. Copy Files

The Copy Files message provides MOM with a list of filename pairs, a direction flag, a user identification on MOM's machine, a file owners name, and a hostname. MOM treats the first name of each pair as a filename local to MOM's machine. It treats the second name as a filename local to the named host. MOM arranges for a copy to be made in the direction specified by the direction flag. MOM acts in the name of the identified user on MOM's machine.

When files are being copied outward from MOM and the copy is successful, MOM deletes the file on her machine.

If the file transfers cannot take place, an error response is given. If the transfer that failed is in to MOM's machine, then MOM deletes all the files which were copied in prior to the failed file.

8.1.1.4.2. Delete Files

The Delete Files message provides MOM with a list of filenames and a user identification on MOM's machine. MOM interprets the filenames as the names of local files, and deletes them. MOM acts in the name of the identified user.

If the files cannot be deleted, an error response is given.

8.1.1.5. MOM-specific PBS Protocol Message Sent by MOM

The following PBS Protocol message is used exclusively by MOM while acting as a client of the Batch Server.

8.1.1.5.1. Job Obituary

MOM uses the Job Obituary message to tell the Batch Server that a batch job has ended and how. The message contains the job_id, the termination status and the total resource utilization of the process which was the job's session leader. The termination status is the value returned through the integer pointer which is the argument of the POSIX wait() function.

8.2. Program: pbs_mom

8.2.1. Overview

8.2.2. Packaging

MOM is composed of three parts,

- PBS-generic routines for communication and server operation, drawn from the Batch libraries under directory *src/lib/** and from Batch Server files from directory *src/server*,
- Machine-independent, MOM-specific information, in the files mom_func.h and various C source files located in the *src/resmom* directory, and
- Machine-dependent, MOM-specific information, in the files mom_mach.h, mom_mach.c, mom_start.c, and pe_input.c.

8.2.3. External Interfaces

MOM has the following external interfaces:

- Arguments supplied by the pbs_mom command line,
- Inter-Server Protocol messages,
- Resource Monitor Protocol messages,
- Batch Protocol messages received from the Batch Server, and
- Task Manager messages exchanged with running jobs and other MOMs.

8.2.4. Machine-independent Files

8.2.4.1. File: pbs_mom.h

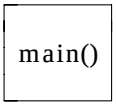
The file *src/include/mom_func.h* contains the machine-independent macro definitions which are unique to MOM as well as the function prototypes for MOM.

8.2.4.2. File: job.h

The file *src/include/job.h* contains many structure and flag defines for both the Batch Server and MOM. The structures for MOM have become more complicated with the need to track **tasks**. The job structure contains a number of fields not needed in the Batch Server. It has entries for the number of nodes in the job, the local MOM's node id, an array of node resources, an array of node entries, and a list of task structures. The array of node entries each give the node id and host name for the node they represent. They also have an RPP stream number and a list of events which are being waited for from other MOM's.

8.2.4.3. File: mom_main.c

The file *src/resmom/mom_main.c* contains the machine-independent source code which is unique to MOM.




```
main(int argc, char **argv)
```

Args:

argc The count of the number of arguments.

argv A null-terminated list of character pointers. If *argv* points to option key letters and arguments, see the pbs_mom(8B) man page.

Return:

zero if success.

non-zero
an error code defined in pbs_errno.h.

Start Up

Mom must be run with a real and effective UID of root. Her service port and that of the server is obtained by calling *get_svrport()*. Mom then processes the options specified on the command line. Resource limits which will be inherited by the job and might not be reset are set to unlimited. Mom then sets up paths and checks the security of her files and directories.

Local host and the name of her host obtained are added to the list of systems which may contact Mom, see *addclient()*. If a configuration file was specified with the -c option, the config file is processes by calling *read_config()*.

The routine *mom_open_poll()* is called to initialize the machine dependent polling routines.

The routine *init_abort_jobs()* is called if jobs were running when mom last ceased operation. This routine will kill those jobs.

Main Loop

In normal operation to place a job into execution, MOM will determine if the job is to run on more than one node by checking the attribute "exec_host". If so, the MOMs on the other hosts are contacted to request they join the job. If this succeeds, or no other nodes are part of the job, MOM will fork herself, see *start_exec()*. The child process will establish the script as standard input, and setup standard output and error as required by the job. It will then set by whatever means are supported on the system the resource limits of the job. The child will will then "exec" the shell on top of itself and become the job.

The single parent MOM after forking the child will determine in *mom_do_poll()* if any of the resource limits cannot be enforced by the system directly and therefore require MOM to monitor the usage by the job by polling. If polling is required, the job is added to a special list. In the main loop, once every {CHECK_POLL_TIME} (120) seconds, Mom will obtain the process infor-

mation for all running processes by calling *mom_get_sample()*. For all running jobs, their resource usage is updated by calling *mom_set_use()*. *rpp_io()* is called to see if any RPP i/o is required. If any (running) job has the Mom flag {MOM_NO_PROC} set, then for each task in the job the session leader's existance is verified by calling kill(2) with signal zero (SIGNULL). If -1 is returned and errno is ESRCH, then the process no longer exists (even as a zombie). We force the task in {TI_STATE_EXITED} state. This allows Mom to catch the termination of tasks for who she is not the parent (say after a restart). Note, the MOM_NO_PROC flag is set in *cput_sum()* if no processes are found when summing up the jobs cpu usage.

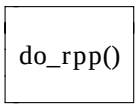
If checkpointing is supported ...
In the main loop when jobs are running, MOM will determine if there is any need to checkpoint jobs by looking for a non-zero *ji_chkpttime*, the checkpoint interval time. If set, MOM checks *ji_chkptnext* to see if the time for the next checkpoint has been reached. If so, that time is updated to now + ji_chkpttime and *start_checkpoint*() is called to checkpoint the job.

Then for each "polled" job, MOM will call *mom_over_limit()* to determine if any of the usage is over limit. When that occurs, a message is written on the standard error file by calling *message_job()* and *kill_job()* is called to terminate the job. kill_job() will be called up to three times (due to problems with IBM poe on the SP-2). The first two times, kill_job() is a called with SIGTERM. The last time, MOM gets serious and calls it with SIGKILL.

When a job terminates, the SIGCHLD signal is sent to MOM. The post job processing requires a two step approach, so the SIGCHLD signal handler only sets a flag, termin_child, which indicates that some child process has terminated. The child may not even be a task, but some other child process of MOM. However, the terminated process (task) cannot be reaped immediately. Reaping a child on a system where resource usage is maintained in the process table cause the process table entry to be freed and the information lost. Before the *wait()* is called, MOM on finding that termin_child is set, will call *scan_for_terminated()* to get the latest resource usage and then determine which task (if any) terminated. The exiting_tasks flag is set. This flag may also be set on recovery. When MOM finds this flag set, *scan_for_exiting()* is called to post process any jobs marked as exited. MOM then sits and waits for another service request.

Termination

When MOM exits as a result of an SIGTERM, *mom_close_poll()* is established in *mom_open_poll()*, such as closing access to the kernel. Then MOM attempts to kill any running job and marks each one as exiting. Clean up will occur when MOM is restarted.

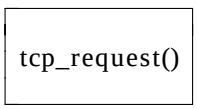



```
int do_rpp(int stream)
```

Args:

stream
a stream index to read.

Read the stream to get the protocol number. Read the protocol version number and call **rm_request()** if it is a Resource Monitor request, or **im_request()** if it is an Inter-MOM request, or **is_request()** if it is an Inter-Server request.

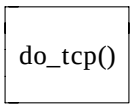



```
int rpp_request(int fd)
```

Args:

fd not used.

Input is coming from an RPP stream. Call **rpp_poll()** to get the stream index to process. If it is a valid stream, call **do_rpp().** Continue this until there are no more streams to process.

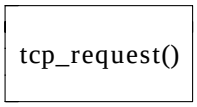



```
int do_tcp(int fd)
```

Args:

fd a file descriptor to read.

Read the file descriptor to get the protocol number. If the call to **disrsi( )** returns DIS_EOF, the connection is closed. If it returns DIS_EOD, there is no more data, but the connection is still open. Read the protocol version number and call **rm_request()** if it is a Resource Monitor request, or **tm_request** if it is a Task Manager request.




```
int tcp_request(int fd)
```

Args:

fd a file descriptor to read.

Input is coming from a tcp stream as either a Resource Monitor request or a Task Manager request. Check that it is coming from a machine in the okclients array then go into a loop calling **do_tcp** until there are no more messages to process.


```
static int read_config(char *file)
```

Args:

file name of the configuration file specified on the -c option.

Returns:

zero if ok, non-zero otherwise. Errors are logged.

Each line in the configuration file is read. Lines starting with a hash mark (#) are comments and are ignored as are null lines.

Non-comment lines can have a static resource definition or a command that causes a function to be called with a token. A resource definition is described in the Resource Monitor IDS. A command begins with a dollar sign ($). The command names and the functions they call are as follows:

command	function
clienthost	addclient
restricted	restricted
logevent	setlogevent

addclient()

```
static u_long addclient(char *hostname)
```

Args:

hostname
name of a host to added to the allowed clients of MOM.

Returns:
the IP address of hostname if ok, zero otherwise.

The routine *get_hostaddr()* is called to return the IP address of the listed host. Any invalid or unknown hosts causes addclient to return 0 and MOM shuts down. Valid addresses are added to the global binary tree *okclients*. If a signal causes MOM to re-read the config file, the IP addresses previously in *okclients* are not deleted. MOM must be restarted to remove an IP address from those allowed to connect.

restricted()

```
static u_long restricted(char *name)
```

Args:

namethe name to be matched against the name of any host sending a request with a non-privleged port number.

Returns:
non-zero if ok, zero otherwise.

The first character of *name* can be a star (*) to allow wildcard matches of hostnames. For example, if *name* is "*.spam.com", any host in the domain "spam.com" will be allowed to perform restricted queries.

setlogevent()

```
static u_long setlogevent(char *value)
```

Args:

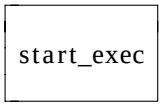
valuenew value for log_event_mask

Returns:
non-zero if ok, zero otherwise.

Set a new value for log_event_mask.

8.2.4.4. File: start_exec.c

The file *src/resmom/start_exec.c* contains machine independent functions used to place a job into execution.




```
void start_exec(job *pjob)
```

Args:

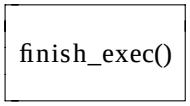
pjob pointer to job to place into execution.

This function is called from MOM's version of *req_commit()* within the file req_quejob.c. The purpose is to place the job into execution. The following are the steps take:

The JOB_ATR_Cookie attribute is set for the job. The cookie is used to validiate inter-mom and task management (tm_ API) calls. By calling *job_nodes()*, the nodes allocated to the job are determined by examining the JOB_ATR_exec_host attribute. Note that the flag {JOB_SVFLG_HERE} was set back in req_commit() when the job was received. It indicates this Mom is designated "mother superior" for the job. Also note that start_exec() is not called on the sister nodes.

If other nodes are to be part of the job... Two sockets (for standard out and error) are opened with will be used by pbs_demux to collect output from tasks on the other nodes. The port number bound to the sockets are saved in *ji_stdout* and *ji_stderr*. The other nodes are sent a inter-mom message with the job information including the above ports, the logical node numbers, and the job attributes.

If the job will only run on the local machine, *finish_exec()* is called. Note, for multiple node jobs, finish_exec() is called when all sisters have acknowledged the JOIN_JOB message, see *im_request()* in mom_comm.c.




```
void finish_exec(job *pjob)
```

Args:

pjob pointer to job to place into execution.

Start a job running by establishing the resource usage limits, setting up the standard output and error files for the job, connecting the script as the standard input to the job and then invoking the login or user specified shell to interpret the script. When called, MOM is running as a single process with root privilege and her current working directory is her private directory.

If other nodes are allocated to the job, this is the Mom which will run the job script. She is known as "Mother Superior". Mother Superior obtains the port number associated with the sockets allocated for communication between the job and the pbs_demux process (which will be started later). The port numbers must be passed to the other Moms associated with the job. The flag {MOM_HAS_NODES} is set in *ji_flags* of the job structure.

The next thing finish_exec() does is obtain the password entry for the user specified by the server in the job attribute JOB_ATR_euser. This is the user name under which the job should be executed. The corresponding uid is save later in the job structure for future use, see *check_pwd()*.

The machine dependent function *mom_do_poll()* is called to determine if the newly started job has resources which require MOM to poll its usage. If it returns true, or if the job has more than one node, then the job is added to a special polling list as described under mom_main.c

If checkpoint is enabled ...
If the job's checkpoint attribute, JOB_ATR_chkpnt, has a value of c=nnn then the user is requesting periodic checkpoint at an interval of nnn minutes. The interval is set into the job structure in *ji_chkpttime* and *ji_chkptnext* is set to time now plus the interval.

If checkpoint is enabled ...
If the job is marked as having been checkpointed, {JOB_SVFLG_CHKPT} is set in *ji_svrflags*, and if the restart file exists, then the machine independent routines *site_mom_prerst*() and *mom_restart_job*() are called to restart the process. The time the job was started which is kept in ji_stime is adjusted to the current time minus the wall clock time the job ran before it was checkpointed and held. This is so the held time is not counted against the job's wall clock time. On the CRAY, if the job is in a suspended state when restarted, the job start time is not adjusted as it will be when the job is resumed.

If the restart fails for a "permanent" reason, the job is marked as exiting and the exit status is set to {JOB_EXEC_FAIL}. If the reason is temporary, see restart() on the Cray, the job is marked as exiting, but the exit status is set to {JOB_EXEC_RETRY} which directs the server to re-queue the job.

If Interactive support is enabled ...
If the job is an interactive job, attribute JOB_ATR_interactive is non-zero, A master side pseudo tty is opened by calling *open_master()*. This is done before finish_exec() forks because the name of the slave tty must be saved in the job's output path attribute JOB_ATR_outpath so that it can be found if a message (qmsg) is sent to the job, see *message_job()*.

If not interactive ...
and if {SHELL_INVOKE} is defined as 1, the default, a pipe is created with will be the shell's standard in and will be used to pass the name of the job script. This is equivalent to saying:
`echo script_path | shell`

A pair of pipes is created for communication between the child of MOM and MOM. MOM then forks the child process which will become the job. Standard files and resource limits must be established by the child process which becomes the job. Certain conditions may exist which result in the failure to establish the files or limits. If the conditions are temporary, the job should be re-queued by the server. If the conditions are permanent, then the job should be aborted. The child returns via the pipe either the **session/job id** (greater than zero) if the job is placed into execution, {JOB_EXEC_RETRY} if a temporary condition prevented failure, or {JOB_EXEC_FAIL} if the job cannot ever be run. When MOM reads from the pipe either of the status that indicates the job did not execute, then that value is saved as the job exit status and returned to the server. Note that the true exit status of a job, the argument to the exit() call, cannot be negative. Thus JOB_EXEC_RETRY and JOB_EXEC_FAIL are negative values to indicate to the server that these are from MOM. After the session id or job status is read from one pipe, MOM will acknowledge by returning the same value back to the child on the other pipe. This releases the child to continue with **exec()** or **exit()**. This prevents the possibility of SIGCHLD interrupting the pipe read and confusing parent mom.

After forking the child and receiving a positive return, the parent MOM will record the job start time (used to determine wall clock execution time), and the session id of the child. The *global id* which was created to hold an SGI Array Session Handle, ASH, is saved and the task state is set to {TI_STATE_RUNNING}.

If there is more than one node, the sockets created for communication between the job and pbs_demux are no longer needed by Mom herself, so they are closed. The associated port numbers are saved in the job structure.

At this point, the main MOM returns to req_commit() and then to the main loop.

The **child process of MOM**
does the following:

1. Insures that the file descriptors for the pipes are greater than 2 (standard error).
2. Determines the correct shell to invoke by calling the machine dependent *set_shell()*.
3. Sets up a whole slew of environment variables, including those passed with the job in the attribute JOB_ATR_variables. HOME, LOGNAME, PATH, SHELL, and USER which would be set by login(8). Other variables set are those mandated by POSIX batch, including "PBS_ENVIRONMENT=PBS_BATCH" and for compatibility with NQS, "ENVIRONMENT=BATCH". In addition, PBS_JOBCOOKIE, PBS_NODENUM, PBS_TASKNUM and PBS_MOMPORT are set so the Task Management library can communicate back with MOM. Also, the machine dependent routine *set_mach_vars()* is called to set any machine specific variables (typically none).

4a. If interactive support is enabled...
If the job is interactive, the environment variable PBS_ENVIRONMENT is set to PBS_INTERACTIVE. The name of the host where qsub is running is extracted from the environment variable PBS_O_HOST and the port number to which qsub is listening is taken from the interactive attribute JOB_ATR_interactive. *conn_qsub()* is called to open a network connection back to qsub. Over the connection we send the job id as a simple validation as to who we are. Qsub sends the window size, terminal type, and terminal characters (special characters) of its controlling terminal. An alarm is established around the code which connects qsub and reads the terminal characters. This prevents a suspended qsub or a network problem from holding up MOM which in turn would hold up the server and the scheduler who is waiting for the pbs_runjob() to be acknowledged.

set_job() is called to establish a new session. This must be done to free the job of any prior controlling terminal. The ownership of the slave pseudo tty is changed to the user and mode is set to 0620. Then the slave side of the pseudo terminal is opened, it becomes the controlling terminal of the job. For the CRAY only − ioctl calls are made to force the slave to be the controlling terminal.

The child process forks a grandchild. This process becomes the writer process, *mom_writer()*. It reads from the master tty (data written by the job on the slave side) and sends it over the socket to qsub. The original child of MOM, parent to the grandchild, sets up stdout and stderr for the prolog to run. This is why the writer process was started so the output can be delivered to the user's screen. The function *run_pelog()* is called to execute the prologue script, if one exists. Run_pelog() is called with {PE_PROLOGUE} to signify the prologue and {PE_IO_TYPE_ASIS} because the standard output and error files of the job are already opened on descriptors 1 and 2. Note, the child's current working directory is still mom_priv. After the prolog is run, it forks again to create grandchild number two. This will become the job while the original child will become the reader process, *mom_reader()*, which reads the input from qsub (socket) and passes it to the job by writing on the master socket. When mom_reader() exits, the pseudo tty is reset to root ownership and mode 0666.

4b. For non-interactive jobs ...
The environment variable PBS_ENVIRONMENT is set to PBS_BATCH.

Standard output and standard error are established depending on the setting of the JOB_ATR_join attribute by calling *open_std_file()*.

The call to *run_pelog()* is made in a similar fashion to the interactive case except this takes place before the call to *set_job()*. This is so the time spent running the prolog will not be charged to the user's job.

The machine dependent function *set_job()* is called to establish the session and process group id. This is machine dependent because a few vendors (such as CRAY) support a "job" concept.

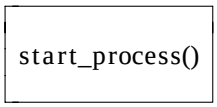
5. For both interactive and normal batch ...
   Resource limits are established by calling *mom_set_limits()*. If this fails, either {JOB_EXEC_RETRY} or {JOB_EXEC_FAIL} is returned to MOM depending on the permanence of the failure and the job exits. Interactive jobs cannot tolerate a JOB_EXEC_RETRY attempt because they have lost the chance to connect with **qsub** so a JOB_EXEC_FAIL will be returned for either case.

6 The argv array to pass to the shell is established. Note that the shell name is prepended in arg[0] with a '-' because of the traditional login shell rules.

The supplementary groups are set by calling the system routine setgroups(). The real group and user id is established to that of the user. The child changes the current working directory to the user's home directory. The user must be able to access the directory or this will fail. Some site clean the permissions on the home directory when an account is disabled, which is one reason the chdir() is delayed until this point, if done as root it would succeed.

The log is closed.

The session id is returned to the parent mom by calling *starter_return()*. Finally, the shell is exec-ed.




```
int start_process(task *ptask, char **argv, char **envp)
```

Args:

ptaskthe task structure which has already been created for the session.

argv an array of arguments to pass to execve().

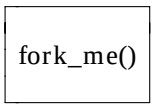
envp an array of environment strings, the last must be NULL.

Returns:

0 if no error occurred.

-1 on error.

Start a process for a spawn request. This will be different from a job's initial shell task in that the environment will be specified and no interactive code need be included.




```
pid_t fork_me(int connection)
```

Args:

connection
all network connections except *connection* are closed.

Returns:

pid of the newly forked child process.

Forks a new process. In the child, closes all network connections except the one specified (-1 means close all). The action for SIGCHLD is reset to {SIG_DFL}, otherwise the *system()* call used in various places will not function. The action for SIGHUP, SIGINT and SIGTERM are also reset to {SIG_DFL} and the signal mask is reset so no signals are blocked. The machine dependent function *mom_close_poll()* is called to close or clean up any files/items/... opened/created in *mom_open_poll()*.

nodes_free()

void nodes_free(job *pjob)

Args:

pjob pointer to the job structure of interest.

Free the ji_nodes array for a job. If any events are attached to an array element, free them as well.

job_nodes()

void job_nodes(job *pjob)

Args:

pjob pointer to the job structure of interest.

Generate a ji_nodes array for a job from the exec_host attribute. Call nodes_free() just in case we have seen this job before. Parse exec_host first to count the number of nodes and allocate an array of nodeent's. Then, parse it again to get the hostname of each node and init the other fields of each nodeent element. The final element will have the ne_node field set to TM_ERROR_NODE.

starter_return()

void starter_return(int up_pipe, int down_pipe, int code)

Args:

up_pipe
file descriptor of pipe to the parent MOM.

down_pipe
file descriptor of pipe from the parent MOM.

code which is written on the up pipe to MOM to indicate the job state.

The code is written to mom and the up pipe is closed. A read is made from the down pipe as a sync mechanism, then it is closed. If the code is less than zero, **exit()** is called.

std_file_name

```
char *std_file_name(job *pjob, enum job_file which)
```

Args:

pjob pointer to the job.

whichfile's name should be returned, values are: {StdOut}, {StdErr}, or {Chkpt}.

Returns:

The file name created for the indicated file. Note, the return points to a static area that will be overwritten on the next call.

If interactive jobs are supported ...
If the job is interactive, JOB_ATR_interactive set, the slave tty name has been stored in the output path attribute, JOB_ATR_outpath. That name is returned.

Otherwise, if the file is to be retained on the execution host as determined by attribute JOB_ATR_keep, the file name generated is the "default" name
`job_name.Xjob_sequence_number`
where X is either `o` or `e`. This file will be created in the user's home directory. The home directory path is maintained in the job structure.

If the file is not kept, then it is created in the PBS spool directory unless MOM is build with {NO_SPOOL_OUTPUT} defined, in which case it is created in the user's home directory. In either case, the name is the 11 character prefix obtained from ji_fileprefix appended with a suffix corresponding to which file is being created.

open_std_file()

```
int open_std_file(job *pjob, enum job_file which, int flag, gid_t gid)
```

Args:

pjob pointer to the job.

whichfile is to be opened, see *std_file_name()*.

flag for open, specifies create or truncate options as well as read/write mode.

gid which group should own the file.

Returns:

descriptor
for the open file, -1 if the open fails.

Calls *std_file_name()* to obtain the name and then opens the file.

bld_env_variables

```
void bld_env_variables(struct var_table *table, char *name, char *value)
```

Args:

table pointer to the var_table structure which controls the buffer and array of variables being built for the job.

nameof a variable to add.

valueof a variable to add.

An environment variable of the form, keyword=value, is added to the set to be place in the job's environment. The argument *name* may be either a name or the whole keyword=value string. If the *value* argument is a null pointer, then *name* is assumed to be the whole string. If *value* is not null, then a '=' is appended to name and the value appended to that. If there is no room in the control table or the buffer, nothing is added.

init_groups()

```
int init_groups(char *user, int pwgroup, int group_size, int *groups)
```

Args:

user name.

pwgroup
: primary user's group from password entry.

group_size
: size of the *groups* array, typically {NGROUPS_MAX}.

groups
: pointer to integer array of size *group_size*.

Returns:

The number of supplementary groups placed in *groups*, -1 on an error.

The primary group gid is placed in *groups*. The C library routine getgrent() is used to scan the group file to locate all groups in which the *user* is a member. The gid for those groups are added to the array *groups*. An error, -1, is returned if the number of groups exceeds the array size given by *group_size*.

catchinter()

```
static void catchinter()
```

This routine applies only to "interactive" jobs. This routine catches the death of child signal when either the reader child or the job grand-child of MOM dies. Remember, the direct child of MOM is not the job in this case, but is the writer() process.

When SIGCHLD is received, the other processes in the group are killed to make sure the job ends. The variable *mom_writer_go* is set to zero, see *mom_writer()*.

check_pwd()

```
struct passwd * check_pwd(job *pjob)
```

This routine obtains the password entry for the user specified by the server in the job attribute JOB_ATR_euser. This is the user name under which the job should be executed. The corresponding uid is save later in the job structure for future use. The execution group is handled likewise. If the execution group is not specified (it always will be) or the {ATTR_VFLAG_DFLT} bit is set indicating the normal login group, the primary group from the

password entry is used. The routine *init_groups()* is called to scan the group file and build a list of the supplementary groups of which the user is a member. This group list and the user's home directory is saved in an *grpcache* structure as an extension to the job structure.

The routine *site_mom_chkuser*() is called. This is a stub routine provided to allow a site the ability to customize checking of an accounts validity. The supplied version always returns false. If a site's modified version returns true, meaning the account is invalid, MOM will set the job state to {JOB_SUBSTATE_EXITING} and the exit status to {JOB_EXEC_FAIL} aborting the job.

mom_restart_job()

```
int mom_restart_job(job *pjob, char *path)
```

Args:

pjob pointer to job structure of job to restart.

path Name of path of directory containing restart files for tasks within the job.

Return:

The number of tasks restarted, -1 implies an error occurred.

The directory specified by *path* is read and for each entry the task id is taken from the entry name. The task must be part of the the job. The machine dependent routine *mach_restart*() is called with the task pointer and path it the specific restart file to restart the job as required on that type of machine.

If there are any errors, -1 is returned.

8.2.4.5. File: catch_child.c

The file *src/resmom/catch_child.c* contains machine independent functions dealing with the termination of a job. A name like on_job_termination would be better suited, but the name started with the first function placed in the file, oh well...

catch_child()

```
void catch_child()
```

This is the signal handler for SIGCHLD for the main mom. All it does is set termin_child to indicate some process died, maybe even a job task.

scan_for_exiting()

```
void scan_for_exiting()
```

This function is called from MOM's main loop when it finds the flag exiting_tasks set. As explained in the general narrative on MOM, resource usage by tasks must be collected before the child process is reaped and the process table entry is erased. This is done in scan_for_terminated(). If the a job is marked {MOM_CHKPT_ACTIVE}, it will be skipped since we do not want to change its state as tasks exit. If a job has {MOM_CHKPT_POST} set, a checkpoint attempt had an error and some tasks were aborted. In this case, the function *chkpt_partial()* is called. If

the death of child is from a task, then exiting_tasks is set. If the task was the original shell started by mother superior, {JOB_SUBSTATE_EXITING} is set if no other nodes are part of the job or none can be communicated with. If other nodes exist which can be communicated with, they are sent a message to terminate the job.

For a job now marked as substate {JOB_SUBSTATE_EXITING}, the following operations are performed:

1. Call *kill_job()* to kill off any processes in the task sessions that tried to escape from the job, i.e. were forked or placed into the background.
2. The job is unlinked from the resource usage polling list.
3. A connection is opened to the server which sent MOM the job. The connection set set to receive the reply from the server by calling *add_conn()* to set the read function to be *obit_reply()*.

4A. A child process is forked. The child runs the epilogue script, if one exists, by calling *run_pelog()* with {PE_EPILOGUE}. If the job is a "normal" batch job, run_pelog() is called with {PE_IO_TYPE_STD} so the epilogue output goes to the job's output file. But if the job is an interactive job, the pseudo terminal connection back to qsub has already been lost, so run_pelog() is called with {PE_IO_TYPE_NULL} so the epilogue output is sent to /dev/null.

A Job Obituary Notice is sent to that server, see *send_jobobit()* in The notice contains the exit status of the job including any of the special status discussed in start_exec.c. The notice also contains the most recent, hopefully last, accounting of the resources used by the job.

4B. In the parent, the real and true MOM, the job substate is set to {JOB_SUBSTATE_OBIT} indicating that the notice has been sent. Note, this state is not recorded on disk (save_job() is not called). If MOM crashes, we want her to resend the obit notice.

MOM will process up to two jobs in the exiting state before returning to the main loop. The two job limit is to keep from being out of touch with the network (mainly doing accept()s) for too long.

The addition of multiple tasks running on more than one MOM has made the state changes more difficult to understand. Just the RUNNING and EXITING states are considered in the following figure.

Job State

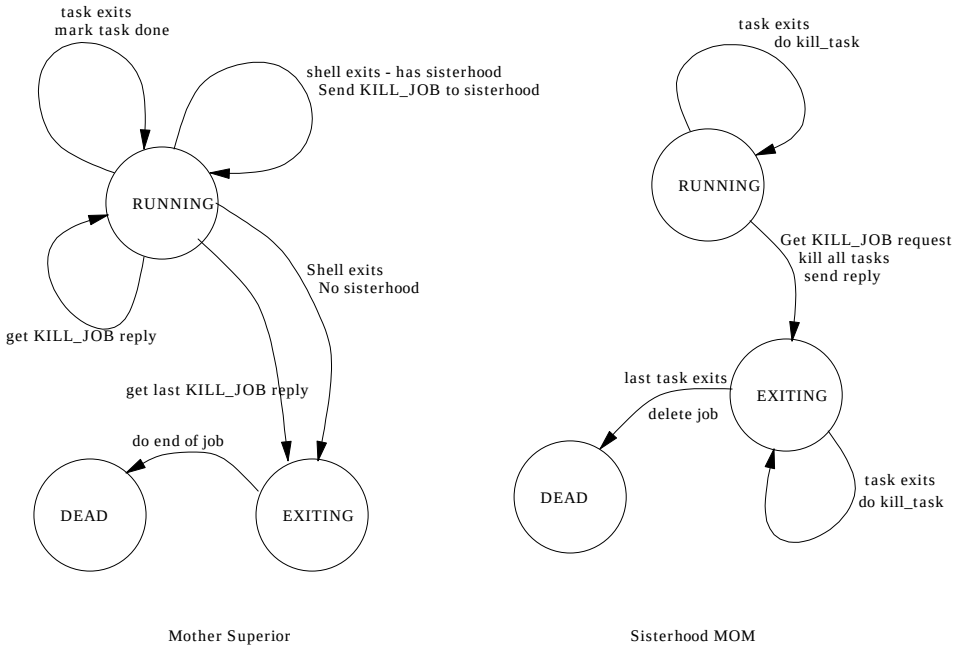


Figure 8 - 1

obit_reply()

```
static void obit_reply(socket)
```

Args:

socketdescriptor of the connection on which data has been received.

This function is entered when data is ready to read on a connection whose read function is this function. That is set up in *scan_for_exiting()* as described above. A request structure is allocated in which the reply is decoded by calling *isode_reply_read()*. If the reply does not decode or the connection has closed (the server timed it out - a bad network situation), the reply code is set to -1.

When scan_for_exiting() sent the obit notice to the server, it set the socket number in the job structure in ji_momhandle. So we now scan the jobs for the job with the substate of {JOB_SUBSTATE_OBIT} and the corresponding socket number recorded there.

The server may respond with one of four different codes:

PBSE_NONE

The job is placed into substate {JOB_SUBSTATE_EXITED} and the connection to the server is added to the set on which MOM will accept requests. The server will then direct disposition of the job's files, etc.

PBSE_ALRDYEXIT

The server is already performing exit processing for the job. MOM must have been restarted after sending a Job Obituary request and is now sending a second. The server will continue processing the job on the thread and connection started by the first request. MOM will update her copy of the job state and close this connection.

PBSE_CLEANEDOUT
: The server did not own the job and the most recent server restart was of type cold. Therefore, the job was discarded by the server and mom should do the same. Thus *mom_deljob()* is called. Note, this is the reason a pointer to the next job in the list has already be recorded. Otherwise, at the bottom of the loop there would be no next job.

-1
: The special connection is closed if EOF is read or the reply did not decode. The job substate is reset to JOB_SUBSTATE_EXITING so another Obit notice will be sent to the server.

Any other
: If the server returns any other response code, such as PBSE_NOJOBID, then the event is logged, and the job discarded by calling *mom_deljob()*. We hope this never happens.

Finally, the request structure is freed and the socket shutdown and closed.

chkpt_partial()

```
void chkpt_partial(job *pjob)
```

Args:

pjob pointer to job which had a checkpoint error

This routine is called to restart tasks for a job which had a checkpoint cause tasks to abort but was not able to finish a complete checkpoint of every task. It gets called from scan_for_exiting() when the work process spawned to do the checkpoint returns.

Loop through each task of the job. Each task that was checkpointed and has been reaped is restarted by calling mach_restart(). If all tasks for the job are running when we are done with the loop, turn off MOM_CHKPT_POST flag so job is back to where it was before the bad checkpoint attempt. Then get rid of incomplete checkpoint directory and move old chkpt dir back to regular if it exists. If any task restart fails, kill the job.

init_abort_jobs()

```
void init_abort_jobs(int mode)
```

Args:

modeof MOM's initialization.

This function is called from mom_main when MOM is first started. One of three conditions exists.

1. There are/were no jobs running on the host. We will not find any in MOM's job directory.
2. There are jobs found and the `-r` command option was set, the mode flag is non-zero. MOM is coming up after being killed or (heaven forbid) crashing. The jobs that were running are no longer the children of MOM, they now belong to the *init* process. MOM will not receive the death of child notice. Therefore MOM goes into a homicidal rage, reaping vengeance for being abandoned, and kills off all the tasks associated with jobs she had managed, by calling *kill_job()*.

   The session id for each task is cleared so that scan_for_exiting() will not issue another kill. If the {JOB_SVFLG_HERE} flag is not set for the job (i.e. local MOM is not mother superior), the job is thrown away. If the flag is on (i.e. we are mother superior), and a sister-

hood exists, a message is sent to all the other MOM's to kill the job. If we are mother superior and no sisterhood exists, the job exit status is set to one of three values depending on the job to indicate it was killed on recovery:

{JOB_EXEC_INITABT} − normal, non-checkpointed job.

{JOB_EXEC_INITRST} − job has a non-migratable (Cray style) checkpoint file, thus could restart.

{JOB_EXEC_INITRMG} − job has a migratable checkpoint file (not implemented).

The substate is set to {JOB_SUBSTATE_EXITING}, and the exiting_tasks flag is set to tell the main loop that jobs have "died".

3. There are jobs but the recovery mode is not set. This by convention indicates that the system was also down and there are not jobs still running. In this case we cannot go killing session as MOM might hit innocent bystanders, the session id may have been already re-used. All of the other procedures described in case 2 are followed.

mom_deljob()

```
void mom_deljob(job *pjob)
```

Args:

pjob pointer to job structure.

This is a system semi-dependent routine. On Unicos (Cray), *rmtmpdir()* is called to remove the temporary directories. Then *job_purge()* is called to completely remove any knowledge MOM has of the job.

8.2.4.6. File: mom_inter.c

The file *src/resmom/mom_inter.c* contains functions to support the execution of an interactive batch job. Mostly, these functions deal with the creation and setup of the pseudo tty used by the job for input and output and the routines to move data between the master tty and the socket connection to qsub, see figure 8−2.

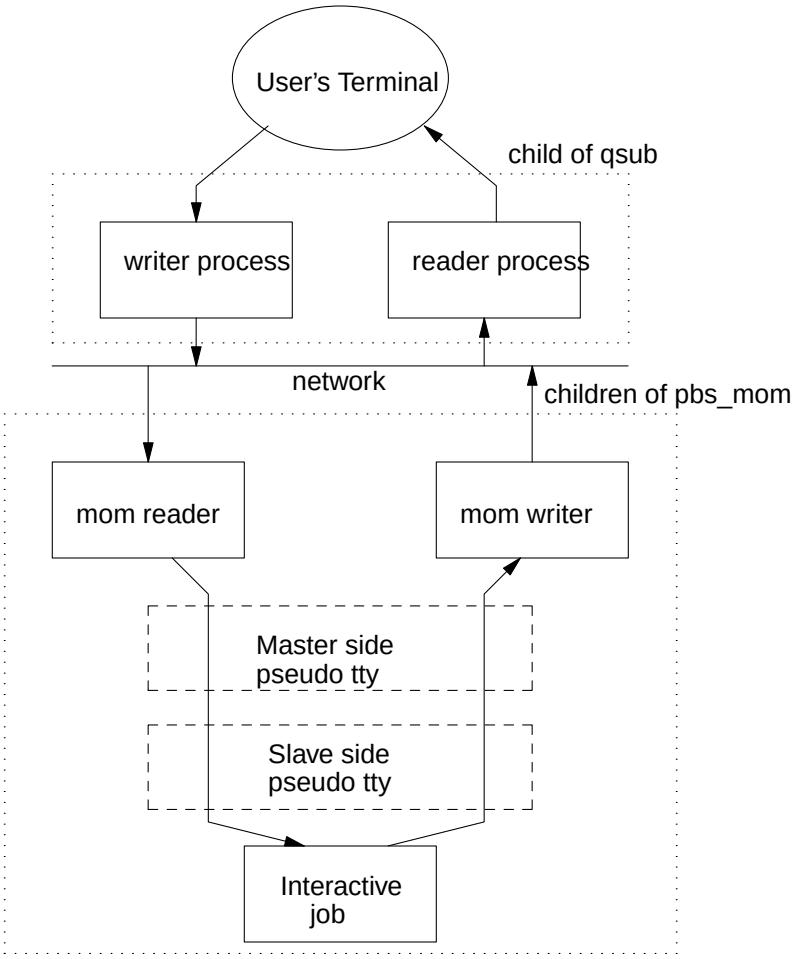


Figure 8−2: Interactive Job Communication Flow

read_net()

```
static int read_net(int socket, char *buffer, int amount)
```

Args:

socketThe socket connection back to qsub.

bufferPointer to a data buffer.

amount

The amount to of data to read from the network.

Returns:

Positive number which is amount of data read, or -1 on error.

This routines will read data from the network until the expected *amount* of data has been read.

rcvttype()

```
char *rcvttype(int socket)
```

Args:

socketThe socket connection back to qsub.

Returns:

The terminal type string.

This routine reads and validates the terminal type string and terminal control characters sent by qsub. The terminal type string must be of the form: TERM=type. The number of control characters expected from qsub is defined by {PBS_TERM_CCA}. See *set_termcc()*, below, for which characters are expected.

set_termcc()

```
void set_termcc(fds)
```

Args:

fds The file descriptor of the slave pseudo tty.

This routine makes the system call *tcsetattr()* with {TCSANOW} to set the control characters of the pseudo terminal to match those of qsub's controlling terminal. The characters set are VINTR, VQUIT, VERASE, VKILL, VEOF, and VSUSP.

rcvwinsize()

```
int rcvwinsize(int socket)
```

Args:

socketconnection back to qsub.

Returns:

Zero if successful, -1 on an error.

This function receives the window size as send by qsub. It is a string of the form: WINSIZE rn cn xn yn, where rn, cn, xn, and yn are numerical values specifying the row size, column size, and number of pixels in x and y. The received values will be used by *setwinsize()* to set the window size for the pseudo tty.

setwinsize()

```
int setwinsize(fds)
```

Args:

fds The file descriptor of the slave pseudo tty.

Returns:
Zero if successful, -1 on an error.

Sets the window size by calling *ioctl()* with {TIOCSWINSZ} and the size values obtained by *rcvwinsize()*.

mom_reader()

```
int mom_reader(int socket, int ptc)
```

Args:

socketconnection to qsub.

ptc The file descriptor of the master side pseudo tty.

Returns:
Zero on EOF, -1 on an error.

Data is read from the network and written to the master tty until either EOF is read or an error occurs. If either the read or the write are interrupted by a signal, the operation is re-tried. Minus 0ne (-1) is returned for any other error. Zero (0) is returned when EOF is reached.

mom_writer()

```
int mom_writer(int socket, int ptc)
Ce
```

Args:

socketconnection to qsub.

ptc The file descriptor of the master side pseudo tty.

Returns:
Zero on EOF, -1 on an error.

Data is read from the master tty and written to qsub over the network until EOF is read, an error occurs or the variable *mom_writer_go* is set to zero in *catchinter()* on death of the job or reader process. If either the read or the write are interrupted by a signal, the operation is re-tried. Minus 0ne (-1) is returned for any other error. Zero (0) is returned when EOF is reached.

conn_qsub()

```
int conn_qsub(char *host, int port)
```

Args:

host the name of the host on which qsub is running.

port the port on which qsub will accept a connection.

Returns:
The socket descriptor if the connection is made, or -1.

The host address is obtained from *get_hostaddr()*. *client_to_svr()* is called to establish the connection.

8.2.4.7. File: requests.c

The file *src/resmom/requests.c* contains various request processors for MOM. These are for requests which are handled completely differently than by the server, therefore MOM has her own separate code. The Queue Job sequence is very similar to the servers, thus that code, with a few #IFDEFs is used directly as described below.

fork_to_user()

```
static pid_t fork_to_user(struct batch_request *request)
```

Args:

request
pointer, must point to a `cpyfiles` request as used by either the Copy Files or Delete Files request from the server.

Returns:

pid of the new process.

The function *fork_me()* is called to fork a new process, a child of mine (MOM). For the child process, the user identification information is obtained from one of two places. If the request is for a job about which MOM knows, she is able to find the job structure for the provided job id, then she uses the execution user id and group id from the basic job structure and the home directory and supplementary groups from the *grpcache* structure extension to the job structure set up by *finish_exec()*. If the job is not known by MOM, true for a stage-in request, them MOM must go to the password entry and call *init_groups()* for the above information. The user name is obtained from the `cpyfile` request.

The current working directory is changed to that user's home and the pid from the fork() call is returned.

add_bad_list()

```
static void add_bad_list(char **plist, char *newentry, int newlines);
```

Args:

plist pointer to character pointer which may point to pre-allocated area or be null.

newentry
The new text entry to add to the list of bad file messages.

newlines
The number of new lines to prefix the new text.

If *plist* is null, no memory has yet been allocated for this message, malloc the amount needed to hold the new text. Otherwise, a prior message has been built, realloc the memory to hold it plus the new message.

Pre-append the required number of new-lines for formatting. Append the new text.

return_file()

```
static int return_file(job *pjob, enum job_file which, int socket)
```

Args:

pjob pointer to a job structure.

whichfile should be returned to the server.

socketconnection to the server.

Returns:

zero on successes.

non-zero
error number if an error occurred.

This function is used to return "standard" files of a job back to the server. Such files are the job's standard output and error and the job's checkpoint file. Typically, this function is called when a job is being rerun. The standard files must be returned to the server in order to have them available to send to a different MOM when the rerun actually occurs.

std_file_name() is called to obtain the file name associated with the job and the type (which). If the file can be opened, one or more Move Job File requests are generated and sent to the server. Each Move Job File request sends a chunk of the file, up to {RT_BLK_SZ} (4k) bytes. *send_jobfile()* is used to encode and send the request to the server. When all chunks have been sent the file is closed. Note the connection to the server is left open by this routine.

local_or_remote()

```
static int local_or_remote(char **path)
```

Args:

path (In and Out) pointer to string containing pathname

Return:

value1 if file is remote, 0 if local on this host.

path If the file is local, *path* is updated to point to the local path, without the `host:` prefix.

The argument *path* is a file name in the form `host:filepath`. If the `host` prefix matches with this machine's host name, 1 is returned and path is undisturbed. Otherwise 0 is returned and path is reset to point after the colon.

The `host` prefix will match if

1. It exactly matches the local host name or if it matches a leading substring of the local host name and the next character in the local host name is a dot ('.'). E.g. a host prefix of `x.y` will match a local host name of `x.y.z`
2. It is `localhost`.
3. The helper function *told_to_cp()* returns true saying the host and path match a $usecp entry in the config file. In this case, *path* is updated to point to the "local" substitute path.

told_to_cp()

```
static int told_to_cp(char  *host, char  *oldpath, char **newpath)
```

Args:

host is a host name from a file destination.

oldpath
is the path from a file destination.

newpath
(RETURN) is updated to the local path if a match is found.

Returns:
1 if match is found, 0 otherwise.

This routine checks an file destination against a $usecp entry in the config file. This entry tells Mom that a remote destination is also mounted locally and what the local path is. This allows the use of /bin/cp instead of rcp to deliver (or stage) the file.

If the *host* and *oldpath* from a destination supplied to *local_or_remote()* matches an $usecp config file entry, then *newpath* is updated to point to alternate patch supplied on the config file entry. As called from local_or_remote(), *host* is the host portion of the output (or staged file) path, *oldpath* is the path portion of that same entry and *newpath* original points to the whole entry. Newpath is changed if host and oldpath match up with an entry.

is_file_same()

```
static int is_file_same(char *file1, char *file2)
```

Args:

file1 file2
name of two files, presumed local to this host.

Returns:
1 if the two names point to the same file (inode), 0 if not.

Using *stat()*, if the two file names point to the same device and same inode, then they are the same file and one (1) is returned. Otherwise zero (0) is returned.

req_deletejob()

```
void req_deletejob(struct batch_request *request)
```

Args:

request
pointer to the Delete Job Request received from the Server.

The job is located by calling *find_job()* with the job id from the request and purged by calling *mom_deljob()*. For Unicos, any temporary directories are removed by calling *rmtmpdir*().

req_holdjob()

```
void req_holdjob(struct batch_request *request)
```

Args:

request
pointer to the Hold Job Request received from the Server.

The job is located by calling *find_job()*. The routine *start_checkpoint()* is called to checkpoint the job if that is supported. If checkpoint is not supported, start_checkpoint() will return [PBSE_NOSUP].

message_job()

```
int message_job(job *pjob, enum job_file jft, char *text)
```

Args:

pjob pointer to job.

jft indicates which file: StdOut or StdErr.

text to write on the file.

Returns:

A PBS error number or zero.

This routine is used to write a message onto either the standard output or standard error file of the job. The file is opened by calling *open_std_file()*. If the last character of the supplied text is not a new-line character, one is appended.

req_messagejob()

```
void req_messagejob(struct batch_request *request)
```

Args:

request
pointer to the Message Job Request received from the Server.

A flag is set according to which file the message is to be written, the default is the job's standard output. The job structure is located by calling *find_job()*. The message from the request is passed to *message_job()*.

req_modifyjob()

```
void req_modifyjob(struct batch_request *request)
```

Args:

request
pointer to the Modify Job Request received from the Server.

attr_atomic_set() is called to decode the resource limits (and attributes). Each is update in the job structure. If a resource list (limit) item is changed, *mom_set_limits()* is called with the mode parameter of {SET_LIMIT_ALTER} to update the limits. Any errors are returned to the server.

req_shutdown()

```
void req_shutdown(struct batch_request *request)
```

Args:

request
pointer to the Shutdown Request received from the Server.

Currently does nothing from a lack of time and understand of how to do it.

req_signaljob()

```
void req_signaljob(struct batch_request *request)
```

Args:

request
pointer to the Signal Job Request received from the Server.

This function forwards a signal from the server to the running job. The signal is an numeric or alphanumeric string with or without the prefix "SIG".

Two names are treated special under Unicos. If {SIG_SUSPEND} (suspend) is received, a running job is to be suspended via the system call suspend(2). This is done within the Cray specific routine *cray_susp_resum()* which is called with with the argument *which* set to 1 to indicate a suspend should be performed. If {SIG_RESUME} (resume) is received, a suspended job is be resumed via the system call resume(2). This is also done by *cray_susp_resum()* where *which* is set to 0 to indicate a resume.

Otherwise, if the first character is numeric, the string is converted into a number. If the signal is a string, it is taken to be signal name, and the SIG prefix, if present, is stripped. The name is converted into a numeric value via a table, *sig_tbl* located in *mom_start.c*. The table is system dependent in order to correctly map the varying signal names. The signal value is issued to the job by calling the routine *kill_job()*.

If the signal being sent to the job is {SIGKILL} and the job is not in substate {JOB_SUBSTATE_RUNNING}, we have troubles as the server would not have passed on the signal job request unless it thought the job was running. So if MOM believes the job is not running, we mark it as {JOB_SUBSTATE_EXITING} and set *exiting_tasks* to cause a (new) Job Obit notice to be sent to the scheduler. This was added in response to bug #779, a job has no live processes but the server thought it was still running. Sending a signal (via qdel) did nothing because no process would die to generate a SIGCHLD to mom to cause her to (re)issue the obit notice. This fix (spelled "kludge") will cause the obit notice.

Another "kludge" is the SIGNUL and no processes found check. If *kill_job()* returns zero, then no processes were found that were part of the job, hence the job should have exited. We use SIGNUL because of timing between the server and mom − the server may well send SIGKILL after a prior SIGTERM because it didn't receive the Obit notice in time, but the job may in fact have exited. Here there would not be any processes alive and we do not wish to

trigger the recycle or log the nominal case.

req_stat_job()

```
void req_stat_job(struct batch_request *request)
```

Args:

request
pointer to the Status Job Request received from the Server.

If a null job id was passed in the request, the request is for status of all jobs, otherwise it is for the one identified job. For each job for which status is to be returned to the server, *mom_set_use()* is called to update the latest resource usage figures. Attributes are returned to the server only if they are modified, {ATR_VFLAG_MODIFY} is set. This is done to keep down traffic and make sure Mom doesn't update any attribute she shouldn't. The attributes returned are: JOB_ATR_resc_used, JOB_ATR_errpath, JOB_ATR_outpath and JOB_ATR_session_id. The resources used is calculated with a special case for "cput" and "mem". These are added with the polled information from any sisterhood to give a total for all nodes in the job.

del_files()

```
static int del_files(struct batch_request *request)
```

Args:

request
pointer to the Copy/Delete File Request received from the Server.

This is a local support function. Before being called the following two things must be true: (1) the external variables useruid, usergid, ngroup (the number of supplementary groups), and groups (the supplementary group array) must be set and the current working directory must be the the owner's home directory. Both of these are accomplished by first calling *fork_to_user()*.

Each file contained the basic Delete (Copy) Files Request is deleted from MOM's spool directory (or the user's home directory) by unlinking it if the file is not a standard (output or error) file and if the remote and local names in the request point to different files. If the remote file is actually local as determined by *local_or_remote()*, and if *is_file_same()* indicates the two names point to the same file, then the file is not deleted as it was here before the job started. We only deleted files staged in or staged out to a different file. For example, if the user said to stage in a file from foo:/tmp/bar to /tmp/bar and if the job ran on host foo, the file should not be deleted.

If the file is marked as being a *standard job file*, meaning output, error, or checkpoint, the unlink is done as root. These files must be listed first by the server, once the process changes to act with the user's level of privilege, it cannot go back. If not marked as a standard job file, the file which is likely one spooled in. It is unlinked as the user. This prevents the removal of files owned by a different user.

req_rerunjob()

```
void req_rerunjob(struct batch_request *request)
```

Args:

request
pointer to the Rerun Job Request received from the Server.

This request is sent to MOM by the server to tell MOM the job is being rerun and the so called *standard files*
(output, error, and checkpoint) should be returned to the server for safe keeping. This happens after the server has killed the job by sending a request for a SIGKILL signal. As the file return will require multiple requests back to the server, *fork_me()* is called to create a child process. The password entry is found for the uid under which the job was executed. A new connection is opened by the child to the server which owns the job, *client_to_svr()*. The local function *return_file()* is called three times, once each for output, error, and checkpoint.

req_cpyfile()

```
void req_cpyfile(struct batch_request *request)
```

Args:

request
pointer to the Copy Files Request received from the Server.

The Copy Files request is sent by the server to MOM after the job has terminated. It directs MOM to deliver the files to their destinations. It may also be sent to stage-in a file before the job is sent to MOM. The routine *fork_to_user()* is called to fork a child and setup: (1) the external variables useruid, usergid, ngroup (the number of supplementary groups), and groups (the supplementary group array) and change the current working directory to the the owner's home directory. The child process sets its real and effective uid and gid and supplementary groups to that of the user. Then for each file listed in the request, the destination is parsed.

If the destination host name, the part before the colon, see *local_or_remote()*, is the same as which MOM is running, the child sets up to do a local copy using the `/bin/cp` command. For a different host, the child will set up to do a remote copy using the `pbs_rcp` command. The copy (cp or rcp) command is built. If the file does not exist, no attempt to copy is made and no error is returned.

If the file is local and if the source and destination is the same file, *is_file_same()*, then the copy operation is skipped. Otherwise, the *sys_copy()* function, see below, is used to issue the copy command. If the return is zero, it is assumed the copy was successful. Then and only if the file was being copied out bound and from MOM's spool directory is that file deleted.

If the sys_copy call returns an error, then we assume the copy failed. If the copy was to stage in files, any prior file in the request, that was successfully copied, is deleted to prevent unused files from being left lying around as the job will not be run. If MOM was copying a file outward from her spool area at the time of the failure, then that file is relinked (moved) into a *undelivered* directory. It is up to the administrator to deal with any files in that directory. Any copy failure results in a log message with an *event class* of "File". The information is relayed back to the server so that the server can send mail to the user.

sys_copy()

```
static int sys_copy(char *ag0, char *ag1, char *ag2, conn)
```

Args:

ag0 Argument zero of the copy, the copy command. It must be a full path name.

ag1 The source file name, it should be full qualified. If a remote file, it should be of the form: user@host:/full/path/name

ag2 The destination file name, it should be full qualified. If a remote file, it should be of the form: user@host:/full/path/name

Returns:

The result code:

0 Successful copy.

13 Exec() of copy program failed.

10xxx Fork() failed. xxx is the system error number.

20xxx Error on wait(), xxx is the system error number.

30xxx The copy process was stopped, xxx is the stop signal.

40xxx The copy process was killed with a signal, xxx is the signal.

Sys_copy will attempt to fork() and exec() the copy program up to 4 times with a 15 second delay between each try. Any failures are logged and if all four attempts fail, the error value described above is returned.

req_delfile()

```
void req_delfile(struct batch_request *request)
```

Args:

request
pointer to the Delete File Request received from the Server.

This request is sent to MOM by the server to tell MOM to delete job related files, see *on_job_end()* and *on_job_rerun()* in server/req_jobobit.c.

The request uses the same structure as the Copy File Request. Only the local file name is used. As with that request, the first thing is to call *fork_to_user()* to directory. However, at this level, the child remains for the time being it root privileges. For each file in the list, *del_files()* is called to delete the file.

start_checkpoint()

```
int start_checkpoint(job *pjob, int abort, struct batch_request *preq);
```

Args:

pjob pointer to the job.

abort If non-zero, the checkpoint call is to abort the running processes.

preq If non-null, points to the request from the server; if null, this is an internal call.

Returns:

Zero if checkpointing is supported and succeeded, [PBSE_NOSUP] if checkpoint is non supported, and other non-zero error returns for errors.

On the Cray, the checkpoint call may wait for two minutes to start, if the processes are swapped out, the actual checkpoint call must be done by a child process to keep from locking up MOM for that time.

This function first calls a machine dependent routine, *mom_does_chkpnt*(), to determine if checkpointing is supported. If it is, a child process is forked to call the routine *mom_checkpoint_job*() to do the checkpoint. If the checkpoint is being performed at the request of the server, preq points the the request; the child process will reply based on the return of mom_checkpoint_job(). The parent (original MOM) will set the function post_chkpt() to be called when the child is done with the checkpoint. Also, if *abort* is set, the flag *MOM_CHKPT_ACTIVE* is turned on so dying tasks don't cause obit messages to be sent.

mom_checkpoint_job()

```
int mom_checkpoint_job(job *pjob, int abort)
```

Args:

pjob pointer to the job

abortkill the job if TRUE.

Form the pathname for the checkpoint directory of the job. If one already exists, rename it with the postfix ".old". Create a new checkpoint directory. On the Cray, check to see if the job is suspended and if *abort* is set. If so, resume the job first so job will be "Q"ueued and then back into "R"unning when restarted. For each task in the job, call the machine dependent routine **mach_checkpoint** to checkpoint each task. If any checkpoints fail and *abort* is set, return the error *PBSE_CKPSHORT*. If any checkpoints fail but *abort* is not set, just remove the checkpoint directory and, if there is an old directory, rename it to the original name.

post_chkpt()

```
void post_chkpt(job *pjob, int ev)
```

Args:

pjob pointer to the job

ev error value

This function is called from scan_for_terminated() when found in ji_mompost to clean up after a checkpoint. We save the value of the flag *MOM_CHKPT_ACTIVE* so we can tell if the checkpoint was being done with abort. The flag *MOM_CHKPT_ACTIVE* is turned off and the value of *ev* is checked to see if there was an error. If no error occurred, turn on the flag *JOB_SVFLG_CHKPT* and return. If an error took place, but the checkpoint was not done with abort, just return. Otherwise, turn on the flag *MOM_CHKPT_POST* and loop through the job's checkpoint directory looking for checkpoint images for tasks. For each checkpoint images found, look for the corresponding task structure and set the task flag *TI_FLAGS_CHKPT*. Then set exiting_tasks so we call scan_for_exiting().

cray_susp_resum()

```
static void cray_susp_resum(job *pjob, int which, struct batch_request *request)
```

Args:

pjob pointer to the job

which1 for suspend, 0 for resume.

request
Pointer to the signal job request.

Under Unicos, the system functions suspend() and resume() can take a while, up to 120 seconds, MOM cannot afford to sit and wait. Therefore, the functions are performed by a child process.

MOM, the parent however needs to know if the operation succeeded or failed in order to update the job structure. When MOM forks, the parent records in the job structure the pid of the child process, *ji_momsubt* (for subtask), and a pointer to a post processing function, *ji_mompost*, which is called when the child process exits.

For suspend, the post processing function is *post_suspend()* located in unicos8 or unicosmk2 mom_start.c. MOM also notes the current time in *ji_momstat* in the job structure. This is needed to adjust the walltime used when the job is resumed.

The child job performs the suspend() or resume() system call and then acknowledges or rejects the original request from the server. The system call may be retried up to 3 times if it returns EAGAIN or EINTR. If the system call does not return an error, the child exits with zero; it exits with 1 if there was an error. See scan_for_terminated() in unicos8 or unicosmk2 mom_start.c.

8.2.4.8. File: prolog.c

The file *src/resmom/prolog.c* continues the various functions to support administrator supplied prologue and epilogue scripts. These scripts are run with root privilege before and after the user's job.

The prologue script arguments (argv) are:

argv[1]
The job ID.

argv[2]
The user's name.

argv[3]
The user's group name.

The epilogue arguments are the above plus:

argv[4]
The Job Name.

argv[5]
The Session ID.

argv[6]
The list of requested resource limits, attribute Resource_List.

argv[7]
The list of resources used, attribute resources_used.

argv[8]
The name of the queue in which the job resides.

argv[9]
The account sting (qsub -A option) if it is set.

The input file to the script is architecture dependent, see pelog_input(). The scripts standard output and standard error are connected to the files which are the output and error of the job. One exception being when the job is interactive, the output and error are closed before the epilogue is run, hence the epilogue is connected to /dev/null for output and error.

pelog_err()

```
static int pelog_err(char *file, int error, char *text)
```

Args:

file name of prologue/epilogue script.

error number to record in log.

text message to record in log.

Returns:
The error number is returned.

This function records a error number and text message in MOM's log when the prologue or epilogue fails.

pelogalm()

```
static void pelogalm()
```

This function is the SIGALRM handler for prolog.c. When the alarm set around the prologue or epilogue script times out before the script completes, this function is called. It kills the child running the script and sets the script exit to -4.

run_pelog()

```
int run_pelog(int which, char *file, job *pjob, int type)
```

Args:

which script to run, {IP_PROLOGUE} or {IP_EPILOGUE}.

file name of script to execute.

pjob pointer to job structure.

type of operation to connect to output/error.

Returns:
The exit status of the script.

This is the heart of the prologue/epilogue processing. If the script file does not exist, there is no action performed and it is not considered an error. Before the script (which may be an executable binary) is executed, the following checks are made to insure that it is "safe" to execute the script:

- The file must be owned by root.
- The file must be a regular file.
- The file must be readable and executable by root (the owner).
- The file must not be writable to any one other than root.

The system dependent input file is opened by calling *pe_input()*. If an error occurs, it is logged, *pelog_err()*, and the error returned. A child is forked, inheriting the null environment from MOM. The parent process sets an alarm to prevent the child from taking forever. The parent then waits for the child to complete. When it does, the exit status is returned.

If the output operation type is {PE_IO_TYPE_NULL}, /dev/null is opened for both standard output and and standard error. This is done when running the epilogue for an interactive job because the pseudo terminal has already be lost. If the output operation type is {PE_IO_TYPE_STD}, the standard output and error files of the job are opened and passed to the script. This is the case for the epilogue for normal jobs. If the output operation type is {PE_IO_TYPE_ASIS}, we go with the current file descriptors for 1 and 2. When called to run the prologue, the caller, *finish_exec()* is already attached to the standard output and error of the job.

8.2.4.9. File: req_quejob.c

MOM borrows the receive job functions req_quejob(), req_jobcredential(), req_jobscript(), req_rdytocommit(), and req_commit() from the server. There are some differences created by "#ifdef PBS_MOM" that should be pointed out. Additionally, MOM has her own version of req_mvjobfile().

req_quejob()

MOM requires that the request be from another daemon, the server. Also MOM does not worry about "queues".

If MOM finds the job being sent to her already exists, she sees if the existing version is marked as a checkpointed job, set in *ji_svrflags*. If so, she keeps the existing version, but marks it as state {JOB_SUBSTATE_TRANSICM} for req_commit(). The server should not be sending a script or the "ready to commit" requests.

For new jobs, MOM insist that the job owner attribute, JOB_ATR_job_owner be set by the server.

When decoding the job attributes, any error is fatal to the request. Also, if the *al_op* field in the received svrattrl structure is DFLT rather than SET, then the attribute being passed (likely a resource_list entry) contains a value set by the server rather than the user based on either a queue or server resource_default attribute (default value). Under Unicos, the default value may be overridden by the limit set in the user's User Data Base (UDB) entry. This check of DFLT is thus required. If DFLT, then {ATR_VFLAG_DEFLT} is set in the attribute (resource) structure at_flags member. This flag will be checked in *mom_set_limits()*, see src/resmom/unicos8/mom_mach.c, when limits are being actually set.

req_jobcredential()

The sender must be a server.

req_jobscript()

The sender must be a server.

req_mvjobfile()

This is MOM's own version. The files are owned by the user and placed in either the spool area or the user's home directory depending on the compile option, see *std_file_name*().

req_rdytocommit

The sender must be a server.

req_commit()

The job is linked into the all job list, marked in state {JOB_STATE_RUNNING} and substate {JOB_SUBSTATE_RUNNING}, and the server's network address is saved in *ji_momt.ji_svraddr*. Then *start_exec()* is called to place the job into execution. On return, the job information is saved with a call to *job_save()*. Then the attributes JOB_ATR_errpath, JOB_ATR_outpath, and JOB_ATR_session_id are marked as modified so their values will be returned (once) to the server in *status_attrib()*, see stat_job.c.

8.2.4.10. File: mom_comm.c

The file *src/resmom/mom_comm.c* groups together functions that deal with communication between MOM and tasks requesting Task Management functions, and communication between MOM's within a job acting as a sisterhood of nodes. Some miscellaneous functions are here for convenience.

save_task()

```
int save_task(ptask)
```

Args:

ptaskA pointer to the task structure representing the target task.

Return:

0 if no errors take place.

-1 if an error occurs.

This function is used to save the critical information associated with a task to disk.

event_alloc()

```
eventent *event_alloc(int com, nodeent *pnode, tm_event_t event, tm_task_id taskid)
```

Args:

com the command associated with the event.

pnodean entry in the nodeent array of the job to which the event belongs.

eventthe event number given by the requesting task or TM_NULL_EVENT if it is an internally generated event.

taskidthe task id of the requesting task.

Return:

pointer to malloc'ed eventent structure

This function will allocate an event and link it to the given nodeent entry.

task_create()

```
task *task_create(job *pjob, tm_task_id taskid)
```

Args:

pjob a pointer to the job structure which the new task will join.

taskidthe task id of the new task.

Return:

pointer to malloc'ed task structure

This function will allocate a task and link it to the given job. If a limit for the number of tasks allowed to be created on a single node exists for the job (taskspn), a NULL is returned if the new task would go over the limit.

task_recov()

```
int task_recov(job *pjob)
```

Args:

pjob a pointer to the job structure which is to have its tasks read from disk.

Return:

0 if no error occurs.

-1 on error.

Recover (read in) the tasks from their save files for a job. This function is only needed upon MOM start up.

tm_reply()

```
int tm_reply(int stream, int com, tm_event_t event)
```

Args:

stream
the TCP stream to communicate with the user task.

com the command to send.

eventthe event number for the message.

Return:

a DIS library error value

Send a reply message to a user proc over a TCP stream. The message will have the protocol type (TM_PROTOCOL), followed by the version (TM_PROTOCOL_VER), the command number then the event.

im_compose()

```
int im_compose(int stream, char *jobid, char *cookie, int com, tm_event_t event, tm_task_i
```

Args:

stream
the RPP stream to another MOM.

jobid the job id of the job this message concerns.

cookiethe job cookie of the job.

com the command of the message.

eventthe event of the message.

taskidthe task which this message concerns.

Return:

a DIS library error value

Send a reply message to another MOM over an RPP stream. The message will have the protocol type (IM_PROTOCOL), followed by the version (IM_PROTOCOL_VER), the job id, cookie, command, event and then task id.

send_sisters()

```
int send_sisters(job *pjob, int com)
```

Return:

Return:
count of messages sent.

Send a message (command = com) to all the other MOMs in the job pjob.

find_node()

```
nodeent *find_node(job *pjob, int stream, tm_node_id nodeid)
```

Check to see which node a stream is coming from. Return a NULL if it is not assigned to this job. Return a nodeent pointer if it is.

job_start_error()

```
void job_start_error(job *pjob, int code)
```

An error has been encountered starting a job. Format a message to all the sisterhood to get rid of their copy of the job. There should be no processes running at this point.

stream_eof()

```
void stream_eof(int stream, int ret)
```

Args:

stream
: an RPP stream that needs to be closed due to an error.

ret
: the DIS error which caused the problem.

Enter a loop to search though all the jobs looking for **stream**. We want to find if any events are being waited for from the "dead" stream and do something with them. If the stream is not found, just return. If it is found, enter a loop for the events being waited for. For each event, check the command and execute code to process an error for that type of request. If the command is *IM_JOIN_JOB*, call *send_sisters* to send an *IM_ABORT_JOB* to all the other MOM's to get rid of their copy of the job. Then mark the job with JOB_EXEC_RETRY. There should be no processes running at this point. If the command is *IM_ABORT_JOB* or *IM_KILL_JOB*, the job is already in the process of being killed but somebody has dropped off the face of the earth. Just check to see if everybody has been heard from in some form or another and set JOB_SUBSTATE_EXITING if so. If the command is a user request (such as IM_SPAWN_TASK), just inform the requesting process. If the command is *IM_POLL_JOB*, mark the job to die. If the stream turns out to come from Mother Superior, we are an orphan and just kill the job.

im_request()

```
void im_request(int stream, int version)
```

Args:

stream
: an RPP stream that has a message to read.

version
: the protocol version read by **do_rpp()** in mom_main.c.

Check that the **version** of the protocol is one we understand. Make sure the address of the incoming stream is from a host that is in our cluster. Read the jobid, cookie, command, event

and task and verify that they are meaningful. A large switch statement is entered with code for each type of command.

IM_JOIN_JOB
: Make sure it is Mother Superior calling. Then read the node id to be assigned to me, the number of nodes in the job and the node id for each node. The job attributes follow and are read by calling *decode_DIS_svrattrl()*. Send a IM_ALL_OKAY message back.

Anything other than IM_JOIN_JOB should be a request for a job we know about so call *find_job* and send an error if we come up empty. Make sure the cookie checks out. If the message is a reply to a request we sent (IM_ALL_OKAY or IM_ERROR), look for the event that corresponds to the message.

IM_KILL_JOB
: Sender is mom superior commanding me to kill a job which I should be a part of. Send a signal and set the jobstate to begin the kill. We wait for all tasks to exit before sending an obit to mother superior.

IM_SPAWN_TASK
: Read the parent node id and the task id for the new task. Next, read strings until a zero length string. These are the argv array for the exec. Finally, read strings until end of message. These are the environment variables. Call *task_create* then send a IM_ALL_OKAY message back.

IM_GET_TASKS
: Sender is MOM which controls a task that wants to get the list of tasks running here. Read the node id of the sending node and call *find_node()* to verify it is okay. Send a reply with the task id of each task running on the local node.

IM_SIGNAL_TASK
: Sender is MOM sending a task and signal to deliver. Read the node id of the sending node, the task id of the task to signal and the signal number to deliver. Call *kill_task* and send a reply back.

IM_OBIT_TASK
: Sender is MOM sending a request to monitor a task for exit. Read the node id of the sending node and the task id of the task to monitor. Check to make sure the task is local. If it has already exited, send a reply with the exit status. If it is still running, generate an obitent structure and link it to the task.

IM_POLL_JOB
: ender is (must be) mom superior commanding me to send information for a job which I should be a part of. Reply with a flag which gives a "recommendation" as to whether the job should be killed or not, followed by the cpu time and memory usage of the tasks on the local node. Sender is (must be) mom superior commanding me to abort a JOIN_JOB request. Make sure it is Mother Superior calling, then call *job_purge()*. This request is only sent to Mother Superior from a sub-mom to get a task id. Reply with a new task id for the job.

If The message received is a reply to one we sent, the event which is being completed will have a command number. Another switch statement will be entered for a reply of either IM_ALL_OKAY or IM_ERROR. A summary for IM_ALL_OKAY follows.

IM_JOIN_JOB
: I'm mother superior and the sender is one of the sisterhood saying she got the job structure sent and she accepts it. Check to see if any other sisters still need to reply. If not, call *finish_exec* to get the job going.

IM_KILL_JOB
: Sender is sending a response that a job which needs to die has been given the ax. Read the summed cpu time and memory usage of the tasks on the node responding. If no nodes have a KILL_JOB request outstanding, set JOB_SUBSTATE_EXITING.

IM_SPAWN_TASK
: Sender is MOM responding to a spawn request. Read the task id of the new task and compose a message to the requesting task with *tm_reply*.

IM_GET_TASKS
: Sender is MOM giving a list of tasks which she has started for this job. Send a reply to the requesting task, reading task id's from the remote MOM and writing them to the task.

IM_SIGNAL_TASK
: Sender is MOM with a good signal to report. Just send a TM_OKAY reply to the requesting task.

IM_OBIT_TASK
: Sender is MOM with a death report. Read the exit value for the task and compose a reply to the requesting task.

IM_POLL_JOB:
: I must be Mother Superior for the job and this is a reply with job resources to tally up. Read the recommendation to kill or not kill the job, the cpu time and memory sums for the sending node. If the recommendation is true, mark the job to be killed.

IM_GET_TID
: Sender must be Mother Superior with a TID. We should have a saved SPAWN request which corresponds to this TID request. Check to see if the SPAWN request needs to be forwarded to another MOM. If so, call *im_compose* with the new task id. If the SPAWN is local, call *task_create* to launch the new task, then reply to the requesting task with the SPAWN result.

The second type of reply which can come back from another MOM from the sisterhood is IM_ERROR. The type of request which is being replied to determines what is to be done with the error.

IM_JOIN_JOB
: A MOM has rejected a request to join a job. We need to send a ABORT_JOB to all the sisterhood and fail the job start to server. I must be mother superior. Call *job_start_error* with the error code sent with the reply.

IM_ABORT_JOB

IM_KILL_JOB
: Both these requests indicate job cleanup failed on a sister. Wait for everybody to respond then finish up. I must be mother superior.

IM_SPAWN_TASK

IM_GET_TASKS

IM_SIGNAL_TASK

IM_OBIT_TASK
: These are all requests which originate with a task. Find the task which needs to be informed of the error and call *tm_reply* to send it.

IM_POLL_JOB
: I must be Mother Superior for the job and this is an error reply to a poll request. The job needs to die so mark it to be killed.

IM_GET_TID
: Sender must be Mother Superior failing to send a TID. Send a fail to the task which called SPAWN.

tm_request()

```
int tm_request(int fd, int version)
```

Args:

fd a file descriptor to read.

version
the protocol version being sent.

Return:

-1 on error.

1 if no more data is available.

0 if more data is available.

Check that the source machine is localhost. If reading the jobid, cookie, command, event and task all work and make sense, the command is checked to see if it is TM_INIT. if so, a reply is generated and sent and the function returns. If the command is not TM_INIT, the node number where the requested action will take place is read. If the node number is part of the job, a large switch statement is entered with code for each type of command. If the action node is not the local host, a message to the remote action node will be composed and sent and an event attached to that node's element in the job's node array.

8.2.4.11. File: mom_server.c

The file *src/resmom/mom_server.c* groups together functions that deal with communication between a server and MOM's composing a cluster. This only includes a few message types, but will become more complex as the scalability of the code improves. Right now, no attempt is made to deal with scale issues.

is_compose()

```
int im_compose(int stream, int command)
```

Args:

stream
the RPP stream to a server.

command
the command of the message.

Return:

a DIS library error value

Send a reply message to a server over an RPP stream. The message will have the protocol type (IS_PROTOCOL), followed by the version (IS_PROTOCOL_VER), and the command.

is_request()

```
void is_request(int stream, int version)
```

Args:

stream
an RPP stream that has a message to read.

version
the protocol version read by **do_rpp()** in mom_main.c.

Check that the **version** of the protocol is one we understand. Make sure the address of the incoming stream is from the server. The commands that are recognized follow:

IS_NULL
This is used to send a "ping" from the server to MOM's that are not active. No response is needed.

IS_HELLO
The server wants us to send a IS_HELLO packet. This is used by the server to contact a MOM that was already up when the server came up. In this case, the server needs to initiate communication with any MOM that has not been heard from. MOM will send an IS_HELLO message back.

IS_CLUSTER_ADDRS
This is a response to a IS_HELLO message. It contains a list of IP addresses of the machines in the cluster. They get added to the okclients binary tree. Since IP addresses do not get deleted from okclients, there is a problem if a server is brought down and comes up again with a different node list. If any MOMs stay up through this process, they will get the new list added to the old.

8.2.5. Machine-dependent Files

Within the directory, *src/resmom* , there is one subdirectory for machine dependent code for each class of machine on which MOM runs. The basic structure of each machine dependent code is identical. Variations exist between systems as to how to accomplish the required function. The following sections will describe in machine independent terms what function each common module performs. Later sections will address the machine dependent methods used where there is significant difference from the "common model."

8.2.5.1. File: mom_mach.h

The file *src/resmom/<machine>/mom_mach.h* contains the machine-dependent macro definitions which are unique to MOM. It also contains the function prototypes for the equivalent machine-dependent functions.

8.2.5.2. File: mom_mach.c

The file *src/resmom/<machine>/mom_mach.c* contains the machine-dependent source code which is unique to MOM and generally relates to setting resource usage limits or determining resource usage by a job.

mom_set_limits()

```
int mom_set_limits(job *pjob, set_mode)
```

Args:

pjob A pointer to the job structure representing the target job.

set_mode
specifies if this call is for the initial setting of limits or for altering existing limits.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

This function recognizes all resources controlled on the machine, tests their values for sanity.

If *set_mode* is {SET_LIMIT_SET}, then it also sets the limits for the designated job to the limits specified by the job's resources. In this case, the function assumes that it is called from the child process before execing the job's shell. Additionally, it assumes that it is running as root and has access to the job's standard error file handle.

If *set_mode* is {SET_LIMIT_ALTER}, this function is being called to test and for some systems (the Cray), alter the limits. Systems which use the bsd based setrlimits() call cannot alter kernel enforced limits because the setrlimits call assumes the limits are being set for the current process. In the alter case, the main MOM cannot alter the job limits, only check them. The Cray's limits() call does allow another session's limits to be set.

In case of an error, in addition to returning a code defined in pbs_errno.h, the function puts an error message on standard error.

For implementation of this function on machine X for all values of X except unicos8 or unicosmk2, the method is to validate the limit value for all supplied resource limits, including default values, and set that limit if valid. If a limit is not supplied, the limit is set to unlimited.

For the different types of Unicos, there is also the User Data Base (UDB) declare limits with which to be concerned, thus things a done a bit differently. A private function which_limit() chooses the real limit based on:

1. If a limit is specified by the user, not a server default, and if less than the UDB limit, the user supplied limit is set. If the user limit is greater than the UDB limit, the job is aborted − why waste cycles since it likely would fail assuming the user specified the limit correctly.
2. If a default limit was established by the server (the user didn't supply one). Then the lessor of the server default and the UDB is the true limit.
3. If no limit was supplied, the UDB limit is used.

mom_do_poll()

```
int mom_do_poll(pjob)
```

Args:

pjob A pointer to the job structure representing the target job.

Return:

0 if the job has no machine-dependent resource limits which require polling.

1 if at least one machine-dependent resource limit requires polling.

This function is called by MOM before forking the job as a child. It tells mom whether it will be necessary to poll the job's condition to determine if any specified resource limit has been exceeded.

mom_open_poll()

```
int mom_open_poll()
```

Args:

None.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

This routine's purpose is to establish a connection with kernel data structures which will be used in job resource use polling cycles.

mom_get_sample()

```
int mom_get_sample()
```

Args:

None.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

If there is at least one machine-dependent resource to be polled for at least one job, this routine is called before each MOM resource limit polling cycle. It samples the state of every job on the system in preparation for job-by-job polling.

mom_over_limit()

```
int mom_over_limit(job *pjob)
```

Args:

pjob A pointer to the job structure representing the target job.

Return:

0 if polling the state of the job shows that all resource consumption is within limits.

1 if polling the state of the job reveals that it has exhausted a controlled resource.

MOM's job polling loop calls this routine to see if the specified job is over its limits. The function returns a logical value telling whether to kill the job.

mom_set_use()

```
int mom_set_use(job *pjob)
```

Args:

pjob A pointer to the job structure representing the target job.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

This function sets the job's resources_used attribute to the list of resources and amounts used so far by the job. Any call to this function must appear in the execution order between a call to mom_open_poll and mom_close_poll and must come after the job's session_id attribute is defined. The values it inserts into the resc_used attribute reflect conditions at the last mom_get_sample call. Note, that the attribute JOB_ATR_resc_used is marked as modified, {ATR_VFLAG_MODIFY} set, on each call so the latest information will be returned to the server, see status_attrib().

On the Cray (unicos8 or unicosmk2), if {JOB_SVFLG_Suspend} is set in ji_svrflags, walltime is not updated as the job is suspended.

mom_close_poll()

```
int mom_close_poll()
```

Args:

None.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

Close polling connections to the kernel.

mom_does_chkpnt()

```
int mom_does_chkpnt()
```

Returns:

True (1) if checkpoint supported, false (0) if not.

This routine is machine dependent. The PBS_MACH types unicos8, unicosmk2 and irix6array currently return true.

mach_checkpoint()

```
int mach_checkpoint(job *pjob, int abort)
```

Args:

pjob A pointer to the job structure representing the target job.

abort A logical value specifying whether the job should be aborted after the checkpoint has been taken.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

If checkpointing is not supported on the machine, mach_checkpoint does nothing and returns {PBSE_NOSUP}. Otherwise, for those machine types that support checkpoint, this function causes the designated job to be checkpointed into the restart file named in the job structure. If the checkpoint file already exists from a prior checkpoint, the file is renamed. Additionally, if *abort* is true, the job is killed if the checkpoint succeeds.

If checkpoint succeeds, any old checkpoint file is unlinked. If checkpoint fails, the new file is unlinked, and old file is renames to the original name.

mach_restart()

```
int mach_restart(task *ptask, char *file)
```

Args:

ptaskA pointer to the task structure for task being checkpointed. At the current time, only irix6array/mom_mach.c uses this parameter.

file the path of the task's checkpoint image.

Return:

0 if success.

non-0an error code defined in pbs_errno.h.

If checkpointing is not supported on the machine, mach_restart does nothing and returns {PBSE_NOSUP}. Otherwise, the system's restart call along with any other required supporting code is executed.

kill_task()

```
int kill_task(task *ptask, int signal)
```

Args:

ptaskpointer to task structure.

signalto send to the running task.

Returns:

Number of processes killed, i.e. zero if no processes belonging to the task were found.

This function sends the specified signal to each process which is a member of the task's session. At the current time, most systems do not support a direct method of signaling the members of a session, only a single process or the members of a process group. There may be more than one process group active within the job session.

If the task session number is less than or equal to one (1), then return without doing anything, either the session has not yet been established, it has already been signaled, or the session no longer exists for other reasons.

The function *mom_get_sample()* is called to update the job usage attribute with the latest information.

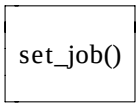
For most systems, the method of signaling the session's processes is to walk the process table looking for any process which is a member of the session. If it is, the system function kill() is

called with the supplied signal. A count of the number of processes found and signaled is returned as the function return.

The Unicos OS does support a kill "job" system call, killm(). Using this call saves walking the process table. However, we can only return 1 or 0 for the kill_task() value since we only know that zero or more than zero processes existed in the task.

8.2.5.3. File: mom_start.c

The file *src/resmom/<machine>/mom_start.c* contains machine dependent code dealing with placing a job into execution and with post termination processing.




```
int set_job(job *pjob)
```

Args:

pjob pointer to job structure

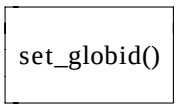
Returns:

zero if successful, non-zero if error.

This dependent routine establishes a new session. Typically, this is done by calling setsid().

The Unicos version requires a bit extra, its concept of "job" is a bit different. Also the batch bit must be set in a sesscnt() call.

Irix 6.x support Project IDs which is an accounting entity. The project id for the job is set here.




```
int set_globid(job *pjob, struct startjob_rtn *sjr)
```

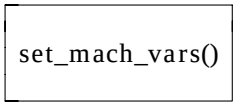
Args:

pjob pointer to job structure

sjr pointer to info returned from new job.

This dependent routine sets a value for the job structure field ji_globid. If any kind of job management software can independently track processes useing a special identifier, that can be formeated into a string in this routine. Otherwise, "none" is filled in.

Irix 6.x supports the Array Session Handle (ASH) which is formatted into a hex number in the ji_globid field. This allows any task spawned to carry the same ASH.




```
void set_mach_vars(char *buffer, int space, char **environ, int enspace)
```

Args:

buffera character buffer in which the various environmental variable strings are placed.

spacethe amount of space (left) in the buffer.

environ
: a pointer to (the first available member of) an array of pointers to strings. This array is included as the environment when the "shell" is exec-ed.

enspace
: the number of unused members in the array environ.

This function is provided in case there is a need for machine dependent environment variables. Any required string of the form "keyword=value" should placed into the buffer provided it does not exceed the available space. A pointer to the start of the string should be placed in environ. No more than enspace - 1 variables should be added.

The enspace array must be null terminated which is all the default function does.

set_shell()

```
char *set_shell(job *pjob, struct passwd *pwd)
```

Args:

pjob pointer to the job.

pwd pointer to the password entry for the user under whose uid the job will be run.

Returns:

pointer
: to the shell to execute as the job.

This routine returns a pointer to the name of the shell program which should be executed. The pointer is to a area which might be overwritten by another call to obtain a different password entry.

The general method of determining which shell is given by the following:

1. The entry in the job attribute *JOB_ATR_shell* which has a host name that matches the current host.
2. The entry in the attribute which has a wild card (null) hostname. IP 3. The user's login shell.

scan_for_terminated()

```
void scan_for_terminated()
```

This routine is called from MOM's main loop when the *termin_child* flag is set in *catch_child()*. Its purpose is to determine which job, if any, has terminated execution and to update the resource usage information for that job.

On most machines, the resource usage information is maintained in the process table of each process and rolled upward to the parent as each child dies. Thus the session leader, the shell, ends up with the total usage numbers. The trick is that the process table entry goes away when the child is reaped. Thus, when MOM received a SIGCHLD and does a wait() to obtain the pid of the dead process to determine which job has terminated, the information in the process table is lost. Hence, for these machines, MOM must before calling wait(), call *mom_get_sample()* to obtain the basic information from the system and then call *mom_set_use()* for each job which might have terminated, i.e. all running jobs.

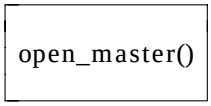
Now, the wait(), actually waitpid(), is called and the returned pid can be matched against the various job's session ids to determine which job has terminated. The exit status returned by waitpid() is saved in ji_exitstat (with some modification) and the job is marked as being in substate {JOB_SUBSTATE_EXITING} to identify the job to *scan_for_exiting()*. exiting_tasks is set so MOM will call scan_for_exiting(). The exit status is filtered by the WIFEXITED and WIFSIGNALED macros. If the exit status is an exit value, it is returned unchanged. If the exit value is a signal number, it plus 10000 is returned. See *req_jobobit()* in server/req_jobobit.c for how the 10000 is used.

See the Cray C90 version for a system which provides integrated support for jobs, limits, and usage.

The Unicos version of scan_for_terminated() also checks if the terminated process was a special helper which was performing a time consuming task for MOM. Such tasks are checkpointing, suspending, or resuming a job. MOM checks the pid returned by waitpid() against *ji_momsubt* in each job. If a match is found, the function pointed to by *ji_mompost* is invoked as

```
void func(job *, int)
```

where the second argument is the exit status of the child.




```
int open_master(char **rtn_name)
```

Args:

rtn_name
: [Return] A pointer to a character pointer in which a pointer to the slave side pseudo terminal name is placed.

Returns:

The file descriptor of the master side pseudo terminal is returned. -1 if one was not opened.

There are several versions of this routine, just about one per system type. Some systems, notably AIX, provide a multiplexor device to provide both the master and slave tty without searching. The Intel Paragon has a similar routine, *openpty()*. On most systems without a multiplexor or library routine, open_master must try opening each possible master name until the open succeeds. The slave name is derived by changing the sub-string

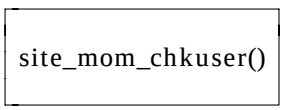
The important issues are to return the file descriptor of the master side as the function return, or -1 on an error, and a pointer to the slave name into the argument.

8.2.6. Site Modifiable Files

MOM contains several modules which are meant to be easily modifiable by a site. The supplied version of these files may be found in the *src/lib/Libsite* directory and are linked via the libsite.a library. How to modify these files is discussed in the IDS chapter on libsite.a.

8.2.6.1. site_mom_chu.c

The file *src/lib/Libsite/site_mm_chu.c* contains the function:



```
int site_mom_chkuser(job *pjob)
```

Args:

pjob pointer to the job being placed into execution.

Returns:

zero if account is valid, non-zero if the account is invalid and hte job should be aborted.

This routine is provided to allow a site to add whatever type of account validation it chooses. It should return non-zero if the job should be aborted for whatever reason. As provided it always returns zero.

8.2.6.2. site_mom_ckpt.c

The file *src/lib/Libsite/site_mom_ckpt.c* contains two functions:

site_mom_postchk()

```
int site_mom_postchk(job *pjob, int hold_type)
```

Args:

pjob pointer to job structure.

type of hold being applied to the job.

Returns:

Zero if successful, non-zero if failed.

This routine is called following a successful checkpoint-and-terminate of a job as the result of a qhold of a running job or a pbs_server shutdown. (This applies only to the Cray implementation.) The return value is used as the exit code of the child process doing the checkpoint. It has little impact on the job.

As an example of usage, at NAS this routine is being used to migrate the checkpoint image of certain large, low priority jobs.

site_mom_prerst()

```
void site_mom_prerst(job *pjob)
```

Args:

pjob pointer to the job structure.

Returns:

zero (0) if successful.

JOB_EXIT_FAIL1
if job should be permanently aborted.

JOB_EXIT_RETRY
if job should be requeued.

This routine is called before a job is restarted from a checkpoint image. (This applies only to the Cray implementation.)

As an example of usage, at NAS this routine is being used to reload the checkpoint image of large, low priority jobs before the restart.

8.2.6.3. site_mom_jset.c

The file *src/lib/Libsite/site_mom_jset.c* contains the following function:

site_job_setup()

```
int site_job_setup(job *pjob)
```

Args:

pjob pointer to the job structure of the the job being placed into execution.

Returns:

Zero on success, non-zero if job should be aborted.

This routine is called from *finish_exec()* shortly after the job session is established. A site may use it to perform any additional session related setup required at that site.

Return zero (0), if the setup is successful, or non-zero if the job is to be aborted.

8.3. Program: pbs_rcp

8.3.1. Overview

Included with the source for MOM, in subdirectory *src/resmom/mom_rcp* is the source code for the rcp(1) command from the bsd4.4-Lite distribution. This code is copyrighted by UCB as noted in the source files. The code has been slightly modified to allow it to compile under systems other than bsd4.4; note the liberal use of functions such as vwarnx() and snprintf() not found in POSIX. The copyright clearly grants the right to modify and redistribute the source.

8.3.2. Why pbs_rcp

Why is this code supplied as part of PBS? Within PBS, there are three cases in which MOM must move files between her machine and some other:

a. Preexecution stage in of files.

b. Post-execution stage out of files.

c. Post-execution return of the job's standard output and standard error.

The PBS project did not wish to be dependent on NFS, AFS, or any other distributed file system in order to support file delivery. Nor did we wish to restrict the source/target of file movement to those systems with a PBS server. This ruled out using the "job" protocol as a file transport. Ftp(1) and ftam require the user's password. We did not wish to require that knowledge. Thus rcp(1) was selected as the transport method. MOM uses the system(3) library routine to execute the rcp command.

However, many rcp implementations come with a serious flaw. They may exit and return an exit status of zero (0), when the file was not delivered. If this happens, MOM would believe that the file was delivered when it was not.

One solution would have been to implement a new copy utility for MOM very similar to rcp. But this would have required it's installation on every system to/from which the user may wish to move files. Rather than duplicate rcp, lets just fix it. As only the rcp used by MOM must be "fixed", the PBS team opted to provide a version of rcp that works correctly. The bsd4.4-Lite version was chosen because of the freedom to copy and modify it granted by its copyright.

8.3.3. Use of pbs_rcp

The supplied rcp source is compiled and the program is named "pbs_rcp" in order to reduce the level of confusion on having two "rcp"s installed on the system. It is installed in the same system binary directory as MOM (pbs_mom). This path is compiled into MOM, see *src/resmom/requests.c*.

When MOM invokes pbs_rcp, MOM has forked a child which as set its effective and real uid to that of the user on whose behalf MOM is operation. This child of MOM, as the user, will use system(3) to fork a shell and execute pbs_rcp. The path to the pbs_rcp is specified in building src/resmom/requests.c and contains the directory where MOM is (will be) installed.

Pbs_rcp, as in normal rcp, must be installed "setuid" and owned by root.

9. IFF - User Credential Granter

9.1. PBS_IFF Overview

As the PBS server is happy to act upon jobs if the user's name in the batch request is the same as the job owner, there must be a method of authenticating or proving that the requesting user is who they claim to be. The standard practise in socket/TCP/IP based applications for user authentication is through the use of "reserved" or "privileged" ports. This provides weak authentication, but it is about the only common game in town, especially if you cannot use encryption. Typically a client cannot bind to a port with a number less than 1024 unless the client is running with root effective privilege. This approach is fine when the clients are fixed and few in number.

As PBS provides the API library and allows users to write their own PBS clients, another method of user authentication is required which does not require root privilege of each client program. In the basic PBS system, user authentication is achieved by the program **pbs_iff**, *identification, friend or foe*.

The basic scheme is to have the client program fork and execute a root privilege program (setuid) which using its real uid to obtain the user's name and its root effective uid to gain access to a privileged port. This process is known as **pbs_iff**. Pbs_iff sends a authentication message to the server over the privileged port. The authentication message contains the requesting user's name, host name and the name (number) of the non-privileged port the parent PBS client is using to contact the server. The server will associate the user and host name with the clients port. As long as the requests that arrive on that port originate from the named host and the user name in the request matches the associated name, the server will accept the request. This scheme is shown in figure 9−3.

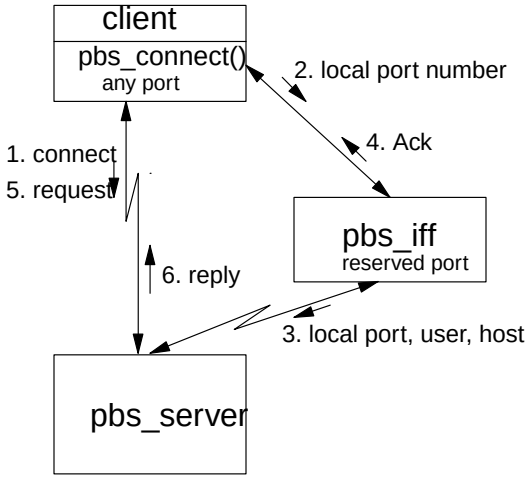


Figure 9−3: PBS IFF Program

The client API library interfaces with pbs_iff via one call within *pbs_connect()*. More information is available in the API section of the IDS.

The simple interface between pbs_connect() and pbs_iff on one end and the server on the other allows for easy replacement of pbs_iff with the user authentication of your choice, such as Kerberos.

9.2. Packaging

The following source files are required to build pbs_iff:

pbs_iff.c

The following libraries are used when linking pbs_iff:

libnet.a libpbs.a and the whole ISODE mess.

9.2.1. External Interfaces

Pbs_iff is called with the usage:

```
pbs_iff [-t] server_host_name server_port_number [client_socket]
```

Where *server_host_name* is the name of the server to which the client will connect. *Server_port_number* is the port number to which the server listens, typically 15000. If the *-t* option is specified, it indicates a test mode. In this mode, pbs_iff sends a message to the server to test connectivity. If *-t* is not given, the normal case, then *client_socket* is required. It is the socket number which the calling client has opened and connected to the server.

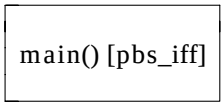
Pbs_iff is forked and exec-ed by a client program, typically by a *popen(3)* call. Pbs_iff returns the type of authentication performed to the client by writing an identifying integer on its standard output if that stream is connected to a pipe.

Provisions are made for having pbs_iff return a credential to the PBS API. In the current release this is not used, a credential type of {int_BATCH_credentialtype_credential__none} is returned and nothing else. If the authentication type is not as defined in the Batch Protocol, then the size of the size of the credential is returned as an integer, followed by the credential itself.

Pbs_iff will create a socket and bind it to a reserved port. This socket is then connected to the server.

9.2.2. File: pbs_iff.c

The file *src/iff/pbs_iff.c* contains the source for the main function of pbs_iff.




```
pbs_iff [-t] server_host_name server_port [client_socket]
```

Args:See above under "External Interfaces".

A test is made to make sure that pbs_iff's standard output is connected to a pipe. The main routine of pbs_iff obtains the real user id and converts that to the user's login name. The server's network address is obtained by calling *get_hostaddr()*. The function *client_to_svr()* is called to allocated and bound (bind()) a socket to a reserved port and make a connection to the server.

If not test mode (-t option), the port name of the client's socket is obtained via getsockname(). If test mode is set, then the port number of pbs_iff's socket is used instead as there is likely not client socket. This port number is placed in the Authenticate User batch request and sent to the server over the privileged port.

If the server accepts the request, pbs_iff returns {int_BATCH_credentialtype_credential__none} to the client. Otherwise, pbs_iff exits without writting to the client.

10. Libraries

10.1. Library: libattr.a - Attribute Library libattr.a

The attribute library, *libattr.a*, consists of routines to manipulate or otherwise handle PBS attributes. This library is used by the server.

As discussed in section 2.1.1, an attribute consists of a name and a value. An attribute can be represented in one of two ways, three if you count the form used by the network encode/decode routines.

The internal form of an attribute is contained in two separate structure: the attribute definition which contains the name, various flags, and pointers to the manipulation routines for that attribute, the *attribute_def* structure; and the *attribute*, structure which contains the machine dependent representation of the value and other flags.

The network independent external form has the value as well as the name in string form. This form is contained in a *svrattrl* structure. When in the external form, the attribute is referred to as being *encoded*, and when in the internal form as being *decoded*.

10.1.1. attr_func.c

The file *src/lib/Libattr/attr_func.c* contains general purpose functions relating to processing attributes.

clear_attr()

```
void clear_attr(attribute *pattr)
```

Args:

pattr Pointer to attribute to clear

The attribute (value) structure is cleared by zeroing each byte. Then the attribute flag {ATR_VFLAG_SET} is cleared.

Note, this function is useful in clearing an attribute structure which has just be allocated. If used to clear a attribute to which a valued has been assigned, the appropriate free routine, see at_free(), must be called before clear_attr is called.

find_attr()

```
int find_attr(attribute_def *attr_def, char *name, int limit)
```

Args:

attr_def
Pointer to the attribute definition structure. This structure contains the name of the attribute.

nameThe name of the attribute to find in the definition structure.

limit The number of attributes defined by the definition structure, the limit on the search.

Returns:

>= 0 The index into the attribute definition (and attribute value) structure.

-1 If error occurred.

The function compares the requested name with the name of each attribute defined in the definition structure. If a match is found, the index into the array of definitions is returned.

Given an attribute in string form, a name string and a value string, this function is used to obtain the index which relates the attribute to both the definition and value structures.

free_null()

```
void free_null(attribute *attr)
```

Args:

attr Pointer to the attribute (value).

This function is a semi-place holder in the attribute definition for attributes which do not have extra space allocated to hold their values. Attributes of type long, boolean, character, and size use this function.

The value field, using at_size as an expedient, is zeroed and in at_flags {ATR_VFLAG_SET} is cleared.

attrlist_alloc()

```
svrattrl *attrlist_alloc(int szname, int szresc, int szvalue)
```

Args:

szname
size of attribute name including terminating null.

szrescsize of resource name including terminating null, zero if no resource name.

szvalue
size of value string including terminating null.

Returns:

pointer
to the created svrattrl entry.

null if error.

The svrattrl structure is used to hold the encoded form of an attribute. It serves two functions. First, it is a single block holding all of the attribute information which can be saved on disk. Second, it provides for a structure that is independent of but similar to the network form of an attribute. The svrattrl structure also contains the *attropl* structure used by many of the routines in the API library, libpbs.a.

The svrattrl structure has two parts, the base portion and the extended portion. The base portion is that defined by the structure definition itself. The extended portion is the extra space allocated immediately following the base and used to hold the strings. Note that the al_flags entry is only used when saving the attribute to disk and the al_op entry is only used for network encoding.

The length of the three strings added to the size of the svrattrl structure itself is the amount of space allocated. The string length members and the pointers to the strings are set. The

other fields are cleared. A pointer to the structure is returned.

attrlist_create()

```
svrattrl *attrlist_create(char *atname, char *rsname, int vsize)
```

Args;

atname
: the attribute name.

rsname
: the resource name or a null pointer.

vsize the amount of space, in bytes, required to hold the encoded value.

Returns:

pointer
: to the created svrattrl entry.

null if error.

The length of the attribute name and the resource name (if present) are determined. Those lengths and the passed size of the value are passed to the function *attrlist_alloc()* to allocate the space.

The attribute name string and if present, the resource name string are copied into the structure. A pointer to the structure is returned.

attrl_fixlink()

```
void attrl_fixlink(list_head *svrattrl_list)
```

Args:

svrattrl_list
: pointer to the head of the list of svrattrl structures.

This routine is a kludgey solution to problem created by the implementation. The design of the API, see pbs_ifl.h and libpbs.a, specified two structures to pass attribute information into the API routines, attrl and attropl. The server also needed a similar structure, but one which was a single unit actually holding the strings as well as pointing to them.

After several iterations, the current svrattrl structure was developed for the server. This structure contain a attropl structure. Thus it can be feed to the API routines without yet another round in the seemingless cycle of format conversion.

However, there is still one problem. The included attropl structure has a "next" pointer which must point to the next attropl structure when the list is passed to the API routines. The server does not use this pointer for two reasons. First, the server already has routines for dealing with a linked list using double linking. Even though the double linking does not add any required functionality, the routines exist, so use them. Second, the next pointer in the attropl structure must point to the next attropl structure. Even if the attropl is the first element in the svrattrl structure, there is a (remote) chance that the next attropl pointer would not be valid as pointer to the next svrattrl.

So we end up with two linkages. The server makes use of *al_link*, the double linkage. Only when the list is being passed to one of the API routines is *al_atopl.next* used. Thus the pur-

pose of attrl_fixlink, to "fix" the attropl linkage to match the svrattrl linkage.

The svrattrl list is walked and each al_atopl.next is set to point to the next al_atopl entry.

free_attrlist()

```
void free_attrlist(list_head *attrlist)
```

Args:

attrlist
: pointer to head of list of svrattrl entries.

Each entry in turn is deleted from the list and freed.

parse_equal_string()

```
int parse_equal_string(char *start, char **name, char **value)
```

Args:

start the start of the string to parse, or null to continue where left off.

nameRETURN: pointer to first element of string, the name is returned.

valueRETURN: pointer to second element of string, the value is returned.

Returns:

function value
: 1 if parse successful,
0 if at end of string, or
-1 if syntax error detected.

*name
: as described above.

*value
: as described above.

This routine parses a string of the form:
`name_1 = value_1[, name_2 = value_2...]`
On the first call to parse_equal_string(), start points to the beginning of the string. A pointer to `name_1` is returned in name, and to `value_1` in value. The value string is null terminated at the comma.

On each following call with start set to the null pointer, the routine will resume the parse where it terminated before, returning `name_2` and `value_2`.

White space around the name and the equal sign is ignored. Commas may be embedded in the value string by enclosing the value string in either single or double quote marks. No parsing takes place until the matching quote is found.

parse_comma_string()

```
char *parse_comma_string(char *start)
```

Args:

start a pointer to a string of comma-separated items, or the null pointer to resume where the prior call left off.

Returns:

pointer
to the first (next) item. A Null pointer is returned when the string is exhausted.

On the first call, start points to a string of the form:
`value_1 [, value_2 ...]`.
Leading white space is skipped and the first item is null terminated at either the first following white space or the comma, which ever occurs first. The function return is a pointer to the start of the first item: `value_1`.

On any following calls, with start set to the null pointer, the scan will resume where it terminated before. When the end of the original string is reached, a null pointer is returned.

10.1.2. Attribute Manipulation Functions

The remaining files which make up libattr.a are attribute manipulation functions. Each attribute defined in an attribute definition structure has a set of manipulation functions which are specific to that type of attribute. The prototypes are declared in attribute.h and the calling sequence for each function is described here (rather than repeat N times and waste paper). The functions are:

at_decode

Convert the value string to the internal representation. For example, for an attribute of type "long", the string "123" is decoded to the integer 123. The typical function for decoding is named **decode_***type*, where *type* is the type of data. Note, if the attribute already has a value, it should be cleared. This is especially true if the value takes up addition storage (type _str, _arst, _resc, _list).

Important

If the decode is successful, the attribute is marked with {ATR_VFLAG_SET} and {ATR_VFLAG_MODIFY}. If the value is the null string, it is assumed the attribute is to be "unset". The attribute is marked with {ATR_VFLAG_MODIFY} and {ATR_VFLAG_SET} is cleared.

Args:

pattr Pointer to the internal attribute into which the value is decoded.

nameThe attribute name. This is most often unused, the exception is decode_unkn().

resource
The resource name. This is unused except in decode_resc().

valueThe value as a comma-separated string.

Returns:

0 Ok, *pattr is set to the new decoded value.

>0 If an error occurred, the PBS error number is returned.

at_encode

Convert the internal representation of the value to the svrattrl form. This consists of a string for the attribute name, one for the resource name if the attribute is a resource item, and one for the value. These strings are contiguous and are headed by a control structure, which specifies the total length of strings and control structure, the length of each individual string, and pointers to each string. The typical function for encoding is named **encode_***type*, where *type* is the type of data.

Attributes which are not set, i.e. {ATR_VFLAG_SET} is cleared or which in the case of attributes with external data, have no value, are not encoded. The return value is zero.

Args:

attr Pointer to the attribute to decode.

phead Head of the list of svrattrl structures to which the encoded value is appended.

atname
The name of the attribute. This string is included in the svrattrl entry as the attribute name. It is typically taken from the attribute definition structure.

rsname
The name of the resource item if the attribute is a resource list. Otherwise, this is a null pointer.

mode The mode of the encode operation. Certain attributes change form depending on the recipient The mode is either {ATR_ENCODE_CLIENT} if the data is to be sent to a client or another server, or {ATR_ENCODE_MOM} if the data is to be sent to MOM when starting a job. {ATR_ENCODE_SVR} if the data is to be sent to another server, the job is being routed. {ATR_ENCODE_SAVE} if the data is being encoded to save on disk. This is unused by most attribute types. See encode_arst() for an example of where it is used.

Returns:

>0 if the encode was successful.

0 if no data encoded because the attribute was not set.

<0 if the encode failed, typically due to a malloc failure.

at_set

Sets the value of an attribute (internal form) to another. There are three operations defined, set A = B, increment A = A + B, and decrement A = A - B. The operators + and - are overloaded depending on the type of attribute. Not all operations are supported for all attributes. The typical function for setting an attribute type is named **set_***type*, where *type* is the type of data.

Important

If the set operation is successful, the attribute is marked with {ATR_VFLAG_MODIFY}. {ATR_VFLAG_SET} is set or cleared depending on whether or not the attributes ends up with a value.

Args:

old Pointer to the attribute to set.

new Pointer to the attribute holding the value to use in setting attr.

op The operation to perform: Set, Incr, or Decr.

Returns:

0 If ok.

>0 If an error, the PBS error number is returned.

at_comp

Returns the results of a comparison of two attributes. The typical function for comparing is named **comp_***type*, where *type* is the type of data.

Note, *comp_resc()* behaves somewhat (is that an understatement) differently.

Args:

one A pointer to an attribute.

two A pointer to another attribute.

Returns:

0 The two attributes are set to the same value, V1 equals V2.

+1 The values of the two attributes are different in some fashion. For numeric data or other attributes for which the full set of comparisons exist, +1 is returned for V1 > V2. For attributes where only equality or inequality comparisons, +1 is returned for V1 not equal to V2.

-1 The values of the two attributes are A1 < A2, if this relation is defined.

at_free

Frees the space allocated to hold the attribute value. Certain types of attributes, long, boolean, character, do not have "extra" space allocated. A null function is used for these types, see *free_null*. The typical function for freeing an attribute value is named **free_***type*, where *type* is the type of data.

Args:

pattr Pointer to attribute for which value space is to be freed.

at_action

Performs an action when the attribute value is modified. There are many times that the server must perform some action when the value of an attribute is changed by a client. An example, when a client changes the value of a hold-type job attribute, the server must update the job state.

The *action function* provides a uniform interface to provide such a function. If the action function pointer is non null, the action function should be called when a client request modifies the attribute value. If the server is doing the modification "on its own," then it should know enough about what else to do. The typical function for performing an action upon a change in value is named **action_***type*, where *type* is the type of data.

Args:

pattr Pointer to attribute which was modified.

pobject Pointer to object which is the parent of the attribute: the job, queue, or server structure.

mode Indicates the circumstances under which the attribute is being:

ATR_ACTION_NEW
Set for the first time, the parent object is new.

ATR_ACTION_ALTEROLD
Altered from the current value to a new value.

ATR_ACTION_ALTERNEW
Altered to this new value.

ATR_ACTION_RECOV
Recovered from disk.

ATR_ACTION_FREE
Freed or unset.

ATR_ACTION_NOOP
A special flag to indicate that nothing should be done. This is currently unique to the routine *depend_on_que()* which does double duty.

Not all of the above values for mode are being used. They include about every possibility that the author could imagine.

Returns:

0 If ok

-1 If error

Theses functions are found in files with the names **attr_fn_*.c**. The specifics of each function for each attribute type is discussed below. If new attribute types are defined, not just new attributes of an existing type, then a new set of functions must be implemented.

It is important to understand the role of two flags in all these operations:

ATR_VFLAG_SET
indicates the attribute has a value. This flag is required because with long integer attributes, any value is possible.

ATR_VFLAG_MODIFY
indicates the attribute value has changed. This could be from unset to set, from one value to another, or from set to unset.

10.1.2.1. attr_fn_acl.c

The functions in file *src/lib/Libattr/attr_fn_acl.c* implementing the various *Access Control Lists*. There three types of access control lists supported:

Host The host access list is a list of host and/or domain names in the standard Internet *host.domain.name* form. This list list is generally used by the server to restrict services to a set of machines. It may also be used to limit access to any queue to jobs from a set of hosts. The name should be prefixed by a '+' or '-' (minus/hyphen) character to indicate that access by the named entity is to be allowed or denied. Absence of either character is taken to mean access is allowed.

A host name should be fully qualified, that is be specified with the full domain name appended: host.domain.name. A domain name must be prefixed by "*.". For example, "-*.foo.bar" means access from any system in **domain** foo.bar is to be denied. While "+foo.bar" means access from the **host** foo.bar is allowed.

Entries are sorted in a special order, from the tail end backwards to the head. This forces the more fully specified entries to the front of the list where they will be found first during a search for a match. Any candidate host that compares equal to an entry from the tail end to the head or up to an asterisk is considered a match.

User User access control lists may be used to restrict certain queues to jobs from a specified list of users. The user list format is very similar to that of the user-list job attribute. Each entry is in the form user-name@host.domain, where host may be wild carded with the use of an '*'. Note, if just a user name is supplied, the server will append "@*". Each entry should be prefixed by a '+' or '-' as with the host ACL.

The ordering of the entries in the list is by sorting on the user name first, then on the host/domain name in the same fashion as with the host ACL.

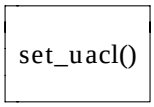
GroupThe group list is the simplest list. It is just a list of group names. Typically, a group ACL is only applied to an execution queue to restrict it to members of certain groups. For a job, the group compared against the list is the group under which the job would execute (which is why we don't have group ACLs on routing queues). There is no special format or sorting performed for group ACLs.

The entries in any list are created from the comma-separated values in a Manager batch request that sets the attribute, see qmgr(1) and pbs_manager(3). The individual entries may be provided in any order. The host ACL function *set_hostacl()* and the user ACL function *set_uacl()* will sort the entries as they are added. This sort order is important to acl_check().

The access list may be lengthy. For this reason, the list is not saved with the other attributes of the parent object. Rather each list is maintained in its own file. The location of the file de-

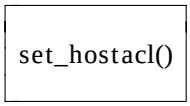
pends on the parent object. The server host ACL is under the server database directory. All of the queue host ACL files are under a directory with the name of the attribute. The file's base name is the queue name.

The internal and external forms of the host access list are identical to the array of strings form used by attributes of type {ATR_TYPE_ARST}. In fact, the at_decode, at_encode, at_comp, and at_free routines for access list attributes are decode_arst(), encode_arst(), comp_arst() and free_arst(). The at_set routine for hosts (set_hacl) and users (set_uacl) are different to provide the special sorting needed for a host access list. A new routine, *acl_check()*, provides the capability to check a candidate name against the access list.



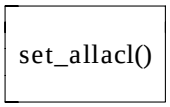

```
int set_uacl(attribute *attr, attribute *new, enum set_op op)
```

This routine is just a wrapper. It calls *set_allacl()* with the additional parameter which indicates the ordering function for the user type of ACL.




```
int set_hostacl(attribute *old, attribute *new, enum set_op op)
```

Again, this is just a wrapper that calls *set_allacl()* with the ordering function specific for the host ACL type.




```
static int set_allacl(attribute *pattr, attribute *new, enum set_op op,
          int (*order_func)() )
```

The set_allacl function updates the array of strings for the attribute pointed to by old according to the operation.

- Set The complete set of access list entries in the old attribute are replaced by those in the new attribute. This is done by freeing the old array and rebuilding it one entry at a time to place them in sorted order. [Its slow to execute, but quick to write, look to optimize later.]
- Incr Each entry in the new list is added to the old list. The new entries are sorted into place. The ordering function order_func() passed as a parameter is used to determine the relative order of two entries. When a new entry is to be inserted between existing entries, both the index to and the actual strings of following entries must be moved up to make room for the new entry. This is a bit messy as the memcpy() does not guarantee to work if the two areas overlap as they will here. So we code the move character by character.
- Decr As in set_arst(), any matching substrings are removed. Each entry in turn is checked for a match. Note, the prefix characters, '+', '-', or '*' must also match.

acl_check()

```
int acl_check(attribute *pattr, char *entry, int type)
```

Args:

pattr Pointer to the host access control list attribute.

entryName of the user, group, or host requesting access.

type the type of the ACL: or

Returns:

1 if the host name matched an entry permitting access.

0 if access is not permitted.

This function will look for a matching entry in a access control list. It starts with the first entry and works till a match is found or the end of the list is reached. Any single leading '+' or '-' in the list entry is ignored in determining a match. A match is found when the supplied entry matches an entry in the list. All comparisons are performed with a routine specific to the ACL type. With the list sorted as discussed earlier, the first match found will always be the most specific match possible in the list.

If a match is found, access is denied if the list entry begins with a "-" character. Otherwise access is allowed. If a match is not found but an entry with only a "+" or "-" character was noted, then that + or - established the default access (allowed for +, denied for -). If a default entry was not found, then access is allowed if the routine was complied with {HOST_ACL_DEFAULT_ALL} defined, otherwise denied.

For a host access list, if the list is not set or is empty, access is still allowed from the local host. For all list types, if this routine is complied with {HOST_ACL_DEFAULT_ALL} defined, then anybody is allowed access if the list is empty or unset. This is very insecure and not recommended.

user_match()

```
static int user_match(char *candidate, char *master)
```

Args:

candidate
a candidate user_name@host requesting access.

master
an entry from the user ACL.

Returns:

0 if the entries match.

1 if they do not match.

This function is used by *acl_check()* for user ACLs. The candidate is compared against the ACL entry starting with the user name up to the end of the master or the occurrence of an '@' in the master. If the master ended, the candidate must also end or start the "@host" section. If the master contains an '@', so must the candidate. If master and candidate match so far through the user name, then *hacl_match()* is called with the host sub-strings (following the asterisk) to complete the comparison.

user_order()

```
static int user_order(char *s1, char *s2)
```

Returns:

- -1 s1 sorts before s2.
- 0 s1 and s2 are equal.
- +1 s1 sorts after s2.

S1 and s1 are stings of the form username@host.domain. The two strings are compared first from the start of the user name up to the asterisk and then from the tail end back to the asterisk by calling *host_order()*.

hacl_match()

```
static int hacl_match(char *candidate, char *master)
```

Args:

- candidate
  - a candidate host name (with domain name) requesting access.
- master
  - a entry from the host ACL of the form host.domain.name or *.domain.name.

Returns:

- 0 if candidate matches master.
- 1 if not a match.

The character by character comparison begins at the tail end of the two strings and continues until (1) a character does not match, (2) the head of either string is reached.

If the two strings are identical, or if the head of the master entry is reached first and its first character is the wild card asterisk, then the two stings are considered equal.

host_order()

```
static int host_order(char *s1, char *s2)
```

Args:

- s1 a string which is a host acl entry.
- s2 a string which is a host acl entry.

Returns:

- -1 if s1 sorts before s2
- 0 if s1 sorts equal with s2
- 1 if s1 sorts after s2

This routine is called by set_allacl() when it is determining the order of the new host acl entry in the list. It compares two strings from the trailing end first. A leading '+' or '-' characters are ignored on each string. The first inequality before the head of the string terminated

the comparison. If the strings are equal in length and sort equal up to the first character of the strings, and one string has a first character that is an '*', then that string is sorted after the other. This places domains after specific host names.

10.1.2.2. attr_fn_arst.c

The functions in file *src/lib/Libattr/attr_fn_arst.c* deal with attributes of type **arst** − **ar**ray of **st**rings. The external form of the value is a comma-separated set of strings: `string1,string2,string3,....` The internal form of attribute (value) is a pointer to a control structure:

```
struct array_strings {
int     as_npointers;   /* number of pointer slots in this block */
int     as_usedptr;     /* number of used pointer slots          */
int     as_bufsize;     /* size of buffer holding strings        */
char   *as_buf;         /* address of buffer                     */
char   *as_next;        /* first available byte in buffer        */
char   *as_strings[2]   /* first (two) strings pointers          */
};
```

The above structure is allocated. A separately allocated buffer is used to hold the comma-separated string which has the commas replaced with nulls, turning it into multiple strings. Each "sub−string" is pointed to by a member of the array of pointers, as_strings.

When more than two sub-strings exist, the array_strings structure is expanded to allow more than two members in as_strings.

decode_arst()

```
int decode_arst(attribute *pattr, char *name, char *rescn, char *val)
```

If the value string is null, then clear the ATTR_FLAG_SET flag to indicate no value present and return.

Allocate a buffer to hold sub−strings. Replace commas with null and count the number of sub−strings. Allocate array_strings control structure large enough for pointers to all sub−strings and initialize the pointers.

Set the ATTR_FLAG_SET flag.

encode_arst()

```
int encode_arst(attribute *pattr, list_head *phead, char *atname,
    char *rsname, int mode)
```

The function *attrlist_create()* is called to create a svrattrl entry of size sufficient to contain all the data in the array of strings. The strings in the array are concatenated together. The nulls separating the encoded sub−strings are replaced with either commas for all modes except {ATR_ENCODE_SAVE}. For save, the sub−strings are jointed with new-lines. The {ATR_ENCODE_SAVE} mode is used for access control lists or other items taken from a editable file. This produces one big string. This string is copied into the svrattrl entry.

set_arst()

```
int set_arst(attribute *old, attribute *new, enum set_op op)
```

If the operation is *Set*, a copy of the attr_strings and buffer replace that of the old attribute. The old attribute attr_strings and buffer are freed.

If the operation is *Incr*, each sub−string in new is added to the set of sub−strings in the old.

If the operation is *Decr*, the first sub−string in old which matches a sub−string in new is removed.

comp_arst()

```
int comp_arst(attribute *one, attribute *two)
```

For each index, that sub−string in the two attributes are compared. If all match, then 0 is returned, the attribute are identical in content. Otherwise, +1 is returned for inequality.

free_arst()

```
void free_arst(attribute *pattr)
```

Frees both the array_strings structure and the associated buffer.

arst_string()

```
char *arst_string(char *string, attribute *pattr)
```

Args:

string a "prefix" string for which to search.

pattr pointer to an attribute of type ATR_TYPE_ARST.

Returns:

pointer
to the attribute value entry, or null.

This function searches the sub-strings which make up the attribute value for one with begins with the string passed in string.

If the attribute is unset, or there is no entry with a match, a NULL pointer is returned. Otherwise, a pointer to the sub-string entry is returned.

10.1.2.3. attr_fn_b.c

The file *src/lib/Libattr/attr_fn_b.c* contains the manipulation functions for attributes of type *boolean*. The boolean value is encoded as an long integer with a value of 1 for true and 0 for false. No extra value space is required.

decode_b()

```
int decode_b(attribute *pattr, char *name, char *rescn, char *val)
```

If the character string *val* is the string *ATTR_TRUE* ("1"), the value is to true (integer 1). If the character string *val* is the string *ATTR_FALSE* ("0"), the value is to true (integer 0). Any other value in *val* causes an error return.

encode_b()

```
int encode_b(attribute *pattr, list_head *phead, char *atname,
     char *rsname, int mode)
```

The function *attrlist_create()* is called to create the svrattrl entry. The value is encoded into a string and placed into the entry.

set_b()

```
int set_b(attribute *old, attribute *new, enum set_op op)
```

The value of old is set according to the op:

Set old set to new.

Incr old set to old inclusive or-ed with new, set bits in old that are set in new.

Decr old set to old and-ed ed with not new, clears bits in old set in new.

comp_b()

```
int comp_b(attribute *pattr, attribute *with)
```

If the two values are equal, 0 is returned. Otherwise +1 is returned,

10.1.2.4. attr_fn_c.c

The file *src/lib/Libattr/attr_fn_c.c* contains the manipulation functions for attributes of type (single) character. The character is stored directly in the attribute structure, no additional space is required.

decode_c()

```
int decode_c(attribute *pattr, char *name, char *rescn, char *val)
```

If *val* points to a string, the decoded value is the first character of the string. Otherwise an error is returned.

encode_c()

```
int encode_c(attribute *pattr, list_head *phead, char *atname,
     char *rsname, int size)
```

The function *attrlist_create()* is called to create a svrattrl entry. The attribute character value is encoded as a null terminated string of length one and placed in the entry.

set_b()

```
int set_b(attribute *old, attribute *new, enum set_op op)
```

The value of old is set according to op:

Set Old is set to new.

Incr For lack of anything better, old is set to the character represented by the integer sum of old and new.

Decr For lack of anything better, old is set to the character represented by the integer difference of old minus new.

comp_c()

```
int comp_c(attribute *pattr, attribute *with)
```

If either attribute pointer is null, -1 is returned, otherwise the normal (ascii) relation between two characters is returned.

10.1.2.5. attr_fn_hold.c

The file *src/lib/Libattr/attr_fn_hold.c* contains special decode and set routines for the hold_types attribute. All other attribute functions are those for standard string attributes.

decode_hold()

```
int decode_hold(attribute *pattr, char *name, char *rescn, char *val)
```

This function is identical to *decode_str()* except that it requires the value string to contain only the characters

set_hold()

```
int set_hold(attribute *old, attribute *new, enum set_op op)
```

This function is similar to *set_str()* except that each hold type character is only allowed to appear in the value string once. For the operation type:

Set If the attribute to be set has a value string, it is freed and we fall into the Incr code.

Incr If the attribute to be set (the old attribute) has a value, the space is reallocated to extend the space by the size of the new string. Otherwise, space for the new string is allocated. Each character in the new string not already in the old string is appended.

Decr Each character in the new string that occurs in the old string is removed from the old string. Any following characters are pushed up over top of the removed character.

comp_hold()

```
int comp_hold(attribute *pattr, attribute *with)
```

If either attribute pointer is null, -1 is returned, if the two hold values are identical 0 is returned, otherwise +1 is returned.

10.1.2.6. attr_fn_inter.c

The file *src/lib/Libattr/attr_fn_inter.c* contains the manipulation functions for an attribute of type long that is treated in a special manner. Internally, if set and non-zero, the job is an interactive job. The numerical value is number of the port to which qsub is listening for a connection from the job to transfer input and output. This is the value sent by qsub and between servers if the job is moved. It is also sent to MOM when the job is run. However, to keep the port number confidential, when the job attributes are being encoded to send to a client (command), the value is encoded as a boolean, true for non-zero and false if zero or unset. The long integer routines are used for all functions except except for the encode, see attr_fn_l.c.

encode_fn_inter()

```
int encode_inter(attribute *pattr, list_head *phead, char *atname,
     char *rsname, int mode)
```

10.1.2.7. attr_fn_l.c

The file *src/lib/Libattr/attr_fn_l.c* contains the manipulation functions for an attribute of type long integer.

decode_l()

```
int decode_l(attribute *pattr, char *name, char *rescn, char *val)
```

If *val* points to a numeric decimal string, it is decoded into its value. Otherwise an error is returned.

encode_l()

```
int encode_l(attribute *pattr, list_head *phead, char *atname,
     char *rsname, int mode)
```

The function *attrlist_create()* is called to create an svrattrl entry containing the attribute name. The integer value is encode by *sprintf()* in a null terminated numeric string which is copied into the entry.

set_l()

```
int set_l(attribute *old, attribute *new, enum set_op op)
```

set_l updates the attribute value with the new value based on the operation type. For long integer the operations are as defined as expected:

Set The replacement of the old value with the new.

Incr The sum of the old and new values.

Decr The difference between the old and new.

comp_l()

```
int comp_l(attribute *attr, attribute *with)
```

comp_l compares the long integer values of the attributes attr and with. The normal relation is returned.

10.1.2.8. attr_fn_ll.c

The file *src/lib/Libattr/attr_fn_ll.c* contains the manipulation functions for an attribute of type *Long* integer. The type Long is defined as the largest supported integer type. See the section **Long Long Integer Attribute Support** near the end of this chapter on libattr.a for information about the Long data support routines.

decode_ll()

```
int decode_ll(attribute *pattr, char *name, char *rescn, char *val)
```

If *val* points to a numeric decimal string, it is decoded into its value. Otherwise an error is returned.

encode_ll()

```
int encode_ll(attribute *pattr, list_head *phead, char *atname,
     char *rsname, int mode)
```

The function *attrlist_create()* is called to create an svrattrl entry containing the attribute name. The Long integer value is encode by *uLTostr()* in a null terminated numeric string which is copied into the entry.

set_ll()

```
int set_ll(attribute *old, attribute *new, enum set_op op)
```

set_ll updates the attribute value with the new value based on the operation type. For Long integer the operations are as defined as expected:

Set The replacement of the old value with the new.

Incr The sum of the old and new values.

Decr The difference between the old and new.

comp_ll()

```
int comp_ll(attribute *attr, attribute *with)
```

comp_ll compares the Long integer values of the attributes attr and with. The normal relation is returned.

10.1.2.9. attr_fn_resc.c

The file *src/lib/Libattr/attr_fn_resc.c* contains the manipulation functions for attributes of type *resource*. Resource attributes are slightly different from other types in that they are a linked list of resources. Each named resource has its own definition structure. The resource value is maintained in a *attribute* sub-structure to allow sharing of decode, set, and compare functions with attributes of the same data type.

Members of a set of resources are not restricted to be of the same data type. Thus the "parent" attribute functions work through a process of (1) identifying the resource by name and (2) invoking the corresponding resource function. See the header file resource.h.

Another difference between regular attributes and resources is the extra level of naming. The resource name, in the external form, is passed as part of the value string, the resource value string is in the form: `resource_name=resource_value`, for example, `cput=10`. The parent resource attribute function will separate the value string into two sub-strings by replacing the '=' sign with a null. It uses the first sub-string to identify the resource and then passes the following sub-string as the "true" value string to the proper decode function.

decode_resc()

```
int decode_resc(attribute *pattr, char *name, char *rescn, char *val)
```

Find resource definition entry matching the resource name which is the first string in the "value" string *val*. This done by calling *find_resc_desc()* and *find_resc_entry()*. If there is not a entry in the list for the named resource, the function *add_resource_entry()* is called to create one.

The pointer to the value string is then reset past the resource name, and the decode function (rs_decode) for the specific resource type is called.

In order to decode a resource, the caller must have either some form of write access to the resource set in resc_access_perm, or it must be set to the special value {ATR_DFLAG_ACCESS} which is done when recovering attributes and resources at server startup, see attr_recov().

encode_resc()

```
int encode_resc(attribute *pattr, list_head *phead, char  *atname,
    char *rsname, int size)
```

This encode routine is a bit different from most. It has to iterate through the linked list of resources, encode each into a *svrattrl* form including the resource name.

Each resource value entry, which is very similar to an attribute value, contains a pointer to the resource definition structure. The resource name is obtained from the definition structure and passed to the resource specific encode routine to create the svrattrl entry.

If the encode mode is set to {ATR_ENCODE_CLIENT} or {ATR_ENCODE_MOM}, then all resources for which the setting of resc_access_perm includes any read access, including any resource entries with the {ATR_VFLAG_DEFLT} flag set are encoded.

Otherwise if the encode mode is set to {ATR_ENCODE_SVR} or {ATR_ENCODE_SAVE} all entries except those with {ATR_VFLAG_DEFLT} set are encoded. This allows default to be status-ed or passed to MOM, but not saved, nor passed to a different server.

set_resc()

```
int set_resc(attribute *old, attribute *new, enum set_op op)
```

This function sets the resource value in the list headed by the attribute *old* to the corresponding resource value in the list headed by the attribute *new*. This is repeated for each resource in new. A resource may not be specified in new which does not exist in old except when the set function is *Set*.

For each resource in new, find the resource in old pointing to the save resource_def structure, "it has the same name." If one is not found in old, and the set operation is not Set, return an error. If the operation is set, then create the corresponding resource in old via add_resource_entry().

Invoke the rs_set() function for this resource.

comp_resc()

```
int comp_resc(attribute *pattr, attribute *with)
```

This routine behaves differently than is specified for at_comp().

Since a resource attribute is a list of separate resources, no single return can indicate the relationships of all the resources in the two attributes. Instead, the function return value for comp_resc() is zero if the compare was successful and the real returns are set in four global variables: comp_resc_gt, comp_resc_eq, comp_resc_lt, and comp_resc_nc. Or, the function return

value is -1, meaning there was a bad attribute pointer. Each of the three global variables variables contains the number of resources in the list headed by *pattr* which compared greater than, equal to, or less than the corresponding resource in the list headed by *with*. Please note the comparison relationship is

(resources in *pattr*) **OP** (resources in *with*)

and only those resources in *with* that are *set* and are not set to a *default* value are checked; if there are resources in *pattr* not in *with*, they are ignored. If there are resources in *with* not in *pattr*, the global comp_resc_nc (not compared) is incremented. The caller may decided if this is an error situation.

The global counts are useful in checking the resource requirements of a job against the resource limits of a queue, see *svr_chkque()*.

For each resource in the list headed by *with*, comp_resc finds the corresponding resource (one that has a pointer to the same resource_def structure) in *attr*.

If the corresponding resource is not set to a value, or set to a queue or system default value, comp_resc_nc is incremented. This is teated as an unlimited amount. This allows an administrator to set up queues without initializing the resources she doesn't care about.

When the corresponding resource is set, the rs_comp() function is called with the two resource entries. If the comparison of the resource from the list *pattr* with the resource from the list *with* is +1 (greater than) then, comp_resc_gt is incremented; if -1 (less than), then comp_resc_lt is incremented; otherwise (equal) comp_resc_eq is incremented. The comparison continues with the next resource in *with*. When completed, return zero as a success flag.

free_resc()

```
void free_resc(attribute *pattr)
```

Each entry in the resource list headed by the attribute is unlinked and the space is freed.

find_resc_def()

```
resource_def *find_resc_def(resource_def *rscdf, char *name, int limit)
```

Args:

rscdf Pointer to the start of the resource definition array.

nameThe name of the resource.

limit The number of resources defined in the array pointed to by rscdf.

Returns:

Pointer
to the resource definition structure matching the specified name.

This function walks the resource definition array till it finds a name match. A pointer to that entry is returned. If a match is not found, a NULL pointer is returned.

find_resc_entry()

```
static resource *find_resc_entry(attribute *pattr, resource_def *rscdf)
```

Args:

pattr A pointer to the attribute which heads a linked list of resource values.

rscdf A pointer to a resource definition.

Returns:

Pointer
to resource entry or NULL if one not found.

This function walks the resource entry list until a entry is found that points to the specified resource definition. A pointer to that entry is returned. If one is not found, a NULL pointer if returned.

add_resource_entry()

```
resource *add_resource_entry (attribute *pattr, resource_def *prdef)
```

Args:

pattr Pointer to the attribute which heads the resource entry list.

prdef Pointer to a resource definition structure.

Returns:

Pointer
to the newly added entry, or NULL on error.

This function inserts a new entry into a resource entry list headed by the attribute pointed to by pattr. The new entry type is given by the resource definition structure pointed to by prdef. The new entry is placed in the list alphabetically by resource name. This is just a convenience for listing the resources.

The existing list is walked until an entry is found that comes after the new entry or till there are no more entries. Should an entry with the name already exist, it is returned. Memory for the new entry is allocated, and the entry is inserted into the list via a call to insert_link(). The type field in the entry is set to that of the definition, and the "value is unset" flag is marked on. A pointer to the new entry is returned.

action_resc()

```
int action_resc(attribute *pattr, void *object, int actmode)
```

Args:

pattr pointer to resource_list attribute for a job.

objectpointer to parent object (job), not used.

actmode
the type of action.

Returns:

zero on all cases.

This is the *at_action()* routine for the job attribute resource_list. It is called whenever the attribute is modified. It will check each resource in the attribute and if that resource has been modified, {ATR_VFLAG_MODIFY} is set, and has an action routine declared, the action routine is called. Note, when calling the resource action routine, the object pointer, second argument, points to the original attribute, not the job. {ATR_VFLAG_MODIFY} is cleared. This prevents additional calls to the individual action routines on the next cycle through the resources when another resource is modified.

Recommended reading, please take a look at *set_node_ct()* in rc/server/svr_resc_def_sp2.c.

10.1.2.10. attr_fn_size.c

The file *src/lib/Libattr/attr_fn_size.c* contains the manipulation functions for attributes of type "size". A size value is an integer with a one or two character optional suffix. The first character is k, m, g, t, or p for kilo (1024), mega, giga, tera, and penta. Upper case letters may also be used. The second character character is either b or B for bytes; or w or W for words, the size of an integer.

The value is contained with in the attribute in two fields defined in attribute.h by *struct size_value* . The first field is an unsigned long holding the specified integer. The second field is the scaling factor, actually a shift count, to the size suffix. A k is represented by 10 (2^10), m by 20, etc.

The use of bytes is assumed unless w or W was specified. The {ATR_VFLAG_WORDSZ} is set in at_flags.

decode_size()

```
int decode_size(attribute *pattr, char *name, char *rescn, char *val)
```

The value string is passed to the function *to_size()* which decodes the numeric part into at_val.at_size.atsv_num, and process the suffix for the setting of at_val.at_size.atsv_shift and at_val.at_size.at_sv_units.

encode_size()

```
int encode_size(attribute *pattr, list_head *phead, char *atname,
       char *rsname, int mode)
```

The function *from_size()* is called to translate from the at_val.at_size structure to a string. The function *attrlist_create()* is called to create a svrattrl entry containing the attribute name. The encoded string is copied into the entry.

set_size()

```
int set_size(attribute *old, attribute *new, enum set_op op)
```

The value of the old attribute is set with the new value based on the operation type. Both values is save in a temporary attribute and the operations are performed on them. If there are any errors, the original value is undisturbed.

Set Temporary is set to the numeric and shift value of the new.

Incr The values are normalized to the smaller of the shift counts by calling *normalize_size()*. Temporary is set to the sum of the temporary and new values.. If the resulting value overflowed, error [PBSE_BADATVAL] is returned.

Decr The values are normalized to the smaller of the shift counts by calling *normalize_size()*. Temporary is set to the difference of the temporary and new values.. If the resulting value underflowed, error [PBSE_BADATVAL] is returned.

If there were no errors, the old attribute is set equal to the temporary attribute.

comp_size()

```
int comp_size(attribute *attr, attribute *with)
```

This function compares numerically, the values of the attribute *attr* and *with*. The two attribute values are normalized by calling *normalize_size()*. If they cannot be normalized, then they are not equal and the return is based on the shift size alone. Otherwise, the comparison is based on the normalized number part.

normalize_size()

```
int normalize_size(struct size_value *a, struct size_value *b,
        struct size_value *c, struct size_value *d)
```

Args:

a pointer to an existing size_value structure.

b pointer to an existing size_value structure.

c pointer to a size_value structure, the structure is updated.

d pointer to a size_value structure, the structure is updated.

Returns:

0 if successful.

!=
if error.

cthe size_value structure pointed to by c is set to the normalized value of a.

d
the size_value structure pointed to by c is set to the normalized value of b.

The data from the two existing size_value structures are "normalized" to each other. This allows the values to be added, subtracted or compared.

The size_value structures are copied from a to c and from b to d. If one but not both of c and d are in byte, not word units, the size of the one not in bytes is multiplied by the word size. If the shift counts in a and b are different, the value in the one with the larger shift is shifted by the difference and the sift counts are set equal to the smaller count.

to_size()

```
int to_size(char *value_string, struct size_value *size)
```

Args:

value_string
the text string specifying a size value.

size pointer to a size_value structure, the structure is updated.

Returns:

0 if success.

!=
if error.

The numeric part of the string is converted to a long by *strtol()*. The suffix letters are used to set the shift count and the word unit flag. Any error results in a return value of [PBSE_BADATVAL].

from_size()

```
void from_size(struct size_value *size, char *cvnbuf)
```

Args:

size pointer to size_value structure to convert to a string.

cvnbuf
point to a buffer in which the string is created.

The numeric part of the size, atsv_num, is converted to an numeric string by *sprintf()*. The shift factor is converted back to the corresponding letter and concatenated to the numeric string. If the units are {ATR_SV_WORDSZ}, then 'w' is appended, otherwise 'b' is appended to the string.

10.1.2.11. attr_fn_str.c

The file *src/lib/Libattr/attr_fn_str.c* contains the manipulation functions for attributes of type "string". The value string is contained in space dynamically allocated and pointed to by the attribute.

decode_str()

```
int decode_str(attribute *pattr, char *name, char *rescn, char *val)
```

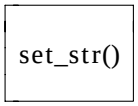
decode_str takes a string and returns it as a string! Space is allocated for the value and the string is copied into it.

encode_str()

```
int encode_str(attribute *pattr, list_head *phead, char *atname,
```

```
char *rsname, int mode)
```

The function *attrlist_create()* is called to create a svrattrl entry containing the attribute name. The value string is copied into the entry.



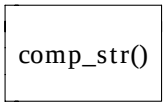

```
int set_str(attribute *old, attribute *new, enum set_op op)
```

The value of the old attribute is set with the new value based on the operation type. For the Set and Incr operations, the space occupied by the old string is freed and new space is allocated.

Set Old is replaced by the new.

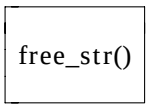
Incr Old is set to the concatenation of the old and new strings.

Decr If old has a substring that matches new, then it is truncated. The search is from the tail end, so if multiple matching substrings exist, the last is the one removed.




```
int comp_str(attribute *attr, attribute *with)
```

comp_str compares the strings values of the attribute and "with", the standard function *strcmp()* is used and its return value is passed back.

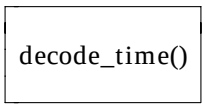



```
void free_str(attribute *pattr)
```

Frees the space allocated to hold the attribute value string.

10.1.2.12. attr_fn_time.c

The file *src/lib/Libattr/attr_fn_time.c* contains the encode and decode functions for attributes of time. Note, this is an interval time, not the date. The time is maintained internally as a long integer number of seconds. For the time set and compare functions, set_l() and comp_l() are used.




```
int decode_time(attribute *pattr, char *name, char *rescn, char *val)
```

The input string is assumed to be in the format `[[hh:]mm:]ss[.sss]`. The string is converted to the corresponding number of seconds.

encode_time()

```
int encode_unkn(attribute *pattr, list_head *phead, char *atname,
    char *rsname, int mode)
```

The interval time value contained in the attribute is converted to a string of the format `hh:mm:ss[.sss]`. The hours, minutes, and seconds are printed as two digits. Zero values are printed as "00".

The file *src/lib/Libattr/attr_fn_unkn.c* contains the manipulation functions for attributes of miscellaneous or unknown name and type. These are the attribute whose name was not directly recognized by the server. It is assumed that they have meaning to the Scheduler or some other server.

These attributes are treated slightly different from most other types in that they are maintained as a linked list of structure *svrattrl*. In other words, the decoded form IS the encoded form.

decode_unkn()

```
int decode_unkn(attribute *pattr, char *name, char *rescn, char *val)
```

It is unfortunate, but **decode_unkn() must be called slightly differently** than the other decode routines. Because decode_unkn must know the attribute name as well as the value, the parameter *name* is required. Since it must be present for one, it is present for all the decode routines.

An entry essentially duplicating the svrattrlst entry containing the triplet name, rescn, and val is made and linked to the attribute pointed to by *pattr*.

encode_unkn()

```
int encode_unkn(attribute *pattr, list_head *phead, char *atname,
    char *rsname, int mode)
```

This encode routine is a bit different from most. It iterates through the linked list of svrattrl entries, and duplicate each one and links the copy into the list headed by *phead*.

set_unkn()

```
int set_unkn(attribute *old, attribute *new, enum set_op op)
```

Because the local server does not recognize the attribute and does not know if duplicates are legal, the "set operation for unknown attributes maps to an append operation. Each of the entries in the *new* attribute are appended to the list of entries in the *old* attribute.

comp_unkn()

```
int comp_unkn(attribute *pattr, attribute *with)
```

The comparison routine is a fake. Again because the local server does not recognize the attributes, it will always return +1. At the time of this writing, the author cannot think of a case where comp_unkn would be used.

free_unkn()

```
void free_unkn(attribute *pattr)
```

Each entry in the attrlist headed by the attribute is unlinked and the space is freed.

10.1.2.13. attr_node_func.c

The routines in file *src/lib/Libattr/attr_node_func.c* are used when carrying out batch requests that involve the modification or checking of status of a pbsnode. Even though pbsnodes were not defined to have attributes, certain modifications of the data in a pbsnode makes use of the data structure type *attribute* to accomplish the modifications. The approach taken directly mimics that which was established for modification of object attributes. Each time a pbsnode is to be modified a temporary array of node-attributes is generated for the pbsnode. This array of attributes is then modified atomically just as real attributes of an object would be modified. After the modification is finished, the temporary array of node-attributes is then used to update the various fields in the subject pbsnode.

Following along the same vein as the section titled Attribute Manipulation Functions the prototypes and returns for the set of node-attribute manipulation functions is presented now rather than with each specific function, since one prototype/return applys to every function callable through any particular function pointer (at_encode, at_decode, at_free, etc) of an *attribute_def* data structure. This saves a lot of repetition of prototypes and returns.

at_encode

Args:

- pattr points to struct attribute data structure being encoded
- ph points to head of a list of *svrattrl* structs which are to be returned
- aname points to the node-attribute's name
- rname points to the resource's name (unused)
- mode mode code, unused

Returns:

- < 0 value is the negative of an error code
- 0 encoding occurred successfully

at_decode

Args:

- pattr points to an attribute data structure into which the external form of attribute data is decoded
- name attribute name

rescn resource name, unused in this application

val attribute value

Returns:

> 0 error occurred, return an error code

0 decode occurred successfully

at_set

Args:

pattr points to struct attribute data structure to get modified

new temporary attribute, holds the decoded, modifying information

op integer code, specifies a modification operation (INC, DEC, etc)

Returns:

> 0 error occurred, return an error code

0 set opeation occurred successfully

at_action

Args:

new either transform pbsnode information into this node-attribute or use this node-attribute to modify the pbsnode's internal data

pnode pointer to the subject pbsnode structure

actmode action mode,

Returns:

> 0 error occurred, return an error code

0 node-attribute initialization or pbsnode modification occurred successfully

at_free

Args:

pattr pointer to the node-attribute

encode_state()

```
int encode_state (attribute *pattr, list_head *ph, char *aname, char *rname, int mode)
```

Once the pbsnode's inuse field datum is placed into an attribute, the attribute is passed via an indirect function call (at_encode) to this function where it is encoded into an svrattrl structure, which is then linked on to the list in the batch reply structure (a substructure within the batch request structure).

encode_ntype()

```
int encode_ntype (attribute *pattr, list_head *ph, char *aname, char *rname, int mode)
```

Once the pbsnode's ntype field datum is placed into an attribute, the attribute is passed via an indirect function call (at_encode) to this function where it is encoded into an svrattrl struc-

ture, which is then linked on to the list in the batch reply structure (a substructure within the batch request structure).

encode_properties()

```
int encode_properties (attribute *pattr, list_head *ph, char *aname, char *rname, int mode
```

Once the pbsnode's *properties list* is placed into a prop attribute (one whose at_val member is a pointer to a struct prop), the attribute is passed via an indirect function call (at_encode) to this function where it is encoded into an svrattrl structure. The structure gets linked on to the list in the batch reply structure (a substructure within the batch request structure).

encode_jobs()

```
int encode_jobs (attribute *pattr, list_head *ph, char *aname, char *rname, int mode)
```

Once the pbsnode's struct jobinfo pointer is placed into a temporary attribute, this function gets indirectly called (at_encode) to walk the list of jobs at the node and produce a comma separated job list for sending back to the requester via an svrattrl structure.

decode_state()

```
int  decode_state (attribute *pattr, char *name, char *rescn, char *val)
```

For this particular function (indirect via at_decode) the two arguments that get used are *pattr*, which points to an attribute whose value is a short, and the argument *val*, the value for the attribute. The value argument, val, is decoded from its form as a string of comma separated substrings and the component values are used to set the appropriate bits in the attribute's value field.

decode_ntype()

```
int  decode_ntype (attribute *pattr, char *name, char *rescn, char *val)
```

For this particular function (indirect via at_decode) the two arguments that get used are *pattr*, which points to an attribute whose value is a short, and the argument *val*, the value of the attribute. At this point in PBS's evolution the two values time-shared and cluster are the only possible values. The one thing that is assumed is that the types are all going to be mutually exclusive.

decode_props()

```
int  decode_props (attribute *pattr, char *name, char *rescn, char *val)
```

For this particular function (indirect via at_decode) the two arguments that get used are *pattr*, which points to an attribute whose value is a short, and the argument *val*, the value for the attribute. Val is a string of comma separated substrings. Once val's components are decoded into a linked list of "prop" structures, this list is hung from the *pattr* argument's *at_val* field.

set_node_state()

```
int  set_node_state (attribute *pattr, attribute *new, enum batch_op op)
```

The information in the *short* attribute, **new*, is used to update the information in the short attribute, **pattr*. The mode of the update is governed by the argument *op* (SET,INCR,DECR). The call is indirect via at_set.

set_node_ntype()

```
int  set_node_ntype (attribute *pattr, attribute *new, enum batch_op op)
```

The value field in attribute **new* is a short. It's generated by the decode routine and used to update the value portion of the attribute **pattr*, the mode of the update is goverened by the argument *op* (SET,INCR,DECR). The call is indirect via at_set.

set_node_props()

```
int  set_node_props (attribute *pattr, attribute *new, enum batch_op op)
```

The information in the props attribute pointed to by *new* is used to update the information in the props attribute pointed to by *pattr*. the mode of the update is goverened by the operations argument *op*, (SET,INCR,DECR). The call is indirect via at_set.

node_state()

```
int  node_state (attribute *new, void *pnode, int actmode)
```

Either derive a state attribute *new* from the pbsnode pointed to by argument *pnode* or update the pbsnode's *inuse* bit field using the state attribute, **new*. The choice of which action to perform is determined by the action mode argument, *actmode* (ATR_ACTION_NEW, ATR_ACTION_ALTER). The call is indirect via at_action.

node_ntype()

```
int  node_ntype (attribute *new, void *pnode, int actmode)
```

Either derive an "ntype" attribute *newnode* from the pbsnode pointed to by argument *pnode* or update a pbsnode's *ntype* field using the ntype attribute, new. The choice of which action to perform is determined by the action mode argument, *actmode* (ATR_ACTION_NEW, ATR_ACTION_ALTER). The call is indirect via at_action.

node_prop_list()

```
int  node_prop_list (attribute *new, void *pnode, int actmode)
```

Either derive a prop list attribute from the pbsnode or update the pbsnode's prop list from the attribute's prop list. The choice of which action to perform is determined by the action mode argument, *actmode* (ATR_ACTION_NEW, ATR_ACTION_ALTER). The call is indirect via at_action.

free_prop_attr()

```
void  free_prop_attr (attribute *pattr)
```

This function calls *free_prop_list()* to remove the null terminated prop list pointed to by *pattr->at_val.at_prop*. As part of the "freeing" operation the attribute's "VALUE is SET" flag gets zeroed and the attribute's value (a struct prop*) is set to 0. The call is indirect via at_free.

free_prop_list()

```
void  free_prop_list (struct prop *pattr)
```

Args:

pattr points to the head of a list of struct prop's that need to be freed

This function walks a null terminated prop list and for each struct prop on the list it frees any string buffer space hanging from it before freeing the space belonging to the struct prop itself.

load_prop()

```
static int  load_prop (char *val, struct prop *pp)
```

Args:

val pointer to the string value for the prop structure

prop pointer to the prop structure in to which to copy the string

This function mallocs buffer space to hold the string pointed to by argument, *val*. It copies the string into this new space and hangs it on the prop struct pointed to by the argument, *pp*.

set_nodeflag()

```
static int  set_nodeflag (char *str, short *pflag)
```

Args:

str points to a state value in string form

pflag pointer to a "bit flags" variable for *state*

Use the value of the input string to set a bit in the bit flags variable pointed to by argument *pflags*. Each call will set one more bit in the flags variable or it will clear the flags variable in the special case where **str* is the value free.

10.1.2.14. attr_atomic.c

The file *src/lib/Libattr/attr_atomic.c* contains routines for performing part of an atomic update of a list of attributes. An atomic update is one where all the updates or changes to an objects attributes are successful, or none are made.

In general, the steps required to perform an atomic update are:

1. Decode the new values, stop if any errors.
2. Duplicate the current values by calling the at_set() function. Only those values which are to be changed, must be duplicated.
3. Update the old value with the new value. This may be done either in the actual old attribute or the copy. If the actual attribute is modified and later error occur, then the actual attribute must be restored from the saved copy. Or the copy can be updated, and when all are successful, then the actual attribute must be replaced by the modified copy.

   The choice of methods depends partly on how expected are errors to occur. If errors are expected, update the copy, otherwise update the original.
4. If the original attributes were updated, free the copies. If the copies were updated, clear the copies but make sure not to free any additional space allocated to a copy, do not call at_free(); the original now points to the same space.

attr_atomic_set()

```
int attr_atomic_set(svrattrl *plist, attribute *old,
        attribute *new, attribute_def *pdef, int limit,
        int unkn, int privil, int *badattr)
```

Args: And a bunch of them there are...

plist pointer to list of new attribute values in the svrattrl form.

old pointer to original attribute array which is to be update.

new pointer to attribute array to be used for copies.

limit number of attributes in the new and old arrays.

unkn attribute index of unknown type attributes if allowed, >= 0. If unknown attributes are not allowed, then this should be less than zero.

privil the privilege level of the client, based on read/write flags in the attribute.

badattr
(RETURN) pointer to an integer in which the number of a bad attribute in the svrattrl is detected.

Returns:

0 is successful.

non zero
error number if an error is detected. The ordinal, starting with 1, of the svrattrl entry is placed in *badattr.

This function performs step 1, 2, and 3 described above. The approach taken in step 3 is the "update the copy" approach. The following is additional useful information: the {ATR_VFLAG_MODIFY} flag is set on all attributes which changed. If the new value is null, effectively an "unset operation", then the produced copy will have the {ATR_VFLAG_MODIFY} flag set but not the {ATR_VFLAG_SET}. Thus it is possible to tell what happened.

Step 4 from above is not included to allow the caller to perform additional checks before doing the actual modification of the attributes. For example, when updating the attributes of a job, there are additional checks to be make depending of the state of the job. These checks do not apply to other objects.

If the calling routine chooses to complete the update, it should:

- Free the current **old** attribute value by calling at_free().
- Replace each element of **old** with the corresponding element of **new** if new has been changed, {ATR_VFLAG_MODIFY} is on. The calling routine should then free the storage assigned to the **new** attributes without calling *at_free()* on the members of new as the same data space is now used by the **current** attributes.

If the calling routine decides not to complete the update, it should completely release **new** by calling *at_free()* on each element and then freeing the array space itself, see *attr_atomic_kill()*.

There is a special test at the beginning where the attribute is identified by calling *find_attribute()*. For the attributes of the server, queues, and jobs in execution queues, unknown attributes are not accepted. The error [PBSE_NOATTR] is returned. However, for jobs in routing queues, it is legal to have unknown attributes and to alter them. Therefore, if the parameter unkn is positive, it is the index of the "unknown" attribute list to which to append the new value.

attr_atomic_node_set()

int attr_atomic_node_set
(svrattrl *plist, attribute *old, attribute *new, attribute_def *pdef,
int limit, int unkn, int privil, int *badattr)

Args:

plist pointer to list of new attribute values in the svrattrl form.

old this argument is not currently used

new	pointer to the node-attribute array
limit	number of elements in the node-attribute definitions array
unkn	if <0 then unknown node-attributes are not permitted
privil	requester's access privilege
badattr	if encounter a bad node-attribute in the batch request, place its list position here

Returns:

0	if successful.
non-zero	error code if an error is detected (if bad node-attribute, put list position in *badattr)

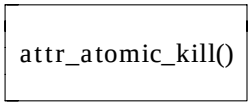
This function atomically updates a node-attribute array with values from a batch request's list of svrattrl structs. If the updating is successful for all the node-attributes in the batch request, the function returns success (0). Otherwise, a non-zero return code is passed back up the call chain and, if the error was due to a bad node-attribute the position of this node-attribute in the request list is passed back via the pointer, badattr.

The sequence of steps involved in the processing is:
For each node-attribute in the request list, call *find_attr()* to determine if the requested node-attribute is in the definitions array. If it isn't in the definitions array, use the pointer badattr to record at what position in the list the error occurred and return back up the call chain with the error code [PBSE_NOATTR.]

At this point the requested node-attribute is defined, now check the definition's indicated privilege against the privilege level of the requester. Return back up the call chain with the error code [PBSE_ATTRRO] if the requester doesn't have sufficient privilege.

Next, since the node-attribute exists in the definitions array and enough privilege exists, decode the data in the request into a temporary node-attribute by invoking, for this particular node attribute, the *at_decode* function from the definitions array. Should an error occur in the decoding, use baddattr to pass back where in the request list it occurred and return back up the call chain with the error code from the decode function.

Now setup to use the data decoded into the temporary node-attribute. If the request has not specified the modification operation for this node-attribute, the operation is taken to be SET, the node-attribute's modification flag is turned off and the node-attribute's *at_set* function from the definition is invoked to perform the modification to the node-attribute. Again, if an error occurs badattr is used to pass back the position where the errror was encountered and the error code from the at_set function is returned back up the call chain to determine the reply to the requestor. Otherwise, if there is another node-attribute in the batch request's svrattrl list repeat the process else return success (0).




```
void attr_atomic_kill(attribute temp, attribute_def *pdef, int limit)
```

Args:

temp	pointer to a temporary array of attributes.
pdef	pointer to attribute definitions.
limit	number of attributes in the array.

This routine assumes that *temp* is a temporary array of attributes, space for which must have been malloc-ed and which should have been initialized by *attr_atomic_set()*. The appro-

priate at_free routine is called on each element of the array and then the whole array is freed by a call to free().

10.1.3. Long Long Integer Attribute Support

This part of the library is intended to make integers larger than "int" available in a portable fashion. Unfortunately, far too many C vendors have wimped out and declared long and int to be the same size (32 bits), rendering long useless. This package is intended to overcome that bit of vendor treachery by inventing two new data types that always designate the largest integer size supported by the compiler. They are:

```
typedef ?               Long;           /* largest signed integer type */
typedef unsigned ?      u_Long;         /* largest unsigned integer type */
```

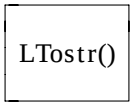
Fortunately, these can be defined to be integer data types larger than int on all platforms to which we have ported PBS and to which we plan to port PBS. In the case of SunOS, that statement is only true using the GCC compiler instead of the native compiler as the native cc does not support long long integers. In all cases except HP-UX, Long and u_Long represent 64-bit integers.

For the most part, this is an include file exercise. The appropriate include file

```
#include "Long.h"
```

contains a good deal more documentation in its comments. There are also five functions associated with the package (four real and one macro).

10.1.3.1. LTostr.c




```
const char *LTostr(Long value, int base);
```

Args:

valueto convert to a string

base to use in conversion

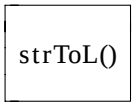
Return:

pointer to string

Convert a signed *Long* to a string. This provides the complementary capability to strToL(). It converts value into a NULL terminated digit string in the base, *base*. The pointer to the string that it returns designates a location in static storage, so it must be copied if it is to survive another call to LTostr() or a call to uLTostr().

Static storage was used to make the function easier to use as an argument to printf(). No free() is required.

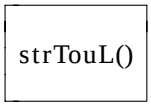
10.1.3.2. strToL.c




```
Long strToL(const char *nptr, char **endptr, int base);
```

Convert string to *Long* integer type. This is directly analogous to POSIX strtol(), in every way.

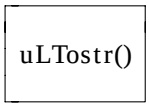
10.1.3.3. strTouL.c




```
Long strTouL(const char *nptr, char **endptr, int base);
```

Convert string to unsigned *Long* integer type. This is directly analogous to POSIX strtoul(), in every way.

10.1.3.4. uLTostr.c




```
const char *uLTostr(u_Long value, int base);
```

Convert unsigned *Long* to a string. It is analogous to LTostr().

10.1.3.5. Long_.c

Provides constant data storage.

10.1.3.6. Long.h


```
Long atoL(char *nptr);
```

This is a macro defined as `strToL((nptr), (char **)NULL, 10)`. It is directly analogous to POSIX atol(), in every way.

10.2. Library: libcred.a - Credential Library libcred.a

The credential library, *libcred.a* , consists of routines to create an encrypted client authentication credential and to decrypt the credential, or ticket, into a credential structure. Together with the **pbs_iff** program, this library provides the basic user/client authentication required by PBS. This system is explained in the PBS ERS and in the IDS section on iff. The routines in this library are used by pbs_iff, and various server programs.

The basic PBS credential or ticket contains the following items:

1. The user's name and the fully qualified host name encrypted together with a key generated by pbs_iff.
2. The above together with a time stamp encrypted with a key generated by the server, and the pbs_iff key itself.

The user name must match that in the batch request, the hostname must match that provided by the network routines, and the time stamp must be within its life time. The structure of

the credential and its life time are defined in credential.h.

Should a site or vendor wish to replace the authentication system with a more general system, such as Kerberos, this library is the place to start.

NOTE:
The routines in this library are set up to call DES encryption routines. Because of restrictions on exporting DES out of the USA, these routines cannot be included in the PBS distribution. The DES routines used by PBS are identical to the MIT Athena routines and other common packages. The source for these routines may be found at many foreign ftp sites.

break_credent()

```
struct credential *break_credent(char *key_string, char *ticket, int size)
```

Args:

svr_key
is an (8 character) key used by the server as the ticket encryption key.

ticketis the full ticket as produced by pbs_iff.

size is the size of the full ticket string.

Returns:

pointer
to a struct credential containing the decrypted information.

This routine, found in file break_credent.c is used by the server to decrypt the full ticket into its parts and build the credential structure.

The key created by pbs_iff and included in the ticket is saved. The provided server key is used to decrypt the sealed portion of the ticket. This yields the time stamp and the sub-credential encrypted by pbs_iff. The four bytes of time are converted into at time_t type and placed in the credential.

The sub-credential is decrypted using the saved pbs_iff key. The user name and host name are copied into the credential. A pointer to the credential is returned.

get_credent()

```
int get_credent(char *server, char **credential)
```

Args:

serverthe full name of the server to contact.

credential RETURN: the address of the credential is returned, ticket spaced is malloc-ed.

Returns:

>=0 positive size of credential.

<0 if error.

This routine, see file get_credent.c is used by the pbs_ifl library to obtain a PBS_IFF user credential to include in a batch request. When the user client program calls *pbs_connect()*, it calls this function to fork and exec the *pbs_iff* program to build the credential. This is done

via a *popen(3)* call invoking pbs_iff with the name of the server and port, as used by pbs_connect(). The credential, or full ticket, is read from the pipe which is closed. If the size of the received credential is incorrect, or the exit status of pbs_iff is not zero, a -1 is returned as an error indicator.

Note, the space for the credential is malloc-ed. It should be freed by the client when no longer required.

make_sealed()

```
int make_sealed(char svrkey[PBS_KEY_SIZE], char *subcred, int size,
    char **sealed)
```

Args:

svrkey
: an (8 character) key that is the server encryption key.

subcred
: the sub-credential passed from pbs_iff.

size
: of the sub-credential.

sealedRETURN: a pointer to the sealed (encrypted) ticket, in static space.

Returns:

: the size of the encrypted ticket is returned as the function value. If an error occurs, -1 is returned.

This routine, see file make_sealed.c is used by the server to produce the sealed ticket portion of the credential. This portion consists of the already encrypted user and host names and a time stamp. These items are again encrypted using the server's key.

The sub-credential, the user and host name encrypted by pbs_iff and sent to the server, is checked to insure its size is correct.

The time is obtained and converted from a time_t (which according to POSIX just might be a real number) to a long integer. Only the least significant 4 bytes of the the integer time is included in the ticket to allow for different word sizes.

The information is encrypted into a string, which is a multiple of 8 bytes. A pointer is returned in ticket and the length of the string is returned as the function value.

make_svr_key()

```
void make_svr_key(char key[PBS_KEY_SIZE])
```

Args:

key
: RETURN: the generated key

This function, also in file make_sealed.c, is called by the server to generate an encryption key. The function does nothing but call the des library routine *des_random_key()*. This function just services to isolate the server from the des library and header files.

make_subcred()

```
int make_subcred(des_cblock key, char *user, char **subcred)
```

Args:

key an encryption key provided by the caller.

user the name of the user to include in the credential.

subcred
RETURN: a pointer to the encrypted string is placed into subcred.

Returns:

the size of the encrypted string is returned as the function value. If an error occurs, -1 is returned.

This routine is used by **pbs_iff** to produce the sub-credential to be sent to a server. At the server, the sub-credential is included in the sealed-ticket which is returned to pbs_iff.

The full host name is obtained by calling *get_fullhostname()*. This process is performed to make sure we get the fully qualified name when the system administrator may have only set the local name (without the domain name).

The information is encrypted using the supplied key into a string, which is a multiple of 8 bytes. A pointer is returned as in subcred and the length of the string is returned as the function value.

10.3. Library: liblog.a - Log Record Library liblog.a

The log record library, *liblog.a* , consists of routines to record a series of events and errors in a log file. the use of this library will provide a consistent format for the records in the log file. Each log file entry is one line of text, terminated with a new line. Each line is made up of several fields, with a semicolon, ';', between fields. The fields are:

date time
The date and time the entry was added to the log. The date is in month/day/year format. The time is in 24 format with hours, minutes, and seconds in the format hh:mm:ss.

Event Type
This field is a hexadecimal number, where each "1" bit identifies the type of event recorded, see log.h.

Server Name
This names the server which recorded the entry. While only a single server records in each file, this field is provided to allow a site to merge the files together for processing.

Object Class
This field identifies the object class affected by the event. Current classes are:
Fil - A job related file
Req - a batch request
Job - a batch job
Que - a queue
Svr - the server

Object Name
The name of the object affected by the event.

Text This field contains the text of the message.

The server lib is divided into two parts, with each part in its own source file. Part one, in file *pbs_log.c*, is code which is independent of the main PBS Server. Part two, in file *log_event.c*, either requires knowledge of the server attributes or is only useful to the server.

10.3.1. File: pbs_log.c

The file *src/lib/Liblog/pbs_log.c* contains the functions to record information to a log file. Either "event" records or error records are recorded. The necessary support functions to open and close log files are also included. The file descriptor and open status for the log file are maintained within the scope of this file.

log_open()

```
int log_open(char *filename, char *directory)
```

Args:

filename
: Name of log file to open. Null to used default name based on the date.

directory
: Name where default log files are maintained.

Returns:

0 On success.

-1 If Error.

The function attempts to open the file specified by *filename* in append mode. If it cannot be opened or it is the null pointer or null string, an error is returned. When opened, the stream is set for no buffering to minimize message lost on a crash of the server. A message is written in the new log file to record the time at which it was opened.

The file descriptor for the opened log file is maintained in a variable whose scope is limited to the file containing the log routines. The julian date when the log file was opened is also maintained as a static variable, see log_record().

log_err()

```
void log_err(int err, char *routine,  char *text)
```

Args:

err The system error number as found in *errno* if it applies. A value of -1 indicates PBS found an error other than on a system call.

routine
: The name of the routine calling log_err.

text The message text to record.

This function is used to record internal errors. The error is recorded in the log file and, if configured, sent to *syslog*.

If the log file is not open when log_err() is called, it will either use the *syslog* facility, if configured, or write to `/dev/console`.

log_record()

```
void log_record(int type, int class, char *name, char *text)
```

Args:

type The type of event. Used to determine if this event is recorded.

class The class of the object: {PBS_EVENTCLASS_SERVER}, {PBS_EVENTCLASS_QUEUE}, {PBS_EVENTCLASS_JOB}... affected by the event.

nameThe name of the effected object.

text The text to record as part of the log entry.

This function is used to record normal events regardless of the event type, see *log_event()* below. When log_record() is called and the current julian date is not the same as the one recorded by log_open(), and if the default log name is being used (the date), then the log is closed and reopened under the new current date.

If an error occurs on the write, the current FILE pointer is saved and the log's pointer is replaced with one pointing to /dev/console. *log_error()* is called to display an error message to the effect that logging is not working.

log_close()

```
void log_close(int msg_flag)
```

Args:

msg_flag
if non-zero, log the "log closed" message.

Closes the currently opened log.

10.3.2. File: log_event.c

The file *src/lib/Liblog/log_event.c* contains log related code that is specific to logging only certain types of events.

log_event()

```
void log_event(int type, int class, char *name, char *text)
```

Args (identical to log_record):

type The type of event. Used to determine if this event is recorded.

class The class of the object: {PBS_EVENTCLASS_SERVER}, {PBS_EVENTCLASS_QUEUE}, {PBS_EVENTCLASS_JOB}... affected by the event.

nameThe name of the effected object.

text The text to record as part of the log entry.

Global variables:

log_event_mask
is a pointer to long type. The value to which it points is a mask which is checked

to determine if the event type is being recorded.

If {PBSEVENT_FORCE} is set in the argument type, the event type will be recorded regardless of the `*log_event_mask` mask value. Otherwise, the type arguement is and-ed with value pointed to by log_event_mask. If the result is non-zero, the event is included in the types to be logged. If the event is to be recorded, the arguments are passed on to *log_record()*. The initial setting of the logging mask includes logging error, system, admin, job, and security events.

log_change()

```
int log_change(attribute *pattr, void *parent, int actmode)
```

Args:

pattr pointer to the server attribute log-file.

parent
Unused.

actmode
Unused.

Returns:

Returns the return value of log_open().

This is the "at_action" function for the server attribute log-file. It is invoked whenever the attribute value is changed. All the function does is call *log_open()* with the log file name.

10.3.3. File: chk_file_sec.c

The file *src/lib/Liblog/chk_file_sec.c* contains the security validation routine chk_file_sec(). This function is in liblog.a because it is a handy library that all the daemon include.

chk_file_sec()

```
int chk_file_sec(char *path, int isdir, int sticky, int disallow, int full)
```

Args:

path The full path name of the file to be checked.

isdir set non-zero if the path should be the name of a directory, zero for a file.

stickyset non-zero if group/other write is allowable on a directory if and only if the sticky bit is set on the directory.

disallow
File mode bits (see sys/stat.h) that should not be allowed on this file.

full if non-zero, check the full path, i.e. all parent directories. Generally set zero only when the parents have already been checked.

Returns:

Zero if path is secure, otherwise a non-zero error code indicating the problem.

This function is used by each of the daemons and by the supplied utility *chk_tree* to perform a security check on critical daemon files. Generally, these files should not be writable to any one other than root. The routine calls itself recursively to check the parent directives. The

check occurs from the root directory downward in case of symbolic links.

10.3.4. File: setup_env.c

The file *src/lib/Liblog/setup_env.c* contains the security routine setup_env(). This function is in liblog.a because it is a handy library that all the daemon include.

setup_env()

```
int setup_env(char *filename)
```

Args:

filename
the name of the environment definition file.

Returns:

Non-zero on any error.

The purpose of this routine is to insure a "secure" environment for the daemons. This prevents an attacker from using the environment to impact the actions of the daemons or any programs run by the daemons.

If the filename is null or the null string, the routine returns without error. Otherwise, the environment will be updated. If the file cannot be read or is empty, the resulting environment is empty.

For each string in the file not starting with '#' or ' ', the string is assumed to be of the form: `name=value` or `name`. If the first form is found that string is placed into the environment. If the second, name only, form is found, and there is an existing environment variable with that name, then the name and that current value are placed into the new environment.

10.3.5. File: svr_messages.c

The file *src/lib/Liblog/pbs_messages.c* contain the text for all messages issued or recorded by various PBS processes. The purpose of placing the messages in one place is to facilitate translation − internationalization.

pbse_to_txt()

```
char *pbse_to_txt(int error)
```

Args:

error a PBS error number.

Returns:

pointer to error message associated with the error, or NULL if none.

This function takes a PBS error number and returns a pointer to the message string associated with the error if one exists.

10.4. Library: libnet.a - Network Library libnet.a

The network library contains functions to support the client / server networking. Most of the TCP/IP dependent activities are packaged here.

10.4.1. File: net_server.c

The file *src/lib/Libnet/net_server.c* contains server side functions to deal with the network. The supplied functions support a *socket* based **TCP/IP** network. The basic services provided are

1. "Bind" to the standard port for PBS service. Prepare to receive (listen) connection requests at those addresses.
2. Wait (select) for requests for connections on the service addresses, or for data on prior accepted connections.
3. Accept (accept) connections.
4. Close connections when all is done.

In addition to the above functions, net_server.c maintains an entry for each active connection identifying the type of connection, the time of last activity on that connection, and the function to call when data is available (to read) on the connection.

init_network()

```
int init_network(unsigned int  port, void (*read_func)())
```

Args:

port The port number to which to bind.

read_func
the function to read data from sockets created by accepting connections on the service port.

Returns:

0 If initialization successfully.

-1 If initialization failed.

Control flow:

If this first time called, initialize connection state table and the set the socket to connection type {Primary}. If this is the second time called, set the socket connection type {Secondary}. If we have already been called twice, return an error. Allocate the socket and bind it to the service port. Note, after the socket is allocated, a call is made to the system function *setsockopt()* with SO_REUSEADDR. Without this call, the server when shut down and brought back up quickly will get an "address already in use" error on the bind. Even worse, a client that leaves a connection open to the server when the server goes down can block the server from starting up. The server would get the above error forever.

Save the read function in a two element array based on first or second call. Note, when data is ready on either the primary or secondary socket, control is passed to the *accept_conn()* routine to accept the connection and allocate a new socket. These sockets are connection type {General}. When data is ready on a general socket, the read routine that was passed on the *init_network()* call is invoked, see accept_conn().

Create the socket and bind it to port number supplied. Add socket to select set and update connection state table. Start listening for connection requests.

wait_request()

```
int wait_request(waittime)
```

Args:

waittime
The maximum time to delay waiting for a request to arrive; the timeout on the select call.

Returns:

0 No errors occurred.

-1 Error on select() call.

Control flow:

Wait for data (select) on set of I/O descriptors.

For each (ready descriptor)
Update time of last activity for connection.
Call function associated with the descriptor (socket) to process data.

For each (active connection)
If (the connection has exceeded it maximum idle time)
Close the connection.

accept_conn()

```
int accept_conn(int sd)
```

Args:

sd Socket with pending connection request

Returns:

0 If no error

-1 If error occurred

This function is called when the *select()* function returns the primary or secondary socket, the one bound to a service port. Data ready on these socket is a request for a connection.

If the maximum number of connections is not exceeded, an *accept()* is performed which returns a new socket, of type General. This socket is added to the connection table. The data function entry for this socket is set to the processing function passed in the init_network call for the parent socket

add_conn()

```
void add_conn(int socket, enum conn_type type, pbs_net_t addr,
      unsigned int port, void (*func)(int))
```

Args:

socketis the socket to be added to the svr_conn array.

type is the connection type, primary, secondary, or general. several new types are possible: ToServerDIS, and FromClientDIS.

addr is the address of the remote host.

port is the port on the remote host.

func is the function to be called to read data from the socket when data is available.

The svr_conn array member, indexed by the socket number, is updated to show its use. The type is always set to {General}.

close_conn()

```
void close_conn(sd)
```

Args:

sd The I/O descriptor (socket) to close.

The descriptor is closed and the connection table is set to "Idle". If there is an auxiliary close function registered in the connection table entry cn_oncl, it is invoked and passed the socket descriptor value. This allows special processing to be performed under certain conditions.

net_close()

```
void net_close(int all_but)
```

Args:

but A socket to leave open.

The network connections are closed. For each possible connection with a socket number that does not match all_buf...

If there is a non-null on-close-function pointer, cn_oncl, it is cleared. This prevents *close_conn()* from invoking the registered function. This function is typically called when a child process is created and we do not want a child to invoke the parents special routine on a descriptor it is closing. The function close_conn() is called on each open socket in the connection table that does not match the parameter *all_but*.

get_connectaddr()

```
pbs_net_t get_connectaddr(int socket)
```

A trivial routine that returns the IP address saved in the servers connection table.

get_connecthost()

```
int get_connecthost(int sock, char *buffer, int size)
```

Args:

sock the socket identifying the connection.

bufferthe buffer into which the host name is returned.

size the length of the buffer.

Returns:

0 if successful.

-1 if the buffer passed in was not large enough to hold the full name of the host. The buffer will still contain whatever part did fit.

The name of the host system to which a connection identified by sock exists is returned into buffer.

The system routine *gethostbyname* is called with the network address saved in the cn_addr member of the "connection" array. If the host is not found, the value in cn_addr is formatted into a numeric network address as is found in the file */etc/hosts*. A maximum of size characters will be copied into buffer. If the name is smaller than size, the length of name characters is copied info buffer. The name in buffer will be null terminated.

10.4.2. File: net_client.c

The file *src/lib/Libnet/net_client.c* contains client side functions to deal with the network.

client_to_svr()

```
int client_to_svr(pbs_net_t hostaddr, unsigned int port, int local_port)
```

Args:

hostaddr
the hexadecimal Internet host address, in network order) to which to connect (pbs_net_t is an unsigned long).

port The port number to which to connect.

local_port a flag, non-zero indicates an attempt should be made to bind the client to a reserved port on the local host.

Returns:

>=0 the socket number for the connection.

PBS_NET_RC_FATAL
(-1) a fatal error occurred.

PBS_NET_RC_RETRY
(-2) a temporary error occurred, may retry.

The server's host address and port are passed as parameters rather than their names to possibly save extra look-ups. It seems likely that the caller "might" make several calls to the same host or different hosts with the same port. Let the caller keep the addresses around rather than look it up each time.

A socket is allocated. If the local_port flag is non-zero, an attempt is made to bind the socket to a reserved port on the local host. This is typically done to authenticate the client to the server as this binding requires root privilege.

The address of the server host and its port placed in the internet address structure and an attempt to connect to it is made. If the connect attempt fails for a reason that might be temporary, e.g. all server ports were busy, then a {PBS_NET_RC_RETRY} is returned. Otherwise {PBS_NET_RC_FATAL} is returned to the caller. If the connect is successful, the socket number is returned.

10.4.3. File: get_hostaddr.c

The file *src/lib/Libnet/get_hostaddr.c* contains functions to look up the internet address of a host and to look up the port number for a given service.

get_hostaddr()

```
pbs_net_t get_hostaddr(char *host_name)
```

Args:

host_name
: the name of the host.

Returns:

address
: of the requested host, the address is host order (pbs_net_t is an unsigned long). Return of zero indicates an error (unknown hostname).

Calls the library routine *gethostbyname()* to find the address. The primary address is returned as a pbs_net_t type.

10.4.4. File: get_hostname.c

The file *src/lib/Libnet/get_hostname.c* contains the single function:

get_fullhostname()

```
int get_fullhostname(char *short_name, char *namebuf, int bufsize)
```

Args:

short_name
: maybe the full name or a short alias for the host.

namebuf
: pointer to a character buffer in which the full name is to be returned.

bufsize
: the length in bytes of namebuf.

Returns:

0 on success, the full hostname has been placed in namebuf.

-1 on an error.

A call is made to the library routine *gethostbyname(3)* to obtain the internet address. This address is passed to *gethostbyaddr(3)* to obtain the full and complete name. This two step process insures that PBS will use the same name regardless of alias ordering, we always use the name returned from the IP address. This name is copied into the caller provided buffer. This routine exists simply, as do most of the routines in this library, to isolate the server from the type of network.

As a kludge to handle the over loading of the colon as a separator between job ids in a dependency specification and within the job to specify test server alternative ports, both ":port" and ":port" are recognized and stripped off.

10.5. libpbs.a - Command API and Data Encode Library libpbs.a

The PBS Command Interface Library (IFL), **libpbs.a**, provides a application programming interface, API, to the PBS server. The PBS commands are implemented using these functions. It is also intended that users be able to write their own commands if they so desire. The data encode/decode are an integral part of the interface library.

10.5.1. Design Concepts of the Interface Library

The IFL is an RPC interface to the services provided by the PBS server. The functions **pbs_connect** and **pbs_disconnect** bind and unbind the library to an instance of a batch server, and the remaining (public) calls are remote calls to functions in the server. A connection identifier is returned by **pbs_connect** to identify the binding, and is input to all the other calls. Additional private routines support the public API calls.

The marshaling of the parameters and the unmarshaling of the replies are handled by private routines (enc_*.c and dec_*.c) included in the library. At the lowest level, the DIS data encode/decode provide machine independent network representation of the data.

The functions in the Interface Library can be divided into 5 groups:

- Those that manage connections.
- Those functions that are essentially direct RPCs for server functions. These map directly to batch server functions. Some are not intended to be called directly, but are rather intended to be used in a set sequence to implement some higher level part of the protocol.
- Higher level functions that call several of the direct RPC style functions to accomplish some higher protocol goal. The function **pbs_submit** is the prime example.
- Functions used to convert between data types, for example.
- Data encode and decode routines.

10.5.1.1. Types Used in Argument Lists

There are several special argument types that occur again and again in the argument lists of the library functions.

For almost all the functions, the first argument is a *connection descriptor*. The *connection descriptor* is returned by **pbs_connect,** and is an input argument to everything else. See the section on Connection Management Functions for more information.

A *job_id* is a character string the syntax and semantics of which are specified in the ERS section 2.7.6.

The *struct batch_status* data structure is the return value for all the functions that return status. It is described in, for example, section 4.3.1 of the ERS.

A *destination id* appears in several of the functions. It refers to a server, or a queue managed by a particular server. The semantics and syntax are described in sections 2.7.3 and 4.3.9 of the ERS (the man page for **pbs_movejobs**).

Finally, a ubiquitous argument called *extend* is used to allow for extensions to the POSIX standard. Its meaning is always context dependent.

10.5.1.2. Naming Conventions

All the functions in libpbs.a that are intended to be called directly have names that begin with **pbs_**. Functions that are not generally expected to be called by casual users, but that are externally visible, have names that begin with **PBS_**. Also visible, but not intended to be called by the user are the data marshaling routines starting with **enc_** and **dec_**, and the DIS encode/decode routines starting with **dis**. Static functions that cannot be called by users have no particular restrictions on the names.

10.5.1.3. Connection Management Functions

The term *connection* refers to the process of creating a binding to a particular server. Successful opening of a connection involves: 1) generating an address from the supplied server name, according to the rules specified in the ERS, 2) opening a socket, and 3) validating the user via the IFF. After it has successfully completed these tasks, **pbs_connect** will return a *connection descriptor*.

The *connection descriptor* is analogous to a UNIX file descriptor. It is actually an index into a small table of *connection state records* (CSR's). Each CSR in the table contains a socket, a stream, a place holder for a returned error number, a pointer to potential error text, and flags.

10.5.1.4. Simple RPC style functions

At the API level there are 21 Batch Requests. They are described in section 3 of the ERS. these functions follow an RPC paradigm − the arguments are marshaled (converted to a form suitable for transmission on the network) into a buffer and sent to the server. The server parses the request, does the action or notes an error, marshals a reply, and sends it. The library function receives this reply, unmarshals it, and returns the results to the caller. The most important characteristic of this model is that the library functions, with some exceptions, are essentially semantics free -- in the ideal they transparently pass their arguments to the server, and transparently return the reply. In the following descriptions, if a function follows this paradigm it is described as a "simple RPC", or words to that effect, and not described further.

The basic structure of these functions includes a call to **encode_DIS_ReqHdr** to generate the common request header data and insert the request id. A call to an *encode_DIS_* function of some sort which encodes the request body. A call to **encode_DIS_ReqExtend** to generate the common extension field (request trailer). Finally, a call to **DIS_tcp_wflush()** completely sends (flushes) the request. A call to **PBS_rdrpy** reads the reply, a little code to check for error status, a call to **PBSD_FreeReply()** and possibly other deallocation routines to deallocate the reply structure, and finally, a return of an error status. Sometimes helper functions that manipulate the more complex data structures are used to keep the code to a reasonable size.

10.5.1.5. Composite and multipurpose functions

The function **pbs_submit** has clean and simple semantics conceptually, but considerations related to atomic transactions and packaging require that the process actually be broken into several RPCs.

On the other hand, in certain cases simplicity of packaging and "object management" dictated that several of the library functions actually resolve down to one RPC call, **PBS_manager**. These functions are **pbs_alterjob**, **pbs_deljob**, **pbs_holdjob**, **pbs_manager**, **pbs_rlsjob**, and **pbs_rerunjob**.

Second are those that call **PBS_status**. They are **pbs_statjob**, **pbs_statque**, and **pbs_statsrv**.

10.5.1.6. Miscellaneous functions

There are two main categories: functions that convert from one data structure to another, or allocate or deallocate data structures; and helper functions that basically make the packaging a little better -- we try to avoid functions more than a few pages long.

10.5.2. API Modules

As a side note, most of the API and supporting routines where in files named pbs_*.c and PBS_*.c. When the newer DIS versions of the calls were introduced, they were placed in files pbsD_*.c and PBSD_*.c to separate them from the old ISODE/ASN.1 versions. The source control system is such that it is simpler to keep the new file names.

Certain of the API functions are spilt into two parts. The first part calls the second to send the request. The first part also includes the code to understand the reply. Where this spilting occurs, it is because the PBS job server needed access to the "sending" piece but not the reply piece.

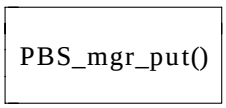
10.5.2.1. File PBSD_data.c

This file provides external data blocks used by various functions.

10.5.2.2. File PBSD_jcred.c

A simple RPC that sends the Job Credentials functions to the server. The credential (unused in this release) is nothing more than a byte array. It has no meaning to the server.

10.5.2.3. File PBSD_manage2.c



A support routine for **PBS_manager** that handles the send side of the RPC. The request body is handled by the routine *encode_DIS_Manage()*.

10.5.2.4. File PBSD_manager.c

Not to be confused with **pbs_manager**, **PBS_manager** is a lower level interface to the "manage" server function. Its interface is as follows:

```
int PBS_manager(connection, function, command, objtype, objname,
                attrib, extend)
```

The arguments for **PBS_manager**are **pbs_manager** in the ERS section 4.3.8, except for *function*, which is an integer used to describe which of the server functions is indicated. **PBS_manager**is *PBS_mgr_put()* for the send side, and *PBS_rdrpy()* for the reply.

10.5.2.5. File PBSD_msg2.c

Like pbs_manage, pbs_msgjob is spilt into two pieces. The public function, pbs_msgjob(), is found in **pbsD_msgjob.c**

PBSD_msg_put()

This part of message job is responsible for encoding and sending the request. The request body is handled by *encode_DIS_MessageJob()*.

10.5.2.6. File PBS_rdrpy.c

PBS_rdrpy()

The function is most frequently seen implementing the reply half of an RPC. It handles the egregious bookkeeping chores associated with obtaining a reply. It allocates a reply structure and uses *decode_DIS_replyCmd()* to read and decode the data. As required, any error codes and messages are placed where required. *DIS_tcp_reset()* is called to reset (pointers to) the buffer used to read/write and decode/encode DIS data.

PBSD_FreeReply()

This function frees the reply structure created in PBSD_rdrpy(). What must be freed depends on the type of reply since the structure is a union.

10.5.2.7. File PBSD_sig2.c

Again, the pbs_signaljob function is spilt into two pieces for sake of the job server. The main function is found in **pbsD_sigjob.c**.

PBSD_sig_put()

This function sends the request using *encode_DIS_SignalJob()* to encode the request body.

10.5.2.8. File PBSD_status.c

PBS_status()

This function is the basic call for obtaining status for all object types. It makes use of the helper function *PBS_status_put()* found in PBSD_status2.c to handle the send side, because the marshaling of an attribute list is rather complex. It returns a pointer to a list of *batch_status* structures via a call to *PBSD_status_get()*. The arguments are as described for **pbs_statjob** in the ERS, section 4.3.15, except for *objtype*, which describes whether the object for which status is being requested is a server, a queue, or a job. In the case of a server, the *id* argument is ignored, because the connection specifies the server.

Note that the DIS string encoding routines do not take kindly to a null string pointer. Here and else where, null pointers are converted to a pointer to the null string.

PBSD_status_get()

This function contains a messy little algorithm to take the reply structure returned by *PBSD_rdrpy()* and convert it into the expected list of **batch_status** structures. Remember, each batch_status structure is also the head of a list of attributes.

alloc_bs()

This private function allocates and initializes the space for a batch_status structure.

10.5.2.9. File PBSD_status2.c

PBS_status_put()

The above mentioned helper function. The complex marshaling of the attribute list is further helped by a call to **PBS_al2AL**.

10.5.2.10. File PBSD_submit.c:

This file contains components of the complex submit request.

PBS_rdytocmt()

This function is a component of the fairly complex commit protocol that guarantee atomic transmission of a job from a client to a server. This protocol is defined in the ERS. It is a pure RPC function as defined above that sends the "Ready to Commit" function to the server, and receives the reply.

PBS_commit()

This function completes the commit protocol required to submit a job. It sends the job id of the job that has just been queued to the server, and acts as a final acknowledgement from the client that it knows that the server has taken the job. The only characteristic of this function that keeps it from fitting the RPC model exactly is that the reply is immaterial, and thus is ignored.

PBS_scbuf()

An RPC-style function that sends a single chunk of a job script to the server. It is called, potentially many times, by the function **PBS_jscript** to send the entire job script to the server.

PBS_jscript()

Sending the job script is another component of the job submission protocol. However, since the script may need to be sent in chunks, **PBS_jscript** calls the function **PBS_scbuf** as many times as necessary to send buffer loads of the script. **PBS_jscript** has no system interaction of its own, and is a very simple function.

PBS_queuejob()

This is another RPC-style function. It sends the first request in the protocol required to submit a job to the server. This request contains all of the job control (attribute) information.

10.5.2.11. File get_svrport.c

The file *src/lib/Libnet/get_svrport.c* (which of course should be named PBS_get_svrport) contains the function:

get_svrport()

```
unsigned int get_svrport(char *service_name, char *proto, unsigned int default)
```

Args:

service_name
: the name of the service.

proto
: protocol: "tcp" or "udp".

default
: port to use.

Returns:

port number in host byte order, or -1 if an error.

The function just calls the library routine *getservbyname()* to obtain the port number. If the service is not found in /etc/services, the *default* port is returned.

This function is contained within Libifl.a, rather than Libnet.h (its natural home) because it is called from pbs_connect() and a user written program might otherwise not need Libnet.a.

10.5.2.12. File pbsD_alterjob.c

pbs_alterjob()

This function is almost a pass-through to **PBS_manage** -- the "alterjob" request is an instance of the Manage Job function. The one bit of mess required is to convert the **attrl** structures into **attropl** structures.

10.5.2.13. File pbsD_asyrun.c

pbs_asyrunjob()

This is the asynchronous version of the pbs_runjob call. It is identical with pbs_runjob() except for the request id passed in the request header.

10.5.2.14. File pbsD_connect.c

These functions provide connection management services.

pbs_connect()

pbs_connect does the following things:

- Reserves a CSR. Subsequent processing will cause fields in the CSR to be filled in, unless an error occurs somewhere along the way, in which case the CSR will be released.
- Refines the supplied server name via the rules specified in ERS section. 2.7.9. This is done through a call to the static function **PBS_get_server**.
- Through socket calls, sets up a TCP connection to the server.
- Makes sure that the user is authenticated to the Server. This is done through a call to another static function *PBS_authenticate()*.
- Lastly setups the DIS encoding buffer by calling *DIS_tcp_setup()*.

PBS_get_server()

The static function **PBS_get_server** implements the rules for instantiating a server name as defined in the ERS. From a string of the form `server_host[:port]` it returns the server host name and the port number. If `:port` is not supplied, the default port is obtained via *get_svrport()*.

pbs_default()

The function **pbs_default** will return the default server name. It is also copied into the private, static areas *dflt_server* and *server_name*. Once the default name has been gotten, it is just supplied on future calls from *dflt_server* with *server_name* updated to that string.

PBS_authenticate()

The static function **PBS_authenticate** sets up a pipe and calls the PBS program **pbs_iff** with the following arguments: the server's name, the server's port, and the number of the socket of the connection to the server created by pbs_connect. The type of the credential is read back over the pipe from pbs_iff. If the credential is not {int_BATCH_credentialtype_credential__none}, which it is always by default, then pbs_authenticate()

returns an error.

In version of PBS prior to 1.1.5, a encrypted version of a credential was generated by pbs_iff and the server and returned to pbs_connect(). This was eliminated in 1.1.5 as a step in readying PBS for general availability including export out side of the USA.

pbs_disconnect

The function **pbs_disconnect** is trivial: it shutdown the TCP/IP stream, closes the socket, and frees the CSR

10.5.2.15. File pbsD_deljob.c

pbs_deljob()

Deleting a job is actually an instance of the "manager" function. Thus **pbs_deljob** just calls **PBS_manager** with an appropriate set of arguments.

10.5.2.16. File pbs_geterrmsg.c

pbs_geterrmsg.c

Return a pointer to the last error text returned from the server.

10.5.2.17. File pbsD_holdjob.c

pbs_holdjob()

Holding a job is another instance of the "manager" function. Thus **pbs_holdjob** also just calls **PBS_manager** with appropriate arguments.

10.5.2.18. File pbsD_locjob.c

pbs_locjob()

A simple RPC that sends a job identifier to the server. The request body is encoded using *encode_DIS_JobId()*.

10.5.2.19. File pbsD_manager.c

pbs_manager()

Another straightforward call to **PBS_manager**, with the specific command being int_BATCH_request_manager.

10.5.2.20. File pbsD_movejob.c

pbs_movejob()

A simple RPC. The interface is defined in the ERS. The request body is encoded using *encode_DIS_MoveJob()*.

10.5.2.21. File pbsD_msgjob.c

pbs_msgjob()

A simple RPC. The interface is defined in the ERS. The request is sent by calling *PBSD_msg_put()*, see PBSD_msg2.c

10.5.2.22. File pbsD_orderjo.c

pbs_orderjob()

A simple RPC. The interface is defined in the ERS. This function also uses *encode_DIS_MoveJob()* to encode the request body.

10.5.2.23. File pbsD_rerunjo.c

pbs_rerunjob()

A call to rerun a job, the request body is encoded via *encode_DIS_JobId()*.

10.5.2.24. File pbsD_resc.c

encode_DIS_Resc()

```
static int encode_DIS_Resc(int sock, char **rlist, int count, resource_t rh)
```

Args

sock sock of stream connection to the PBS server.

rlist A array of resource strings.

countThe number of strings in rlist.

The value of an existing resource handle or {RESOURCE_T_NULL} for a new handle.

Returns:

The function return value is 0 on success or a PBS error number if an error occured on the request.

This internal routine encodes and writes the resource request body for the resource query, resource reserve, and resource release requests. See PBSD_resc().

PBS_resc()

```
static int PBS_resc(int c, int reqtype, char **rlist, int count, resource_t rh)
```

Args:

c Connection handle for connection to server.

reqtype

the type of request, either PBS_BATCH_Rescq, PBS_BATCH_ReserveResc, or PBS_BATCH_ReleaseResc.

rlist An array of resource strings.

countNumber of strings in rescl

rh The value of an existing resource handle or {RESOURCE_T_NULL} for a new handle.

Returns:

The function return value is 0 on success or a PBS error number if an error occured on the request.

This function is used by pbs_rescquery(), pbs_rescreserve(), and pbs_rescrelease() to format and send the releated resource request. The body is sent by calling *encode_DIS_Resc()*.

The resource strings are transparent to this routine. The general format is one of the following forms:

```
resource_name
resoruce_name=
resource_name=value
```

Currently, the only resource name recognized by the PBS server is nodes.

pbs_rescquery()

```
int pbs_rescquery(int c, char **rlist, int count,
                  int *available, int *allocated, int *reserved, int *down)
```

Args:

c connection handle from pbs_connect().

rlist An array of resource strings.

count Number of strings in rescl; also size of the following integer arrays.

available
Return: number of available resources matching the specification given in the equivalent position of rlist.

allocated
Return: number of already allocated resources matching the specification given in the equivalent position of rlist.

reserved
Return: number of reserved resources matching the specification given in the equivalent position of rlist.

down Return: number of resources matching the specification given in the equivalent position of rlist which are marked as down or off-line.

Returns:
Zero on success or a PBS error number. On success, the integer arrays pointed to by available, allocated, reserved, and down are filled in.

A Query Resource batch request is made to the server via the function *PBS_resc*(). The reply from the server is read by *PBSD_rdrpy*() and if there are no errors, the return arrays are filled in. The reply structure is freed by calling *PBSD_FreeReply*().

pbs_rescreserve()

```
int pbs_rescreserve(int c, char **rlist, int count, resource_t *rh)
```

Args:

c connection handle to the server returned by pbs_connect().

rlist An array of resource strings.

count Number of strings in rescl.

rh Input/Return: Pointer to a resource handle.

Returns:
Zero is returned on success or on error a PBS error number is returned. Also on success, if rh points to a null resource handle, {RESOURCE_T_NULL}, the location pointed to by rh is filled in with a new resource handle.

A {PBS_BATCH_ReserveResc} request is sent to the server by calling *PBS_resc*(). The reply is read via *PBSD_rdrpy()* and if there is not a handle, rh is updated with the resource handle.

Note that if the server is able to only reserve part of the requested resources, the error [PBSE_RMPART] is returned. A resource handle is also returned in that case and the resources which could be allocated are assigned to the returned handle. Should the caller wish to give up the "partial" allocation, *pbs_release*() should be called with the returned resource handle.

pbs_rescrelease()

```
int pbs_rescrelease(int c, resource_t rh)
```

Args:

c connection to the server returned by pbs_connect().

rh A resource handle returned by pbs_rescreserve().

Returns:

Zero on success or a PBS error number if a error occured.

The function *PBS_resc*() is used to send a {PBS_BATCH_ReleaseResc} request to the server.

totpool()

```
int totpool(int con, int update)
```

Args:

con Connection to the PBS server returned by pbs_connect().

update

If non-zero, make a new resource query to the server.

Returns:

Returns the total number of nodes known to the server.

If the update flag is non-zero, a new node resource query is sent to the server by calling *pbs_rescquery*() with a single string resource list of nodes. Data from the query is maintained in static memory.

If the update flag is zero, the values from the the prior query is returned from global memory. This allows for multiple totpool() and usepool() calls to be made with only one actual query going to the server.

The return value is the sum of the number of available, allocated, reserved, and down nodes.

usepool()

```
int usepool(int con, int update)
```

Args:

con Connection to the PBS server returned by pbs_connect().

update

If non-zero, make a new resource query to the server.

Returns:

Returns the number of nodes known by the server to be in use.

If the update flag is non-zero, a new node resource query is sent to the server by calling *pbs_rescquery*() with a single string resource list of nodes. Data from the query is maintained in static memory.

If the update flag is zero, the values from the the prior query is returned from global memory. This allows for multiple totpool() and usepool() calls to be made with only one actual query going to the server.

The return value is the sum of the number of allocated, reserved, and down nodes.

avail()

```
char *avail(int con, char *nodes)
```

Args:

con is a connection to the server returned by pbs_connect().

nodesis a node specification requested for a job. It is the nodes=spec string from the -l option of the qsub for the job.

Returns:

yes The character string "yes" is returned if the requested nodes are available. If the job is now run, a set of nodes can be allocated to the job which will satisified the request.

no The character string "no" is returned if the requested nodes are not currently available. They may be available at a later time. Some required node either is allocated to a job, is off-line, is down, or is reserved.

never The character string "never" is returned if no combination of the known nodes will ever satisify the request. For example the request is for more nodes than exist.

? The character string "?" is returned if the request is in error.

The nodes specication in the argument nodes is passed to a *pbs_rescquery()* call. Because the specification is generally complex, only the avaliable number returned by the server is meaninful. If it is greater than zero, it is the count of the number of nodes in the request and indicates they are available. If available is zero, one or more of the requested nodes is currently unavailable. If negative, the request cannot ever be satified.

10.5.2.25. File pbsD_rlsjob.c

pbs_rlsjob()

Another call to **PBS_manage**. The function **pbs_rlsjob** is a variant of the HoldJob request.

10.5.2.26. File pbsD_runjob.c

pbs_runjob()

A simple RPC. The interface is defined in the ERS. The body is encoded with *encode_DIS_RunJob()*.

10.5.2.27. File pbsD_selectj.c

pbs_selectjob()

This function sends a Select Jobs request and places the returned job IDs in a array. The static functions *PBSD_select_put()* and *PBSD_select_get()* do the work.

pbs_selstat()

This function also uses *PBSD_select_put()* but to send a Select Status request. Status of the selected jobs are returned rather than the Job ID, so the function *PBSD_status_get()*, see PBSD_status.c, is used to process the reply.

PBSD_select_put()

The static function PBSD_select_put() uses *encode_DIS_attropl()* to encode the list of *attropl* structures which are the job selection criteria.

PBSD_select_get()

The static function PBSD_select_get() decodes the reply to a Select Job request and builds the return value, a null terminated array of pointers to Job IDs. For historical reasons, the reply structure is build with the Job IDs in a linked list.

10.5.2.28. File pbsD_sigjob.c

pbs_sigjob()

A simple RPC. The interface is described in the ERS. The function *PBSD_sig_put()* does the work of sending the request, see PBSD_sig2.c.

10.5.2.29. File pbsD_stagein.c

This call, provided mainly for the job scheduler, directs the server to begin staging in files for the specified job. The body of the request is generated by *encode_DIS_RunJob()* Though the header has a different Request ID.

10.5.2.30. File pbs_statfree.c

pbs_statfree()

Deallocates an object of type struct batch_status.

10.5.2.31. File pbsD_statjob.c

pbs_statjob()

This function is a call to *PBS_status()*, specifying a job as the type of object for which status is desired.

10.5.2.32. File pbsD_statque.c

pbs_statque()

This function is a call to *PBS_status()*, specifying a queue as the type of object for which status is desired.

10.5.2.33. File pbsD_statsrv.c

pbs_statsrv()

This function is a call to *PBS_status()*, specifying a server as the type of object for which status is desired.

10.5.2.34. File pbsD_submit.c

pbs_submit()

This function trundles through the various components of the job submission protocol. First it checks to see that there actually is a script to send, then it calls *PBSD_queuejob* to initiate the protocol. If that works, then it calls *PBSD_jscript* to send the script to the server. Next it calls *PBSD_rdytocmt* to indicate that it is prepared to forget about the whole thing. And if *that* works, it then calls *PBSD_commit* and returns the job id to the caller.

10.5.2.35. File pbsD_termin.c

pbs_terminate()

A simple RPC. The arguments are as described in the ERS. The *shutdown* function is sent to the server at the other end of the connection. The request body is encoded using *encode_DIS_ShutDown()*.

10.5.3. Request/Reply Encode/Decode Modules

The routines that marshal and unmarshal the data being passed in request and replies are described in this section. The typical function used to marshal the data (enc_*.c) takes its parameters and uses the proper sequence of DIS write routines to encode the data as host independent strings. The typical function which unmarshals data calls the DIS read routines in the same order and places the data into a *batch_request* or *batch_reply* structure.

Unless otherwise noted, these routines return zero on success and a DIS error code on failure.

10.5.3.1. File enc_CpyFil.c

encode_DIS_CopyFiles()

Encodes the Copy Files request used between the Server and Mom. The data is taken directly from the *batch_request* structure.

Data items sent are:

```
string          job id
string          job owner               (may be null string)
string          execution user name
string          execution group name    (may be null string)
unsigned int    direction
unsigned int    count of file pairs in set
set of          file pairs:
                unsigned int    flag
                string          local path name
                string          remote path name (may be null string)
```

10.5.3.2. File enc_JobCred.c

encode_DIS_JobCred()

Encodes the request to send an opaque job credential.

Data items sent are:

```
unsigned int    Credential type
string          the credential (octet array)
```

10.5.3.3. File enc_JobFile.c

encode_DIS_JobFile()

Encodes a block of a job related file (script, checkpoint, standard out/error). Data items sent are:

```
u int   block sequence number
u int   file type (stdout, stderr, ...)
u int   size of data in block
string  job id
cnt str data
```

10.5.3.4. File enc_JobId.c

encode_DIS_JobId()

Encodes a Job Id. This is used in a number of other routines. The ID is sent as a string.

10.5.3.5. File enc_JobObit.c

encode_DIS_JobObit()

Encodes a Job Obituary Notice (batch request). The data sent is:

```
string          job id
unsigned int    status
list of         svrattrl
```

Also see *encode_DIS_svrattrl()*.

10.5.3.6. File enc_Manage.c

encode_DIS_Manage()

Everybodies favorite routine, encodes the Manage batch request. Data sent is:

```
u int           command
u int           object type
string          object name
list of         attropl
```

Also see *encode_DIS_attropl()*.

10.5.3.7. File enc_MoveJob.c

encode_DIS_MoveJob()

Encodes a Move Job request. Data sent is:

```
string        job id
string        destination
```

10.5.3.8. File enc_MsgJob.c

encode_DIS_MessageJob()

Encodes a Message Job request. Data sent is:

```
string          job id
unsigned int    which file (fileopt)
string          the message
```

10.5.3.9. File enc_QueueJob.c

encode_DIS_QueueJob()

Encodes the Queue Job request, part of a Submit job complex request. Note that a null pointer to either Job ID or Destination is replaced with a pointer to the null string so DIS does not go off the deep end. Data send is:

```
string  job id
string  destination
list of attribute, see encode_DIS_attropl()
```

10.5.3.10. File enc_Reg.c

encode_DIS_Register()

Encodes the Register Dependency request. Data encoded is:

```
string          job owner
string          parent job id
string          child job id
unsigned int    dependency type
unsigned int    operation
signed long     cost
```

10.5.3.11. File enc_ReqExt.c

encode_DIS_ReqExtend()

This function appends the "entension" to each request. An extension is provided for sites and/or future modifications. The extension is a null terminated character string. It is not generally used.

An integer of value 1 or 0 is always sent. If the extension is not null, the 1 indicates that the string follows. Otherwise, 0 is sent as the integer and no string is sent.

10.5.3.12. File enc_ReqHdr.c

encode_DIS_ReqHdr()

This function encodes the request header that is part of every batch request. The protocol ID and version are used to validate the protocol and for future modifications. The header contains the batch request ID which identifies the type of request and hence the format of the request body. The body is always followed by the extension, at least the extension flag.

Data encoded is:

```
u int     Protocol ID
```

```
u int     Protocol version
u int     Request ID
string    Name of user making the request
```

10.5.3.13. File enc_RunJob.c

encode_DIS_RunJob()

Encodes a Run Job (and other) request. Data sent is:

```
string          job id
string          destination
unsigned int     resource_handle (reserved for future use)
```

10.5.3.14. File enc_Shut.c

Encodes a Terminate (shutdown) server request. Data encoded is:

```
unsigned int    manner
```

10.5.3.15. File enc_Sig.c

encode_DIS_SignalJob()

Encodes a Signal Job request. Data encoded is:

```
string          job id
string          signal
```

Note the signal may be either the name of the signal or its value, but in either case it is a string.

10.5.3.16. File enc_Status.c

encode_DIS_Status()

Encodes a status (job, queue, server) request. The request type identifies which. The data encoded is:

```
string          object id
list of         attrl,     see encode_DIS_attrl()
```

10.5.3.17. File enc_Track.c

encode_DIS_TrackJob()

Encode a track job report (request). Data encoded is:

```
string          job id
unsigned int    hop count
string          new location
u char          job state
```

10.5.3.18. File enc_attrl.c

encode_DIS_attrl()

Used to encode a list of *attrl* structures as defined in the API. The very first data item encoded is an unsigned integer giving the number of attrl entries in the list. This may be zero, in which case nothing else follows. Then for each entry, the following is encode:

```
u int   size of the three strings (name, resource, value), including
        the terminating nulls
string  attribute name
u int   1 or 0 if resource name does or does not follow
string  resource name (if one)
string  value of attribute/resource
u int   "op" of attrlop, forced to "Set"
```

10.5.3.19. File enc_attropl.c

encode_DIS_attropl()

This is identical to the above function with the addition of the *op* field. Following the initial count of list items, the following data is sent:

```
u int   size of the three strings (name, resource, value) including
        the terminating nulls
string  attribute name
u int   1 or 0 if resource name does or does not follow
string  resource name (if one)
string  value of attribute/resource
u int   "op" of attrlop
```

10.5.3.20. File enc_reply.c

encode_DIS_reply()

Encodes a Batch Reply structure, *batch_reply*. This function does it all, the reply header and the reply union. The header consists of:

```
u int       Protocol type
u int       Protocol version
u int       Return code
u int       Auxiliary return code
```

```
 u int        Union discriminator
```

The reply union then follows. It is one of the following sets of data:

```
Null Reply

  nothing else is sent

Queue Reply, Ready to Commit Reply, or Commit Reply

  string     Job Id

Select Reply

  u int      A count of the number of Job Ids, if 0 nothing else follows
  sequence of Job Id strings

Status Reply

  u int      Number of status reply objects which follow

  u int      Object type
  string     Object name
  list of attropl,         see encode_DIS_svrattrl()

Text Reply

  string     counted byte string

Locate Job reply

  string     location of job
```

One weirdness in this is that a status reply may be *decoded* into one of three forms, but they have only one source format and on the wire they all look alike. The server maintains attributes in a *svrattrl* structure which is the source format. The server decodes into the same form while the API routines deal with either *attrl* or *attropl* structures. Regardless sof which structure will be the destination, the format on the wire closely resembles that of the attropl. See encode_DIS_svrattrl(), encode_DIS_attrl(), and encode_DIS_attropl().

10.5.3.21. File enc_svrattrl.c

encode_DIS_svrattrl()

This encodes a list of the server's *svrattrl* structures. The first item encodes is a unsigned integer count of the number of items in the list. If zero, no more data is encoded as part of this sequence. Then for each structure in the list:

```
 u int   size of the three strings (name, resource, value) including the
         terminating nulls.
 string  attribute name
 u int   1 or 0 if resource name does or does not follow
 string  resource name (if one)
 string  value of attribute/resource
```

```
u int   "op" of attrlop
```

10.5.3.22. File dec_Authen.c

decode_DIS_Authen()

Used by the Job Server to to decode the Authenticate User request body sent by the pbs_iff process. The useful data items are the user name from the header, already decoded, and the one item from the body, the unsigned integer port number from which the client has connected.

10.5.3.23. File dec_CpyFil.c

decode_DIS_CopyFiles()

Used by MOM to decode the body of a Copy Files request. The data in the request is:

```
string          job id                  (may be null)
string          job owner               (may be null)
string          execution user name
string          execution group name    (may be null)
unsigned int    direction
unsigned int    count of file pairs in set
set of          file pairs:
        unsigned int    flag
        string          local path name
        string          remote path name (may be null)
```

10.5.3.24. File dec_JobCred.c

decode_DIS_JobCred()

Used by the server to decode a Job Credential request (currently not used). The data items are:

```
unsigned int    credential type
counted string  the message
```

10.5.3.25. File dec_JobFile.c

decode_DIS_JobFile()

Used by the server or Mom to decode Job Related Job File Move request. Data items are:

```
u int           block sequence number
```

```
u int           file type (stdout, stderr, ...)
u int           size of data in block
string          job id
counted string  data
```

10.5.3.26. File dec_JobId.c

decode_DIS_JobId()

The Job Id, a simple string, is decoded into a fixed size character array using *disrfst()*. This saves having to copy it in memory.

10.5.3.27. File dec_JobObit.c

decode_DIS_JobObit()

Used by the server to decode a Job Obituary Notice (request). Data items are:

```
string          job id
unsigned int    status
list of         svrattrl,  see decode_DIS_svrattrl()
```

10.5.3.28. File dec_Manage.c

decode_DIS_Manage()

Used by the server and Mom to decode the body of a number of requests. The request id is in the header which has already be decoded.

The data decoded is:

```
unsigned int    command
unsigned int    object type
string          object name
list of attropl attributes,   see decode_DIS_svrattrl()
```

10.5.3.29. File dec_MoveJob.c

decode_DIS_MoveJob()

Used by the server to decode a Move Job request body, also the Order Job body. Data decoded is:

```
string          Job ID
string          destination or second Job ID
```

10.5.3.30. File dec_MsgJob.c

decode_DIS_MessageJob()

Used by the server and Mom to decode a Message Job request. The data items are:

```
string          job id
unsigned int    which file
string          the message
```

10.5.3.31. File dec_QueueJob.c

decode_DIS_QueueJob()

Used by the server and Mom to decode the Queue Job request which is part of the complex send job sequence. Data decoded is:

```
string  job id
string  destination
list of attributes (attropl),  see decode_DIS_svrattrl()
```

10.5.3.32. File dec_Reg.c

decode_DIS_Register()

Used by the server to decode the Register Job Dependency request body. Data decoded is:

```
string          job owner
string          parent job id
string          child job id
unsigned int    dependency type
unsigned int    operation
signed long     cost
```

10.5.3.33. File dec_ReqExt.c

decode_DIS_ReqExtend()

This function is used by the server and Mom to decode the optional extension field on a batch request. The first data item is a unsigned integer. A value of one indicates a string follows, zero says it does not.

10.5.3.34. File dec_ReqHdr.c

decode_DIS_ReqHdr()

This function is used by the server and Mom to decode the batch request header. The header contains, among other things, the request ID. The body which follows is decoded based on the id. Data decoded is:

```
unsigned int   Protocol ID
unsigned int   Protocol Version
unsigned int   Request  ID
string         Name of user making the request
```

10.5.3.35. File dec_RunJob.c

decode_DIS_RunJob()

This function is used by the server to decode the body of the Run Job and Async Run Job request. Data decoded is:

```
string          job id
string          destination
unsigned int     resource_handle (reserved for future use)
```

10.5.3.36. File dec_Shut.c

decode_DIS_ShutDown()

This function is used by the server to decode the body of the Server Shutdown (terminate) request. Data decoded is one unsigned integer for the manner of shutdown.

10.5.3.37. File dec_Sig.c

decode_DIS_SignalJob()

Used by the server and Mom to decode the Signal Job request. Data decoded is:

```
string          job id
string          signal name or numeric string of value
```

10.5.3.38. File dec_Status.c

decode_DIS_Status()

This decodes the body for a Status Job, Status Queue, or Status Server request. Data decoded is:

```
string          Job ID
list of         attrl,      see decode_DIS_svrattrl()
```

10.5.3.39. File dec_Track.c

decode_DIS_TrackJob()

Used to decode the Track Job Notice (request) by the server. Data is:

```
string          Job ID
unsigned int    hop count
string          location (new server name)
u char          state
```

10.5.3.40. File dec_attrl.c

decode_DIS_attrl()

This is a support routine used by API routines, not the servers, when decoding a list of attrl, attropl, or svrattrl which are all the same on the wire. This routine is used when the destination structure is a *attrl*.

Data decoded is:

```
u int   number of attrl in list,    if non-zero it is followed by for each:

u int   size of the three strings (name, resource, value)
string  attribute name
u int   1 or 0 if resource name does or does not follow
string  resource name (if one)
string  value of attribute/resource
u int   "op" of attrlop
```

The “op” is discarded for an attrl.

10.5.3.41. File dec_attropl.c

decode_DIS_attropl()

This is a support routine used by API routines, not the servers, when decoding a list of attrl, attropl, or svrattrl which are all the same on the wire. This routine is used when the destination structure is a *attropl*.

Data decoded is:

```
u int   number of attrl in list,    if non-zero it is followed by for each:

u int   size of the three strings (name, resource, value)
string  attribute name
u int   1 or 0 if resource name does or does not follow
string  resource name (if one)
string  value of attribute/resource
u int   "op" of attrlop
```

10.5.3.42. File dec_rpyc.c

decode_DIS_replyCmd()

This support routine is used to decode the reply sent by a server in response to a request. It is used by the **API**. It differs from *decode_DIS_replySvr()* in the structures used by the server and API to hold status data. The expected data consist of the reply header:

```
u int   protocol type
u int   protocol version
u int   return code
u int   auxiliary return code
u int   union choice discriminator
```

The remainder of the data depends on the type of reply given by the union choice discriminator. The union data is described under *encode_DIS_reply()*.

10.5.3.43. File dec_rpys.c

decode_DIS_replySvr()

This function is used by a server to decodes a batch reply. It is used by the **server**. It differs from *decode_DIS_replyCmd()* in the structures used by the server and API to hold status data. The data decoded is described under decode_DIS_replyCmd() and encode_DIS_reply().

10.5.3.44. File dec_svrattrl.c

decode_DIS_svrattrl()

Another support routine used when decoding data for the server. This routine is like *decode_DIS_attrl()* and *decode_DIS_attropl()* except that the data is placed into a *svrattrl* structure. See decode_DIS_attrl() for the data decoded.

10.5.3.45. File tcp_dis.c

This file contains functions used as I/O primitivies by DIS encode/decode routines when the data is being moved over TCP/IP. Support routines to handle the data buffers are also included.

The DIS routines use the following I/O primitivies via a set of function pointers:

dis_getc
: Get a single character from the source; for tcp set to *tcp_getc()*.

dis_gets
: Get a string from the source; for tcp set to *tcp_gets()*.

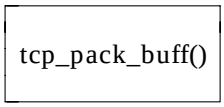
dis_puts
: Put a string to the sink; for tcp set to *tcp_puts()*.

dis_skip
: Skip over bytes in the source; for tcp (unused but) set to *tcp_rskip()*.

disr_commit
: Advance/restore the pointer to commited data in the read buffer, for tcp set to *tcp_rcommit*.

disw_commit
: Advance/restore the pointer to commited data in the write buffer, for tcp set to *tcp_wcommit*.

For tcp, the get and put routines work out of a pair of buffers, one for write (put) and one for read (get). When the read buffer is empty or the write buffer is full, *tcp_read()* or *tpc_write()* is called to fill or empty the buffer as required.

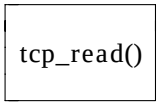



```
static void tcp_pack_buff(int n)
```

Args:

n 0 for input buffer, 1 for output buffer

This routine packes the buffer by moving any uncommited data to the beginning and adjusting the pointers.




```
static int tcp_read(int fd)
```

Args:

fd the file descriptor/socket from which to read

Returns:

>0 number of characters read

0 EOF

<0 on error

Any data left in the buffer and as yet unread, is moved to the beginning by *tcp_pack_buff()*

An original read typically follows a select() indicating that data is ready to be read. But if not all of the required data in a request is sent, the reader could “hang” waiting for more, a bad thing for a server. Hence, a local select() with a timeout is performed before the blocking read. A default timeout of 120 seconds is provided. If data does not arrive within the timeout period, a "premature end of data" is returned.

Data is read into the buffer and the EOD (end of data) pointer set after it.

DIS_tcp_wflush()

```
int DIS_tcp_wflush(int fd)
```

Args:

fd file descriptor/socket to which to write

Returns:

zer on success, -1 on error

Any "committed" data (between the start of the buffer and the trailing pointer) in the write buffer is writted to fd. The buffer is packed by *tcp_pack_buff()*.

DIS_tcp_reset()

```
void DIS_tcp_reset(int fd, int i)
```

Args:

i 0 for input buffer, 1 for output buffer

The various pointers for the buffer used by the specified file descriptor are reset to the beginning of the buffer.

tcp_rskip()

```
static int tcp_rskip(int fds, size_t ct)
```

Args:

fds file/socket descriptor.

ct Amount of data in the read buffer to skip over.

Returns

always 0

This function is unused in PBS.

tcp_getc()

```
static int tcp_getc(int fd)
```

Args:

fd file/socket descriptor on which to read data

Returns:

Character "read" or -1 if error/eof

Returns the next character from the read buffer. If the buffer is empty, *tcp_read()* is called to fill it.

tcp_gets()

```
static int tcp_gets(int fd, char *str, size_t ct)
```

Args:

fd file/socket descriptor on which to read data

str pointer to location to deliver array of bytes

ct number of bytes to deliver

Returns:

Number of bytes delivered or -1 on error/eof

Returns *ct* characters from the read buffer. If the buffer is empty, *tcp_read()* is called to fill it.

tcp_puts()

```
static int tcp_puts(int fd, char *str, size_t ct)
```

Args:

fd file/socket descriptor on which to write data

str pointer to source of array of bytes to copy into the buffer

ct number of bytes to write

Returns:

Number of bytes copied or -1 on error/eof

ct bytes are moved from *str* to the write buffer. If there is insufficient room, the buffer is written by *DIS_tcp_wflush()*.

tcp_rcommit()

```
static int tcp_rcommit()
```

Args:

fp unused

commit_flag

commit forward/backward

Returns

always zero

If *commit_flag* is true, commit the data in the read buffer by advancing the trailing pointer to the leading pointer. If false, uncommit data by the reverse operation.

tcp_wcommit()

Identical to above except works on write buffer.

DIS_tcp_setup()

```
void DIS_tcp_setup(int fd)
```

Sets up the DIS function pointers to use the above TCP based routines. Also sets up a buffer for the given file descriptor and resets the read and write buffer pointers.

[This page is blank.]

10.6. Library: Resource Monitor Library libnet.a

The resource monitor library contains functions to facilitate communication with the resource monitor. It is set up to make it easy to connect to several resource monitors and handle the network communication efficiently. In all these routines, the variable **pbs_errno** will be set when an error is indicated. The lower levels of network protocol are handled by the "Data Is Strings" **dis** library and the "Reliable Packet Protocol" **rpp** library.

delrm()

```
static int delrm(int stream)
```

Args:

stream
the stream number.

Returns:

0 if all is well *-1* indicates an error.

Search to find the connection. If it exists, close the stream and free the structure.

startcom()

```
static int startcom(int stream, int com)
```

Args:

stream
the stream number.

com command number.

Returns:

DIS_SUCCESS if all is well *anything* else indicates an error

Internal routine to compose and send the beginning of a command down a stream. A call is made to **diswsi()** with the number *RM_PROTOCOL* followed by *RM_PROTOCOL_VER* and finally *com* .

simplecom()

```
static int simplecom(int stream, int com)
```

Args:

stream
the stream number.

com command number.

Returns:

0 if all is well *-1* indicates an error

Internal routine to compose and send a "simple" command. This means anything with a zero length body. Search to find the stream number and compose the command. Use *startcom()* and **rpp_flush()** to send it across the stream. Call **rpp_eom()** to prepare the stream for reading.

simpleget()

```
static int simpleget(int stream)
```

Args:

stream
the stream number.

Returns:

0 if all is well *-1* indicates an error

Internal routine to read the return value from a command. This means anything with a zero length body. Use the function **disrsi()** to read the socket. Check the response code to see if the command succeeded.

closerm()

```
int closerm(int stream)
```

Args:

stream
the stream number.

Returns:

0 if all is well *-1* indicates an error

Close connection to resource monitor. Use *simplecom()* to send a *RM_CMD_CLOSE* command. Then use *delrm()* to close and cleanup the connection.

downrm()

```
int downrm(int stream)
```

Args:

stream
the stream number.

Returns:

0 if all is well *-1* indicates an error

Shutdown resmom. Use *simplecom()* to send a *RM_CMD_SHUTDOWN* command followed by *simpleget()* to get the response. Then use *delrm()* to close and cleanup the connection.

configrm()

```
int configrm(int stream, char *file)
```

Args:

stream
: the stream number.

file
: the configuration file name.

Returns:

0 if all is well *-1* indicates an error

Cause the resource monitor to read the file named. Use *startcom()* and **rpp_flush()** to send a *RM_CMD_CONFIG* command followed by *simpleget()* to get the response.

addreq()

```
int addreq(int stream, char *line)
```

Args:

stream
: the stream number.

line
: string for request.

Returns:

0 if all is well *-1* indicates an error

Begin a new message to the resource monitor if necessary. Then, add a line to the body of an outstanding command to the resource monitor.

allreq()

```
int allreq(char *line)
```

Args:

line
: string for request.

Returns:

the number of streams acted upon.

For each stream, begin a new message to the resource monitor if necessary. Then, add a line to the body of an outstanding command to the resource monitor.

getreq()

```
char *getreq(int stream)
```

Args:

stream
the stream number.

Returns:
a pointer to the next response line or a NULL if there are no more or an error occurred.

Finish and send any outstanding message to the resource monitor. If nothing has previously been read, call *simpleget()* to read the command response. Call **disrst()** to read the request response. If *fullresp()* has been called to turn off "full response" mode, search down the line to find the equal sign just before the response value. The returned string (if it is not NULL) has been allocated by **malloc** and **free** must be called when it is no longer needed to prevent memory leaks.

flushreq()

```
void flushreq()
```

Finish and send any outstanding messages to all resource monitors. For each active resource monitor structure, check if outstanding data is waiting to be sent. If there is, send it and mark the structure to show "waiting for response".

fullresp()

```
void fullresp(int flag)
```

Args:
flag to indicate mode.

If flag is true, turn on "full response" mode where *getreq()* returns a pointer to the beginning of a line of response. This is the default. If flag is false, the line returned by *getreq()* is just the answer following the equal sign.

activereq()

```
int activereq()
```

Return the stream number of the next stream with something to read or a negative number (the return from **rpp_poll**) if there is no stream to read.

10.7. Library: libpbs.a - Reliable Packet Protocol libpbs.a

The reliable packet protocol library contains routines to provide reliable, flow-controlled, two-way transmission of data. Each data path will be called a "stream" in this document. The advantage of RPP over TCP is that many streams can be multiplexed over one socket. This allows simultaneous connections over many streams without regard to the system imposed file descriptor limit.

Each stream has a state associated with it. The state values are:

```
#define RPP_DEAD        -1
#define RPP_FREE        0
```

```
#define RPP_OPEN_PEND   1
#define RPP_OPEN_WAIT   2
#define RPP_CONNECT     3
#define RPP_CLOSE_PEND  4
#define RPP_LAST_ACK    5
#define RPP_CLOSE_WAIT1 6
#define RPP_CLOSE_WAIT2 7
#define RPP_STALE       99
```

A state diagram which gives the transitions follows. The label for each arc specifies an input such as a packet or user call, followed by an output. Some transitions take place with no output. In the state *RPP_OPEN_WAIT* it is legal to write but not read. In the state *RPP_CONNECT* both reads and writes are legal. No other state allows reads or writes.

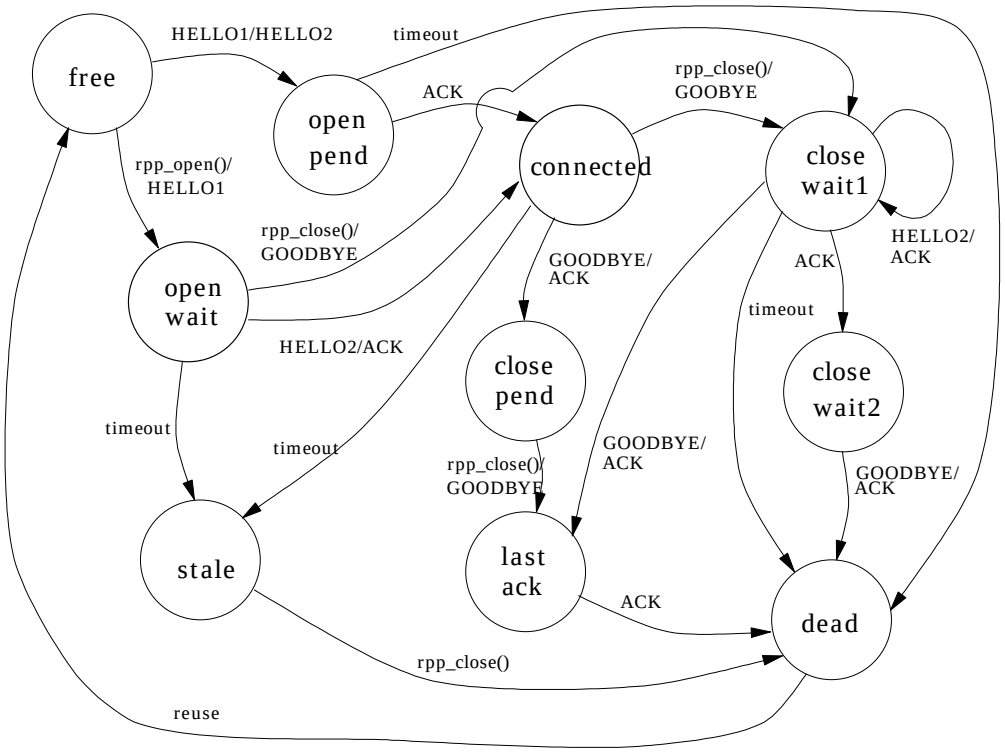


Figure 10 - 1

10.7.1. Structures and Defines

This library sends UDP packets over the network. Each packet can optionally have a CRC checksum calculated to insure data integrity. To compile without the CRC checksum calculation, define NO_CRC. All processes using RPP must have the same definition of NO_CRC to be able to communicate. The structures used to organize the individual data streams and packets are as follows:

```
struct  send_packet     {
        u_char  *data;
        u_short type;
        u_short sent_out;
        int     len;
        int     index;
        int     sequence;
```

```
        time_t  time_sent;
        struct  send_packet     *next;
        struct  send_packet     *up;
        struct  send_packet     *down;
};

struct  recv_packet     {
        u_char  *data;
        u_short type;
        int     len;
        int     sequence;
        struct  recv_packet     *next;
};

struct  pending {               /* data pending commit */
        u_char  *data;
        struct  pending *next;
};

struct  stream {
    int          state;
        struct  sockaddr_in     addr;
        struct  in_addr         *addr_array;
        int                     stream_id;
    time_t           open_key;
        int                     msg_cnt;
        int                     send_sequence;
        struct  pending         *pend_head;
        struct  pending         *pend_tail;
        int                     pend_commit;
        int                     pend_attempt;
        struct  send_packet     *send_head;
        struct  send_packet     *send_tail;
        int                     recv_sequence;
        struct  recv_packet     *recv_head;
        struct  recv_packet     *recv_tail;
        int                     recv_commit;
        int                     recv_attempt;
};
```

The different packet types are as follows:

```
#define RPP_ACK         1
#define RPP_DATA        2
#define RPP_EOD         3
#define RPP_HELLO1      4
#define RPP_HELLO2      5
#define RPP_GOODBYE     6
```

10.7.2. Functions File: rpp.c

The file *src/lib/Libifl/rpp.c* contains functions entry points to the RPP package as well as internal functions not visible to a user.

crc()

```
u_long crc(u_char *buf; u_long clen)
```

Args:

buf the data area to checksum.

clen the length of the data area.

Returns:

the checksum of the data area.

Compute a POSIX 1003.2 checksum. This routine is copyrighted by The Regents of the University of California. It is derived from software contributed to Berkeley by James W. Williams of NASA Goddard Space Flight Center. The disclaimer required for redistribution and use is included with the source code.

rpp_form_pkt()

```
static void rpp_form_pkt(int index, int type, int seq, u_char *buf; int len)
```

Args:

indexthe stream number.

type the type of packet.

seq the sequence number.

buf a memory area to use.

len the length of the memory area.

Returns:

none

Create a packet of the given type, fill in the sequence and index number. If buf is NULL, malloc an area for just a header. If buf is not NULL, it should contain space for len+RPP_PKT_HEADER bytes.

rpp_check_pkt()

```
static struct stream *rpp_check_pkt(int index, struct sockaddr_in *addrp)
```

Args:

indexthe stream number.

addrpthe network address of a packet.

Returns:

the stream structure pointer of the stream addrp matches or NULL.

Check the port and family of *addrp* with those of the stream structure indicated by *index*. If they match, compare the IP addresses. If they match too, return a pointer to the stream. If they don't match, loop through the *addr_array* for the stream. This contains the alternate IP address for the host. If any of these match, return a pointer to the stream. If not, return NULL.

rpp_send_out()

```
static void *rpp_send_out()
```

Args:

none

Returns:

none

Loop through the *send_packet* structures linked together in a queue staring from the global variable *top* and working down to *bottom*. If a packet has been sent before and it has not been sent for *RPP_TIMEOUT* seconds it will be sent again. If a packet has not been sent yet and the number of packets "out on the wire" *pkts_sent* is less than *RPP_HIGHWATER* it will be sent. The system call **sendto()** is used to send an UDP packet to the packet's recipient. When the loop through the outstanding packets is done, the *busy_count* variable is set. If it is less then *pkts_sent* it is set to *pkts_sent*. Otherwise, a calculation is done to slowly filter down the busyness factor. This is used in *rpp_stale()* to see if too many retrys have been done on a packet.

rpp_send_ack()

```
static int *rpp_send_ack(struct stream *sp, int seq)
```

Args:

sp the stream to send down.

seq the sequence number to acknowledge.

Returns:

-1 on error

0 if not

Send an *RPP_ACK* packet for the sequence number given. If the *stream_id* for the stream is less than zero, don't send anything because the *RPP_HELLO2* packet has not been received yet to tell us the other side's stream number.

clear_stream()

```
static void *clear_stream(struct stream *sp)
```

Args:

sp the stream to clear.

Returns:

none

Remove packets from receive, pending and send queues for a stream, free all the memory and set the stream state to *RPP_DEAD*.

rpp_recv_pkt()

```
int rpp_recv_pkt(int fd)
```

Args:

fd

Returns:

>=0 stream index number

<0 code

Do a recvfrom on the socket given by *fd* to get a packet. This may change the state of a stream. Read the packet information and check the crc. If there is no error, check the packet type. For an *RPP_ACK*, find the packet that is being acknowledged and compute any required state change. Take the acknowledged packet off the send queue for the stream and return the stream index number. If the packet is an *RPP_GOODBYE*, send an acknowledge for it and change state as needed. If it is an *RPP_DATA* or *RPP_EOD*, check to make sure it is not an old sequence number that we already have before putting in in the receive queue for the stream, then acknowledge it. If the packet is an *RPP_HELLO1*, this is the start of an open sequence. *RPP_HELLO1* packets have the remote side's stream index in the "streamid" field and a stream key value in place of the sequence number. Create a stream entry in the stream_array and send a *RPP_HELLO2*. If the packet is a *RPP_HELLO2,* we *RPP_HELLO2* packet has this side's stream index in the streamid field as usual and the remote side's stream index overloaded in the "sequence" field. Send an acknowledge with the stream key value as the sequence number and change state. Return the index of the stream the packet belonged to or -2 if it was not data, or -1 if there was an error. Return -3 if there was no data to read.

rpp_recv_all()

```
static int rpp_recv_all()
```

Args:

none

Calls *rpp_recv_pkt* for each socket being used by the library.

rpp_stale()

```
static void rpp_stale(struct stream *sp)
```

Args:

sp pointer to stream being checked.

Check to see if any packet being sent out on a stream has been sent more than a reasonable number of times.

rpp_dopending()

```
static int rpp_dopending(int index, int flag)
```

Args:

index the stream number.

flag flag to determine if we send an *RPP_EOD* packet.

Returns:

-1 on error

0 otherwise

Form data packets for any pending data. If flag is true, create an *RPP_EOD* packet too.

rpp_flush()

```
int *rpp_flush(int index)
```

Args:

index the stream number.

Returns:

-1 on error

0 otherwise

Flush all data out of a stream by calling *rpp_dopending()* if there is data pending or an end-of-message mark needs to be sent. Then call *rpp_send_out()* and *rpp_recv_pkt()*.

rpp_bind()

```
int *rpp_bind(int port)
```

Args:

port the port number to use.

Returns:

-1 on error

0 otherwise

The library can manage more than one socket. If the global *rpp_fd* is -1, call **socket()** to open a new UPD socket, then **fcntl()** to set the FD_CLOEXEC and FNDELAY flags. Optionally, call **atexit()** with **rpp_shutdown()** as the function pointer to call before exit. Then call **bind** to fix the socket to the given port number.

rpp_open()

```
int *rpp_open(char *name, int port)
```

Args:

namethe name of the host.

port the port number to use.

Returns:

-1 on error

0 otherwise

If the UPD socket has not been opened yet, call **rpp_bind** to set it up. Create an entry in stream_array for the new stream and copy the information for the host into it. Call *rpp_form_pkt()* to put a RPP_HELLO1 packet into the send queue for the stream and call *rpp_send_out()* and rpp_recv_pkt() to send it and get a response.

rpp_close()

```
int *rpp_open(int index)
```

Args:

indexthe stream number.

Returns:

-1 on error

0 otherwise

If the stream is in state *RPP_STALE*, call *clear_stream()* to finish it off. If it is in state *RPP_CLOSE_PEND* change state to *RPP_LAST_ACK* and send a *RPP_GOODBYE*. If it is in state *RPP_OPEN_WAIT* or *RPP_CONNECT* change state to *RPP_CLOSE_WAIT1* and call *rpp_dopending()* if there is anything on the pend queue. Then send a *RPP_GOODBYE* packet.

rpp_write()

```
int *rpp_write(int index, void *buf, int len)
```

Args:

indexthe stream number.

buf the data to write.

len the number of characters to write.

Returns:

-1 on error

>=0 the number of characters written.

Check if any data has been written to the stream which has not remained unacknowledged for so long that it is "stale". If so, abort the write and return -1. Otherwise, create RPP_DATA packets for the data in buf and link them to the pending queue for the stream.

rpp_attention()

```
static int *rpp_attention(int index)
```

Args:

index the stream number.

Returns:

TRUE or FALSE

Check a stream to see if it needs "attention". If the stream state is *RPP_STALE*, the user needs to call close so return TRUE. If there is a message to read, return TRUE, otherwise return FALSE.

rpp_read()

```
int *rpp_read(int index, void *buf, int len)
```

Args:

index the stream number.

buf the data area to read into.

len the size of buf.

Returns:

-1 on error

-2 if other side has closed

>=0 number of bytes read

Call *rpp_attention()* in a loop until it returns TRUE or an error occurs. Then find the packet in the receive queue for the stream that the field recv_attempt point into. Copy data starting from there until the end of a message or the end of the provided buffer.

rpp_rcommit()

```
int *rpp_rcommit(int index, int flag)
```

Args:

index the stream number.

flag TRUE to commit or FALSE to decommit.

Returns:

-1 on error, TRUE or FALSE.

Commit data which has been read up to recv_attempt if flag is TRUE. Otherwise, set recv_attempt back to the previous commit point recv_commit. Return -1 on error, FALSE on decommit or if end-of-message has not been reached, TRUE if end-of-message has been reached.

rpp_eom()

```
int *rpp_eom(int index)
```

Args:

indexthe stream number.

Returns:

-1 on error, TRUE or FALSE.

Reset end-of-message condition on a stream. Any packets on the receive queue are freed. Return -1 on error, -2 if stream is closed, 0 otherwise.

rpp_wcommit()

```
int *rpp_wcommit(int index, int flag)
```

Args:

indexthe stream number.

flag TRUE to commit or FALSE to decommit.

Returns:

-1 on error

0 otherwise.

If flag is TRUE, call *rpp_dopending()* to transfer any packets from the pend queue to the send queue, then call *rpp_send_out()* and *rpp_recv_pkt()* to send the packets and get any response. If flag is FALSE, free any packets from the pend queue which follow the first which is kept.

rpp_poll()

```
int *rpp_poll()
```

Args:

none

Returns:

>=0 the stream number of any stream with a message waiting,

-1 on error

-2 otherwise

Call *rpp_recv_pkt* until an error occurs or a -3 is returned meaning no data was read. Then loop over all streams calling *rpp_attention* to see if there is any condition which needs to be reported. If so, return the stream number, otherwise return -2.

rpp_getc()

```
int *rpp_getc(int index)
```

Args:

index the stream number.

Returns:

-1 on error

>=0 the character read

Call **rpp_read** to read one character from a stream.

rpp_putc()

```
int *rpp_putc(int index, int c)
```

Args:

index the stream number.

c the character to write.

Returns:

-1 on error *0* otherwise.

Call **rpp_write()** to write one character to a stream.

10.8. Library: libsite.a - Site Modifiable Library libsite.a

The site modifiable library contains stubs or the default version of routines specifically set up to be modifiable by an individual site. By placing the PBS project released version of these routines in a library, the site can implement and compile its own version without fear of that version being overwritten by the next release of PBS. This library is currently linked with the Server and with MOM.

10.8.1. How to Modify these Routines

The site should make a source file for each function to be replaced and compile the objection into the appropriate directory in the target tree. The object file name should start with *site_* and end of course with *.o*. The function names must agree with the name in the library as documented in this section of the IDS. The calling parameters must also match the library version as documented.

When the daemon is being linked, it will pickup any (site provided) object files with the name *site_*.o*. Any functions provided in these object files will be linked into the daemon satisfying the external and therefore the object module from this library will not be included.

For example, if a site wishes to replace the function site_check_user_map() with something local, you should write the code in a file site_foo.c. Then

```
cp site_foo.c $(PBS_TARGET)/obj/server/site_foo.c
cd $(PBS_TARGET)/obj/server/site_foo.c
cc -c -I$(PBS_SRC)src/include site_foo.c
```

When the server is linked, the local site_check_user_map() will be picked up from site_foo.o.

The description of the routines included in libsite.a can be found under the daemon affected in the subsection entitled *Site Modifiable Files*. These subsections are typically located at the end of the daemon chapter.

11. Interprocess Communication

11.1. InterProcess Communication"

The current version of PBS uses the *Data is Strings*, DIS, encoding for the interprocess communication between all parts of PBS. Details of DIS and the RPC used between clients (commands) and the server, betweens servers, between the scheduler and server, and between the server and Mom are covered in chapter 11 **Network Protocol** of the ERS.

The communication between the Scheduler and Resource Monitor is different and is discussed in chapter 7 of the IDS.

[This page is blank.]

12. Graphical User Interface

12.1. GUI Overview

This section describes the different routines supporting the graphical user interfaces **xpbs** and **xpbsmon**. Most of the routines are written in Tcl/Tk.

12.2. xpbs Packaging

The main file called **xpbs** contains the main() section of the GUI; it starts up appropriate routines on its event loop to respond to actions like mouse presses. Related procedures, call-back functions are grouped together in a file. Files with the ".tk" suffix contain Tk-related procedures while those with ".tcl" suffix contain non-Tk related routines. Bitmap files used by the GUI are located in the **bitmaps** directory. Codes written in C are in the **Ccode** directory. Help files accessed by the GUI are in the **help** directory.

12.3. File: xpbs

This file is where the main event loop is located.

12.3.1.

main()

```
main{argc argv}
```

Args:

argc The number of arguments on the command line.

argv The **argv** list contain the following arguments:
[−admin]

Returns:

Zero, if no command line syntax errors are detected. Positive, otherwise.

Control Flow:

set appropriate Tcl/Tk library directories, program version number, and program paths.

Get the user's name.

Process command line arguments (if any).
if argument is "-admin" then
set perm_level to "admin"
else
set perm_level to "user"
endif

Load xpbs resource values as supplied from the X resources files: global and user's .xpbsrc file.
Set default values for unset xpbs resources.
Set the colors of various widgets based on xpbs color resources.
Save the initial values of the xpbs resources.

Set the mainWindow path and make it visible.

Set listbox-related parameters.

Set the months values.

Set polling-related parameters such as the data update sequence and the trackjob update sequence.

Call set_pbs_commands
Call set_pbs_options
Call set_pbs_defaults
Build the main display of xpbs containing the necessary widgets.
Set properties involving the window manager.
Register bindings to the main listboxes.
Get the data for the Hosts, Queues, and Jobs listbox. Place a delay of 500 ms to allow the window display to complete its drawings.

12.4. File: main.tk

This file contains the routines for creating the main **xpbs** display.

12.4.1.

build_main_window()

```
build_main_window{}
```

Control Flow:

Create the menubar, hosts, queues, jobs, and statusbar frames.
Make the menubar (fixed located) frame visible.

Fill the menubar with widgets.

Make the hosts, queues, jobs frames visible.
Fill the hosts frame with widgets.
Fill the queues frame with widgets.
Fill the jobs frame with widgets.
Fill the statusbar with widgets.

Create the iconized versions of the hosts, queues, jobs, and info frames.
Fill the iconized frames with widgets.

Make the iconized views visible if appropriate icon* resources are set in xpbsrc file.

12.4.2.

fillIconizedFrame()

```
fillIconizedFrame{text cmd widget_name}
```

Args:

text	The leading text label to be written on the icon bar.
cmd	The Tcl procedure to call when the maximize button is clicked.
widget_name	The widget where the icon frame will be displayed.

Control Flow:

Create the label for the icon bar.
Create the maximize button for the icon bar.
Make label and button visible.

12.4.3.

iconizeHostsView()

```
iconizeHostsView{}
```

Control Flow:

Make Hosts icon bar visible.
Remove from view the Hosts frame.
Set iconview(hosts) to true

12.4.4.

iconizeQueuesView()

```
iconizeQueuesView{}
```

Control Flow:

Make Queues icon bar visible.
Remove from view the Queues frame.
Set iconview(queues) to true

12.4.5.

iconizeJobsView()

```
iconizeJobsView{}
```

Control Flow:

Make Jobs icon bar visible.
Remove from view the Jobs frame.
Set iconview(jobs) to true

12.4.6.

iconizeInfoView()

```
iconizeInfoView{}
```

Control Flow:
Make Info icon bar visible.
Remove from view the Info frame.
Set iconview(info) to true

12.4.7.

maximizeHostsView()

```
maximizeHostsView{}
```

Control Flow:
Make Hosts frame visible.
Remove from view the Hosts icon bar.
Set iconview(hosts) to false.

12.4.8.

maximizeQueuesView()

```
maximizeQueuesView{}
```

Control Flow:
Make Queues frame visible.
Remove from view the Queues icon bar.
Set iconview(queues) to false.

12.4.9.

maximizeJobsView()

```
maximizeJobsView{}
```

Control Flow:
Make Jobs frame visible.
Remove from view the Jobs icon bar.
Set iconview(jobs) to false.

12.4.10.

maximizeInfoView()

```
maximizeInfoView{}
```

Control Flow:
Make Info frame visible.
Remove from view the Info icon bar.
Set iconview(info) to false.

12.4.11.

fillHostsFrame()

```
fillHostsFrame{widget_name}
```

Args:

widget_name The widget frame where Hosts-related widgets are to be placed.

Control Flow:
Create the Hosts leading bar.
Create the Hosts body frame.
fill the Hosts leading bar with widgets.
fill the Hosts body frame with widgets.

12.4.12.

fillHostsHeaderFrame()

```
fillHostsHeaderFrame{widget_name}
```

Args:

widget_name The widget frame where Hosts leading bar-related widgets are to be placed.

Control Flow:
Create the header label.
Create the minimize button.
Make the label and button visible.

12.4.13.

fillHostsListFrame()

```
fillHostsListFrame{widget_name}
```

Args:

widget_name The widget frame where Hosts body-related widgets are to be placed.

Control Flow:

Create the Hosts listbox.
Create the command buttons, including only those that are consistent with user's perm_level.
Make the listbox and the buttons visible.

12.4.14.

fillQueuesFrame()

```
fillQueuesFrame{widget_name}
```

Args:

widget_name The widget frame where Queues-related widgets are to be placed.

Control Flow:

Create the Queues leading bar.
Create the Queues body frame.
fill the Queues leading bar with widgets.
fill the Queues body frame with widgets.

12.4.15.

fillQueuesHeaderFrame()

```
fillQueuesHeaderFrame{widget_name}
```

Args:

widget_name The widget frame where Queues leading bar-related widgets are to be placed.

Control Flow:

Create the header label.
Create the minimize button.
Create the "Listed By Hosts(s)" criteria read-only entry widget.
Make the label and button visible.

12.4.16.

fillQueuesListFrame()

```
fillQueuesListFrame{widget_name}
```

Args:

widget_name The widget frame where Queues body-related widgets are to be placed.

Control Flow:
Create the Queues listbox.
Create the Queues command buttons, including only those that are consistent with user's perm_level.
If perm_level is set to "admin", include only buttons
Make the listbox and the buttons visible.

12.4.17.

fillJobsFrame()

```
fillJobsFrame{widget_name}
```

Args:
widget_name The widget frame where Jobs-related widgets are to be placed.

Control Flow:
Create the Jobs leading bar.
Create the Jobs criteria frame.
Create the Jobs body frame.
fill the Jobs leading bar with widgets.
fill the Jobs criteria frame with widgets.
fill the Jobs body frame with widgets.

12.4.18.

fillJobsHeaderFrame()

```
fillJobsHeaderFrame{widget_name}
```

Args:
widget_name The widget frame where Jobs leading bar-related widgets are to be placed.

Control Flow:
Create the Jobs header label.
Create the Jobs minimize button.
Create the "Listed By Queue(s)" criteria read-only entry widget.
Make the label, button, and criteria visible.

12.4.19.

fillJobsMiscFrame()

```
fillJobsMiscFrame{widget_name}
```

Args:

widget_name The widget frame where Jobs criteria-related widgets are to be placed.

Control Flow:

Create the Jobs select criteria buttons. Configure the buttons with the appropriate text variables.
Create the "Select Jobs" button and configure its Tcl command.
Create the criteria label.
Make the buttons and label visible.

12.4.20.

fillJobsListFrame()

```
fillJobsListFrame{widget_name}
```

Args:

widget_name The widget frame where Jobs body-related widgets are to be placed.

Control Flow:

Create the Jobs listbox.
Create the Jobs command buttons, including only those that are consistent with user's perm_level.
Make the listbox and the buttons visible.

12.4.21.

fillStatusbarFrame()

```
fillStatusbarFrame{widget_name}
```

Args:

widget_name The widget frame where Info bar-related widgets are to be placed.

Control Flow:

Create the Info leading bar frame.
Create the Info body frame.
Fill the Info leading bar with widgets.
Fill the Info body frame with widgets.

12.4.22.

fillStatusbarHeaderFrame()

```
fillStatusbarHeaderFrame{widget_name}
```

Args:

widget_name The widget frame where Statusbar-related widgets are to be placed.

Control Flow:
Create the label for the Info leading bar.
Make the label visible.

12.4.23.

fillMenubarFrame()

```
fillMenubarFrame{widget_name}
```

Args:

widget_name	The widget frame where menu buttons are to be placed.

Control Flow:
Create the menu buttons and configure them with appropriate commands.

12.5. File: wmgr.tk

This file contains window manager-related routines.

12.5.1.

set_wmgr()

```
set_wmgr{toplevel_win}
```

Args:

toplevel_win	the top level window to decorate.

Control Flow:
set window minimum size.
set the window title.
set the window icon.

12.6. File: bindings.tk

This file contains routines for customizing the bindings for some of the widgets.

12.6.1.

listbox_non_contiguous_selection()

```
listbox_non_contiguous_selection{ W cur_selection new_selection}
```

Args:

W	names a listbox widget
cur_selection	index to the currently selected/highlighted entry of the listbox.
new_selection	index to a newly selected/highlighted entry of the listbox. Can be "end" to refer to the last item on the listbox.

Control Flow:

This is a routine that is only invoked in Tk 3.6 which does not support non-contiguous selection of entries on a listbox.

If no selection is currently selected, then
simply select the entry at "new_selection".
endif

let min_cur_selection be the smallest cur_selection index value
let max_cur_selection be the larget cur_selection index value
if new_selection is > max_cur_selection, then
position the new entry after the entry at max_cur_selection
add the new entry to current selection list
elseif new_selection < min_cur_selection, then
position the new entry before the entry at min_cur_selection
add the new entry to current selection list
endif

Adjust the listbox view so that the entry at the smallest selected index value is shown at the top of the listbox.

12.6.2.

bind_listbox_single_select()

```
bind_listbox_single_select{widget_name}
```

Args:

widget_name names a listbox widget to be made into a single selection only listbox

Control Flow:

Disable bindings for B1-Motion, Shift-1, Shift-B1-Motion, 2, B2-Motion keys.

12.6.3.

bind_listbox_select()

```
bind_listbox_select{widget_name, boxframe}
```

Args:

widget_name names a listbox widget to be made into a single selection only listbox

boxframe associated box frame.

Control Flow:

Whatever is selected in the 'widget_name', then a boxSelect is done to 'boxframe'.

12.6.4.

bind_text_readonly()

```
bind_text_readonly{widget_name}
```

Args:

widget_name names a text widget to be made read-only.

Control Flow:

disable text widget's bindings for <1>, <Double-1>, <Triple-1>, <B1-Motion>, <Shift-B1-Motion>, <Return>, <BackSpace>, <Delete>, <Control-h>, <Control-d>, <Control-v>.

12.6.5.

bind_entry_readonly()

```
bind_entry_readonly{widget_name}
```

Args:

widget_name names an entry widget to be made read-only.

Control Flow:

disable entry widget's key bindings.

12.6.6.

register_dependency()

```
register_dependency{}
```

Control Flow:

Make the following bindings to the Queues listbox:

For button presses <1>, <B1-Motion>, <Shift-1>, <Shift-B1-Motion>, <Cntrl-1>,
Depending on what Queue listbox entries got selected,
load the appropriate jobs on the Jobs listbox.

set the queuesSelected global variable
if all Queues listbox entries are selected, then
Set the queuesSelMode button to "Deselect All"
else
Set the queuesSelMode button to "Select All"
endif

For button press <Double-1>,
get the details about the selected queue.

Make the following bindings to the Hosts listbox:

For button presses <1>, <B1-Motion>, <Shift-1>, <Shift-B1-Motion>, <Cntrl-1>,
Depending on what Hosts listbox entries got selected,
load the appropriate queues on the Queues listbox.

set the hostsSelected global variable

if all Hosts listbox entries are selected, then
Set the hostsSelMode button to "Deselect All"
else
Set the hostsSelMode button to "Select All"
endif

For button press <Double-1>,
get the details about the selected host.

Make the following bindings to the Jobs listbox:

For button presses <1>, <B1-Motion>, <Shift-1>, <Shift-B1-Motion>, <Cntrl-1>,
set the jobsSelected global variable
if all Hosts listbox entries are selected, then
Set the hostsSelMode button to "Deselect All"
else
Set the hostsSelMode button to "Select All"
endif

For button press <Double-1>,
get the details about the selected host.

12.6.7.

register_trackjob_box()

```
register_trackjob_box{listbox}
```

Args:

listbox names a listbox widget that holds those job ids that have returned output files.

Control Flow:

Set up a binding so that single clicking a job id entry will bring
up the output file/error file contents window.

12.6.8.

register_default_action()

```
register_default_action{toplevel button}
```

Args:

toplevel	names a toplevel to bind the <Return> key to.
button	the button on the toplevel window that will be invoked by default.

Control Flow:

Bind the <Return> key of toplevel so that button will flash and then execute its associated Tcl command.

12.6.9.

bind_entry_tab()

```
bind_entry_tab{entry_name next_entry_name prev_entry_name {env 1}}
```

Args:

entry_name	names an entry widget to be bound to <Tab>, <Cntrl-f>, <Cntrl-b> keys.
next_entry_name	the entry widget that will get the focus when <Tab>, or <Cntrl-f> key is pressed while in entry_name.
prev_entry_name	the entry widget that will get the focus when <Cntrl-b> key is pressed while in entry_name.
env	flag that is set when binding an entry related to environment variables.

Returns:

Sets the following global variables:
next<entry_name> - next_entry_name
prev<entry_name> - prev_entry_name

Control Flow:

Bindings for <Tab> and <Cntrl-f>:

Set the focus to next_entry_name
Position the cursor to be at the end of the string.

if env is set, then load the next_entry_frame with the environment variable value of the name specified in entry_name.

Bindings for <Cntrl-b>:

Set the focus to prev_entry_name
Position the cursor to be at the end of the string.

12.6.10.

bind_entry_overselect()

```
bind_entry_overselect{entry_name}
```

Args:

entry_name names an entry widget to be converted into a user-friendly widget.

Control Flow:

Specify bindings to allow overwriting of selected/highlighted text when any key is pressed, allow copying and pasting text via sole use of mouse button, and allow left and right arrow keys to be used.

12.6.11.

bind_text_overselect()

```
bind_text_overselect {entry_name}
```

Args:

entry_name names an entry widget to be converted into a user-friendly widget.

Control Flow:

Specify bindings to allow overwriting of selected/highlighted text when any key is pressed, allow copying and pasting text via sole use of mouse button, and allow left, right, up, or down arrow keys to be used.

12.6.12.

register_spinbox_entry()

```
register_spinbox_entry {entry_name}
```

Args:

entry_name names a spinbox entry widget. This widget has an associated global variable named: vlist.<entry_name> - a list of valid discrete values.

Control Flow:

Addition to <AnyKeyPress> binding:
If spinbox_value_list is a range of numbers (min-max), then
blank out entry if any of the following conditions is met:
case when range does not contain negative numbers:
(1) user entered something that is non-numeric
(2) user entered something not between min and max
case when range contains negative numbers:
(1) if user typed as first character non-numeric or not "-"
(2) if the rest of characters typed by user is non-numeric
(3) if entry value does not fall between the min and the max.
else
blank out entry if user typed something where the leading characters
don't match (regular expression) any of the values in spinbox_value_list.
endif

Bind <FocusOut> to spinbox entry so as
to call check validy of spinbox value

12.6.13.

register_entry_fixsize()

```
register_entry_fixsize {entry size}
```

Args:

entry	names a spinbox entry widget.
size	size to constraint the enty widget with.

Control Flow:

Addition to <AnyKeyPress> binding:
allow insertion of new chars if it will not exceed entry length of 'size'.

12.7. File: pbs.tcl

This file contains routines that are dependent on PBS commands.

12.7.1.

getdata()

```
getdata{server_names {jobs_info_only 0}}
```

Args:

server_names	the list of servers to query for data
jobs_info_only	a boolean value that instructs getdata to only obtain Jobs information.

Control Flow:

```
set busy_cursor
if jobs_info_only; then
  set dataCmd to "PBS_QSTATDUMP_CMD -J <select_options> server_names"
  unset jobinfo array which holds the Jobs listbox entries
else
  set dataCmd to "run PBS_QSTATDUMP_CMD <select_options> server_names"
  unset hostinfo which holds the Hosts listbox entries
  unset qinfo which holds the Queues listbox entries
  unset jobinfo which holds the Jobs listbox entries
  clear out the entries from the hosts, queues, and jobs listboxes
endif

execute dataCmd
if execution failed ; then
  remove_busy_cursor
  return
endif

initialize where - refers to what listbox an output of data should
    go based on its preceding header
Foreach line_of_output obtained from dataCmd;
```

```
do
  NOTE: assuming of course that the header precedes all data.
  if line_of_output matches
     HOSTS_COLUMN_LABEL; then set where to "hosts"
     QUEUES_COLUMN_LABEL; then set where to "queues"
     JOBS_COLUMN_LABEL; then set where to "jobs"
     LINES_TO_IGNORE; then throw away the line_of_output
  else (assuming that we've already obtained our "where" listbox),
    initialize fkey
    if where listbox is "hosts", set fkey indices to HOSTS_LISTBOX_FKEY
    if where listbox is "queues", set fkey indices to QUEUES_LISTBOX_FKEY
    if where listbox is "jobs", set fkey indices to JOBS_LISTBOX_FKEY

    From the line_of_output, build the list of fkey (foreign key)
    values, separated by @, by getting the corresponding values to fkey
    indices. For example, suppose:
      the line_of_output is "23.43 al bayucan", and fkey is {1, 3},
      then fkeyval will be set to "23.43@bayucan"

    if where listbox is "hosts",
       append line_of_output to hostinfo(fkeyval) array
       append line_of_output to Hosts listbox
       Non-contiguous highlight/select the line_of_output in Hosts listbox
        if its primary key value matches one of hostsSelected
    else if where listbox is "queues",
       append line_of_output to qinfo(fkeyval) array
       if line_of_output's fkeyval matches one of hostsSelected entries,
       then
         append line_of_output to Queues listbox
         Non-contiguous highlight/select the line_of_output in Queues listbox
          if its primary key value matches one of queuesSelected
       endif
    else if where listbox is "jobs",
       append line_of_output to jobinfo(fkeyval) array
       if line_of_output's fkeyval matches one of queuesSelected entries,
       then
         append line_of_output to Jobs listbox
         Non-contiguous highlight/select the line_of_output in Jobs listbox
          if its primary key value matches one of jobsSelected
       endif
    endif

    reset values for hostsSelected, queuesSelected, jobsSelected based on
      what's currently selected/highlighted on the different listboxes.

    if no data found, popup an Info box.
    Send to InfoBox the done message.
    remove busy cursor
```

12.7.2.

loadJobs()

```
loadJobs{}
```

Control Flow:

Clear the entries of Jobs Listbox.
set jobsSelMode to "Select All"
set jobsSelected to none

Load values of jobinfo, whose indices match any of queuesSelected, into Jobs listbox.

12.7.3.

loadQueues()

```
loadQueues{}
```

Control Flow:

Clear the entries of Queues Listbox.
set queuesSelMode to "Select All"
set queuesSelected to none

Clear the entries of Jobs Listbox.
set jobsSelMode to "Select All"
set jobsSelected to none

Load values of qinfo, whose indices match any of hostsSelected, into Queues listbox.

12.7.4.

getHostsDetail()

```
getHostsDetail{}
```

Control Flow:

if hostsSelected isEmpty; then
  popup InfoBox asking user to select a host
endif
run PBS_HOSTS_DETAIL_CMD on hostsSelected

12.7.5.

getQueuesDetail()

```
getQueuesDetail{}
```

Control Flow:
if queuesSelected isEmpty; then
popup InfoBox asking user to select a queue
endif
run PBS_QUEUES_DETAIL_CMD on queuesSelected

12.7.6.

getJobsDetail()

```
getJobsDetail{}
```

Control Flow:
if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
run PBS_HOSTS_DETAIL_CMD on jobsSelected

12.7.7.

runDelete()

```
runDelete{}
```

Control Flow:
if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
call qdel procedure

12.7.8.

runHold()

```
runHold{}
```

Control Flow:
if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
call qhold procedure

12.7.9.

runRelease()

```
runRelease{}
```

Control Flow:

if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
call qrls procedure

12.7.10.

runRerun()

```
runRerun{}
```

Control Flow:

if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
run cmdpath(QRERUN) on jobsSelected
get new data for jobs only

12.7.11.

runRun()

```
runRun{}
```

Control Flow:

if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
run cmdpath(QRUN) on jobsSelected
get new data for jobs only

12.7.12.

runQsig()

```
runQsig{}
```

Control Flow:

if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
call qsig procedure

12.7.13.

runQmsg()

```
runQmsg{}
```

Control Flow:
    if jobsSelected isEmpty; then
      popup InfoBox asking user to select a job
    endif
    call qmsg procedure

12.7.14.

runQmove()

```
runQmove{}
```

Control Flow:
    if jobsSelected isEmpty; then
      popup InfoBox asking user to select a job
    endif
    call qmove procedure

12.7.15.

runQstop()

```
runQstop{}
```

Control Flow:
    if queuesSelected isEmpty; then
      popup InfoBox asking user to select a queue
    endif
    run cmdpath(QSTOP) on queuesSelected
    get new data

12.7.16.

runQstart()

```
runQstart{}
```

Control Flow:
    if queuesSelected isEmpty; then

popup InfoBox asking user to select a queue
endif
run cmdpath(QSTART) on queuesSelected
get new data

12.7.17.

runQenable()

```
runQenable{}
```

Control Flow:
if queuesSelected isEmpty; then
popup InfoBox asking user to select a queue
endif
run cmdpath(QENABLE) on queuesSelected
get new data

12.7.18.

runQdisable()

```
runQdisable{}
```

Control Flow:
if queuesSelected isEmpty; then
popup InfoBox asking user to select a queue
endif
run cmdpath(QDISABLE) on queuesSelected
get new data

12.7.19.

runQalter()

```
runQalter{}
```

Control Flow:
if jobsSelected isEmpty; then
popup InfoBox asking user to select a job
endif
call qalter procedure

12.7.20.

runQorder()

```
runQorder{}
```

Control Flow:
if # of jobsSelected is != 2 ; then
popup InfoBox asking user to select 2 jobs
endif
run cmdpath (QORDER) on jobsSelected
get new data for jobs only

12.7.21.

runQterm()

```
runQterm{}
```

Control Flow:
if hostsSelected isEmpty; then
popup InfoBox asking user to select a host
endif
call qterm procedure
get new data

12.7.22.

runQsub()

```
runQsub{}
```

Control Flow:
if hostsSelected isEmpty ; then
popup InfoBox asking user to select ONE host
elseif # of hostsSelected != 1; then
bring up a bridge window asking users to limit selection
endif
call qsub procedure

12.7.23.

build_opt()

```
build_opt{cmdline {pbsdir "#PBS"} {do_qalter 0}}
```

Args:

cmdline — a boolean value where if set to 1 means command line options will be returned; otherwise, PBS directive lines will be returned as a string.

pbsdir — the PBS directive line parse string if **cmdline** is set to 1.

do_qalter — a boolean value (0 or 1) signalling options to be build is for the qalter command instead of qsub.

Returns:

The command line options string (if cmdline set to 1); otherwise, returns the PBS directive lines.

Control Flow:

```
if do_qalter ; then
 option_value will be taken from qalterv (input_array)
 the current_dialog_box is ".qalter"
 set the default array to "def_qalter"
else
 option_value will be taken from qsubv (input_array)
 the current_dialog_box is ".qsub"
 set the default array to "def_qsub"
endif

Foreach option_name in options
do
 get option_name's dialog boxes location
 get option_name's option letter
 get option_name's special instructions

 if current_dialog_box is one of option_line's dialog boxes AND
     option_value is found and notEmpty and not match option_name's default
       value
 then
  if option_name's special instruction says the option is a toggle (meaning
    no arguments), then
      create an opt array entry with index "option letter" for option_name,
       and value "option_value".
  else
      create an opt array entry (if one doesn't exist) with index
       "option letter" for option_name and value "option_value".
     if opt array entry already exists, prefix with a "," and append
       "option_value"
  endif
 endif

 NOTE: opt is an array whose element names correspond to option letters, and
                  element values correspond to actual arguments to
                    option letters.

Build the options commandline or directive lines depending on whether or not
 cmdline is set, using the opt array.
```

12.7.24.

set_opt_default()

```
set_opt_default{array {do_main 1}    {do_depend 0} {do_staging 0} {do_misc 0}
                      {do_qalter 0} {do_email 0}}
```

Args:

array	the array to be loaded with default widget values.
do_main	a boolean value that says set widget values in main window.
do_depend	a boolean value that says set widget values in job dependency window.
do_staging	a boolean value that says set widget values in file staging window.
do_misc	a boolean value that says set widget values in misc window.
do_qalter	a boolean value that says the current dialog box is qalter
do_email	a boolean value that says set widget values in email dialog box.

Returns:

The array values corresponding to widget defaults.

Control Flow:

Foreach widget_variable in default
do
  set array values with widget values depending on which group of widgets
    were instructed to be set.
done

12.7.25.

load_qsub_input()

```
load_qsub_input{fd}
```

Args:

fd	the file stream descriptor containing "job_attribute = value" data lines.

Returns:

Zero if qsubv array was successfully loaded with widget values; 1 otherwise.

Control Flow:

Initialize attrlist, queueName, and serverName buffers

Foreach data received in fd
do
  skip empty data
  get attribute and value parts of the data
  set appropriate entries in qsubv depending on the attribute value
done

Special processing:

  Break up resource attribute list
  Compare the queue_name read from data with the entries in Submit window's
    destination listbox.

- if queue_name doesn't start with "@", highlight/select an entry
in the destination listbox that matches "<queue_name>" (exactly) or
"<queue_name>@".
- if queue_name starts with "@", then we have an @serverName
specification. Highlight/select an entry in the destination listbox
that begins with the string specified by <queue_name>
For example, "fast" will match "fast" and "fast@vn" entries, but it will
not match "fast1" or "fast1@vn". "@vn" will match "@vn.nas.nasa.gov"
or even "@vn.larc.nasa.gov".
Now adjust the view of the destination listbox so that the selected entry can
be seen.

close the file stream descriptor

12.7.26.

oper()

```
oper{operator}
```

Args:

operator the operator string - should be one of "=", "!=", ">=", ">", "<=", "<"

Returns:

the comparison operator string used in PBS select command. Returns empty string if the mapping could not be found.

Control Flow:

```
case "="  return ".eq."
   "!="   return ".ne."
   ">="   return ".ne."
   ">"    return ".gt."
   "<="   return ".le."
   "<"    return ".lt."
   *other* return ""
```

12.7.27.

oper_invert()

```
oper_invert {str}
```

Args:

str the operator string - should be one of "eq", "ne", "ge", "gt", "le", "lt"

Returns:

the comparison operator display string used in the GUI. Returns empty string if the mapping could not be found.

Control Flow:

```
case "eq" return "="
   "ne"   return "!="
```

```
"ge"    return ">="
"gt"    return ">"
"le"    return "<="
"lt"    return "<"
*other*  return ""
```

12.7.28.

cvtdatetime_arg()

```
cvtdatetime_arg{mon day year hh mm ss}
```

Args:

mon	the month string: Jan - Dec
day	the day string
year	the year string
hh	the hour string
mm	the minute string
ss	the seconds string

Returns:
the date/time argument string suitable for PBS commands execution.

Control Flow:
Map month string to a corresponding numeric value
Prefix day with a 0 if day value is between 0 and 9
Prefix hour with a 0 if hour value is between 0 and 9
Prefix minute with a 0 if minute value is between 0 and 9
Prefix seconds with a 0 if seconds value is between 0 and 9

12.7.29.

build_sel_options()

```
build_sel_options{}
```

Returns:
the options string suitable for the PBS qselect command.

Control Flow:
Get option names from the select_opt array
Get option values from the selv array

Append to options_list:
(1) the option_letter as obtained from select_opt array
(2) the option_value if it is not empty and not "-ANY".

Return the options_list

12.7.30.

xpbs_help()

```
xpbs_help{help_category callerDialogBox}
```

Args:

help_category	the kind of help text to exhibit. Help files are named as 'help_category'.hlp
callerDialogBox	the widget path of the calling dialog box. This is so that we can put back the input focus to the originating dialog box.

Control Flow:

run a "cat <helpdir>/<help_category>.hlp" with output going to an Info box.

12.7.31.

resources_help()

```
resources_help {callerDialogBox, suffix}
```

Args:

callerDialogBox	the widget path of the calling dialog box. This is so that we can put back the input focus to the originating dialog box.
suffix	a help page suffix.

Control Flow:

run a "man pbs_resources_$suffix | col -b" with output going to an Info box.

12.7.32.

set_default_qsub_main()

```
set_default_qsub_main {}
```

Control Flow:

set main submit window-associated array elements of qsubv to default

12.7.33.

init_qsub_main_argstr()

```
init_qsub_main_argstr{}
```

Control Flow:

set to default the argument strings associated with dSubmit main window using the array elements found in qsubv

12.7.34.

set_default_qsub_depend()

`set_default_qsub_depend {}`

Control Flow:

set depend window-associated array elements of qsubv to default

12.7.35.

init_qsub_depend_argstr()

`init_qsub_depend_argstr{}`

Control Flow:

set to default the argument strings associated with depend using the array elements found in qsubv

12.7.36.

set_default_qsub_staging()

`set_default_qsub_staging {}`

Control Flow:

set file staging window-associated array elements of qsubv to default

12.7.37.

init_qsub_staging_argstr()

`init_qsub_staging_argstr{}`

Control Flow:

set to default the argument strings associated with file staging using the array elements found in qsubv

12.7.38.

set_default_qsub_misc()

`set_default_qsub_misc {}`

Control Flow:

set misc window-associated array elements of qsubv to default

12.7.39.

init_qsub_misc_argstr()

`init_qsub_misc_argstr{}`

Control Flow:

set to default the argument strings associated with misc using the array elements found in qsubv

12.7.40.

set_default_qsub_datetime()

`set_default_qsub_datetime {}`

Control Flow:

set datetime window-associated array elements of qsubv to default

12.7.41.

init_qsub_datetime_argstr()

`init_qsub_datetime_argstr{}`

Control Flow:

set to default the argument strings associated with datetime using the array elements found in qsubv

12.7.42.

set_default_qsub_email()

`set_default_qsub_email {}`

Control Flow:

set email addresses window-associated array elements of qsubv to default

12.7.43.

init_qsub_email_argstr()

`init_qsub_email_argstr{}`

Control Flow:

set to default the argument strings associated with email addresses using the array elements found in qsubv

12.7.44.

set_default_qalter_main()

```
set_default_qalter_main {}
```

Control Flow:

set main qalter window-associated array elements of qalterv to default

12.7.45.

init_qalter_main_argstr()

```
init_qalter_main_argstr{}
```

Control Flow:

set to default the argument strings associated with main qalter window using the array elements found in qalterv

12.7.46.

set_default_qalter_depend()

```
set_default_qalter_depend {}
```

Control Flow:

set qalter depend window-associated array elements of qalterv to default

12.7.47.

init_qalter_depend_argstr()

```
init_qalter_depend_argstr{}
```

Control Flow:

set to default the argument strings associated with depend using the array elements found in qalterv

12.7.48.

set_default_qalter_staging()

```
set_default_qalter_staging {}
```

Control Flow:

set staging window-associated array elements of qalterv to default

12.7.49.

init_qalter_staging_argstr()

`init_qalter_staging_argstr{}`

Control Flow:

set to default the argument strings associated with file staging using the array elements found in qalterv

12.7.50.

set_default_qalter_misc()

`set_default_qalter_misc {}`

Control Flow:

set misc window-associated array elements of qalterv to default

12.7.51.

init_qalter_misc_argstr()

`init_qalter_misc_argstr{}`

Control Flow:

set to default the argument strings associated with misc widgets using the array elements found in qalterv

12.7.52.

set_default_qalter_datetime()

`set_default_qalter_datetime {}`

Control Flow:

set datetime window-associated array elements of qalterv to default

12.7.53.

init_qalter_datetime_argstr()

`init_qalter_datetime_argstr{}`

Control Flow:

set to default the argument strings associated with datetime widgets using the array elements found in qalterv

12.7.54.

set_default_qalter_email()

`set_default_qalter_email {}`

Control Flow:

set email window-associated array elements of qalterv to default

12.7.55.

init_qalter_email_argstr()

`init_qalter_email_argstr{}`

Control Flow:

set to default the argument strings associated with email widgets using the array elements found in qalterv

12.7.56.

set_pbs_options()

`set_pbs_options {}`

Control Flow:

sets the various PBS options for qsub, qalter, and qselect

12.7.57.

set_pbs_defaults()

`set_pbs_defaults {}`

Control Flow:

set various widget defaults using the arrays def_qsub and def_qalter

12.7.58.

about()

`about {}`

Control Flow:

popup the "About.." dialog box
display the XPBS logo.
create the message widget
display both the image and the message.

12.8. File: common.tk

This file contains various routines that don't quite fit in the framework of the other "files".

12.8.1.

lintmax()

```
lintmax {intlist}
```

Args:

intlist a positive list of integers

Returns

element with the maximum value; -1 if errors are detected.

Control Flow:

sort the integer list from highest value to smallest value
return the highest value

12.8.2.

lintmin()

```
lintmin {intlist}
```

Args:

intlist a positive list of integers

Returns

element with the minimum value; -1 if errors are detected.

Control Flow:

sort the integer list from smallest value to highest value
return the smallest value

12.8.3.

popupDialogBox()

```
popupDialogBox{dialog_top title {grab_window 1} {class ""}}
```

Args:

dialog_top	frame of the dialog box to bring up
title	title to be given to the dialog box
grab_window	boolean value (0 or 1) that says set a grab on the dialog box.
class	class name to associate with the dialog box.

Returns

the top and bottom frames of the dialog box.

Control Flow:

Create a toplevel window called 'dialog_top' for dialog box toplevel. Declare it with 'class' if class name is given.
Give a title text to the dialog box.
if grab_window then set a grab on the dialog box.
Set focus on the dialog box.
create the top and bottom frames of the dialog box and make them visible.
Return the top and bottom frames.

12.8.4.

win_cmdExec()

```
win_cmdExec{callerDialogBox command_list}
```

Args:

callerDialogBox	dialog box that called this procedure. This is for setting the input focus back to the calling dialog box after running this routine.
command_list	a command to execute.

Returns

the exit code of the executed command.

Control Flow:

set busy_cursor
Send a message to InfoBox regarding execution of 'command_list'.
Execute 'command_list'.
Popup an error dialog box if command execution failed; otherwise, popup an output box.
After command execution, send message to InfoBox about its completion and remove busy_cursor.

12.8.5.

busy_cursor()

```
busy_cursor{}
```

Control Flow:

Look for an active window by checking the **activeWindow** array.
Configure the active window's mouse cursor to be an hourglass.

12.8.6.

remove_busy_cursor()

```
remove_busy_cursor{}
```

Control Flow:

Look for an active window by checking the **activeWindow** array.
Configure the active window's mouse cursor to go back to the default image.

12.8.7.

cmdExec()

```
cmdExec{command_list {popupError 0} {retcode_only 0}}
```

Args:

command_list	a command to execute.
popupError	boolean value (0 or 1) that says to popup an error box if an error was encountered.
retcode_only	boolean value (0 or 1) that says to return the exit code of the command instead of its output.

Returns

the output of the command unless exit code is specified (see retcode_only argument).

Control Flow:

set busy_cursor
Send message to InfoBox regarding start of **command_list** execution.
Execute **command_list**.
If an error occurred during command execution, popup an error dialog box.
Send message to InfoBox regarding command execution completion.
remove_busy_cursor
return output of the command if **retcode_only** is not set; otherwise,
return exitcode

12.8.8.

InfoBox_sendmsg()

```
InfoBox_sendmsg { message {line_number 1} {append 0} {xview_increment 1} }
```

Args:

message	message to send to InfoBox.
line_number	the line_number entry of the InfoBox (relative to 0) where message will be inserted.
append	boolean value (0 or 1) that says to append the message to an existing **line_number**.
xview_increment	how much to right shift the horizontal display of the InfoBox.

Control Flow:

if not append
insert new message at 'line_number'
delete previous message at 'line_number' if more than one entries are found
and 'line_number' is < InfoBox size.
if append

get the previous message at 'line_number'
insert the previous message and the new message to 'line_number'
delete the previous message if there are more than one entries in InfoBox
and 'line_number' is < InfoBox size.

Adjust the horizontal view of the listbox so that the command at the end
is visible. For Tk versions < 4.0, make use of the 'xview_increment' variable.

12.8.9.

cmdExec_bg()

```
cmdExec_bg{command_list {popupError 0}}
```

Args:

command_list	the command to execute in the background.
popupError	boolean value (0 or 1) that says to popup an error dialog box when command execution has encountered an error.

Returns
a stream descriptor where input can be read.

Control Flow:
Use Tcl pipe mechanism to execute 'command_list'.
Return pipe stream descriptor.

12.8.10.

popupOutputBox()

```
popupOutputBox{output}
```

Args:

output	output message string to display.

Control Flow:
Create the Ouput dialog box.
Populate the top frame with a read-only text box widget. Also create a
vertical scrollbar for the text widget.
Make text widget and scrollbar visible.
Populate the bottom frame with an 'ok' binding.
Insert 'output' to the text widget.
Bind <Return> key to 'ok'.

12.8.11.

popupErrorBox()

```
popupErrorBox {retcode command_list errmsg {width_pixels 500}
```

```
                {callerDialogBox ""} }
```

Args:

retcode	exit code of the executed command.
command_list	command string that was executed.
errmsg	error message resulted from command execution.
width_pixels	the width of the error dialog box.
callerDialogBox	the name of the calling dialog box.

Control Flow:

Create the Error dialog box.
Populate the top frame with a message widget containing information about exit code, error message, and the name of the command executed.
Make message widget visible.
Populate the bottom frame with an 'ok' binding.
Bind <Return> key to 'ok'.
wait for the window to be destroyed before setting the focus back to the calling dialog box.

12.8.12.

popupInfoBox()

```
popupInfoBox {callerDialogBox msg {width_pixels 500} {focusBox}}
```

Args

callerDialogBox	dialog box that called this routine.
msg	the message string to display.
width_pixels	the width of the error dialog box.
focusBox	a widget name

Control Flow:

Create the Info dialog box.
Populate the top frame with a message widget containing the 'msg' string.
Make message widget visible.
Populate the bottom frame with an 'ok' binding.
Bind <Return> key to 'ok'.
wait for the window to be destroyed before setting the focus back to 'focusBox'.

12.8.13.

create_DateTime_box()

```
create_DateTime_box {frame_name def_mon def_day def_yr def_hr
                     def_min def_sec ARR }
```

Args:

frame_name	frame to place the date/time spin boxes.
def_mon	default month entry value.
def_day	default day entry value.
def_yr	default year entry value.
def_hr	default hour entry value.
def_min	default minutes entry value.
def_sec	default seconds entry value.
ARR	array holding the entry text variables associated with the various spin-boxes. This array will hold the elements: qtimeMon, qtimeDay, qtimeYear, qtimeHH, qtimeMM, and qtimeSS.

Returns

the frame_name, month, day, year, hour, minute, seconds entry widget names, list of scrollbars used, and list of labels used.

Control Flow:

Create the frames to hold the spinbox entries, labels, and scrollbars.
Create the necessary labels.
Make the frames and labels visible.
Create the 6 spinboxes, specifying the appropriate default entry values.
Return widget names involved.

12.8.14.

disable_label()

```
disable_label {label_name color_name}
```

Args:

label_name	label widget name to disable.
color_name	color of the label widget when disabled.

Control Flow:

Save the current color of the label widget to the global selColor array.
Configure the label widget to have 'color_name'.

12.8.15.

enable_label()

```
enable_label{label_name}
```

Args:

label_name	label widget name to enable.

Control Flow:

Get the active color of the label widget from the selColor array.
Configure the label widget to have the active color.

12.8.16.

disable_scrollbar()

```
disable_scrollbar{scrollbar color_name}
```

Args:

scrollbar	scrollbar widget name to disable.
color_name	color of the scrollbar widget when disabled.

Control Flow:

Save the current background/trough color, foreground color, and active foreground colors of the scrollbar widget.
If widget is already disabled, then return
Configure the scrollbar widget so that background/trough, foreground, and active foreground colors are all set to 'color_name'.

12.8.17.

enable_scrollbar()

```
enable_scrollbar{scrollbar}
```

Args:

scrollbar	scrollbar widget name to enable.

Control Flow:

Get the current background/trough color, foreground color, and active foreground colors of the scrollbar widget from selColor.
Configure the scrollbar widget so that background/trough, foreground, and active foreground colors are all set to the active colors.

12.8.18.

disable_dateTime()

```
disable_dateTime {entryList scrollList labelList}
```

Args:

entryList	a list of date/time entry widgets.
scrollList	a list of date/time scrollbar widgets.
labelList	a list of date/time label widgets.

Control Flow:

Get the disable color from the global variable/resource disabledColor.
Disable the spinbox entries.
Disable the spinbox scrollbars.
Disable the spinbox labels.

12.8.19.

enable_dateTime()

```
enable_dateTime {entryList scrollList labelList}
```

Args:

entryList	a list of date/time entry widgets.
scrollList	a list of date/time scrollbar widgets.
labelList	a list of date/time label widgets.

Control Flow:

Enable the spinbox entries.
Enable the spinbox scrollbars.
Enable the spinbox labels.

12.8.20.

construct_array_args()

```
construct_array_args {arr sep {header_str ""} }
```

Args:

arr	the array the construct an argument string.
sep	the separator of array elements in the resulting argument string.
header_str	a leading string to insert in the resulting argument string.

Returns

a string containing the elements of 'arr' separated by 'sep'.

Control Flow:

Initialize the return string.
Cycle through the sorted elements of 'arr', ignoring empty slots, and construct the return string.
Return the resulting string.

12.8.21.

deconstruct_array_args()

```
deconstruct_array_args {arr_str arr sep {header_str_to_ignore ""} }
```

Args:

arr_str	the argument string to deconstruct.
arr	the array string to construct.
sep	the separator of array elements in the resulting argument string.

header_str_to_ignore a leading string to ignore when scanning 'arr_str' for input to 'arr'.

Returns

the array 'arr' containing the characters in the string 'arr_str' that appear in between 'sep'.

Control Flow:

Parse through 'arr_str' looking for characters in between 'sep' and not matching 'header_str_to_ignore'. Fill the array 'arr' with the characters found.

12.8.22.

set_dateTime()

```
set_dateTime { m d y hour min sec {default 0} {valuestr ""} }
```

Args:

m	a month spinbox entry textvariable.
d	a day spinbox entry textvariable.
y	a year spinbox entry textvariable.
hour	a hour spinbox entry textvariable.
min	a minutes spinbox entry textvariable.
sec	a seconds spinbox entry textvariable.
default	boolean value (0 or 1) that says to set values of textvariables to default.
valuestr	a string of numbers to set textvariables to.

Control Flow:

Get the date valuestr.
Set month, day, year, hour, minutes, and seconds textvariable values to:
(1) default (Jan 1 1970 00:00:00) if 'default' is set, or if unable to obtain the current date/time string.
(2) current date/time string.
For day, hour, minute, seconds values that fall between 0 and 9, make sure they are prefixed with a 0.

12.8.23.

digit()

```
digit{number_str}
```

Args:

number_str a number string to convert to a digit.

Returns

the digit part of a number string (i.e. 08 returns 8; 9 returns 9, etc...)

Control Flow:

Use of a case/switch statement should be sufficient with a regular expression

match on the number string.

12.8.24.

packinfo()

```
packinfo {slave}
```

Args:

slave slave widget to return packing information.

Returns

pack information for slave widget.

Control Flow:

Return the output from "pack info (or newinfo for Tk < 4.0)".

12.8.25.

clear_array()

```
clear_array {array_name}
```

Args:

array_name name of an array

Returns

the array 'array_name' whose elements are empty strings.

Control Flow:

Cycle through the elements of 'array_name', and resetting to empty string each element's value.

12.8.26.

load_argstr()

```
load_argstr {arr_str outersep arraylist innersep_list {header_str_to_ignore}}
```

Args:

arr_str	string containing tokens that will populate 'arraylist'.
outersep	char separating tokens in 'arr_str'.
array_list	list of arrays to be loaded with tokens found 'arr_str'.
innsersep_list	list of characters to separate sub-tokens within each token.
header_str_to_ignore	string at the beginning of 'arr_str' to ignore.

Control Flow:

Get all the tokens separated by 'outersep' in 'arr_str'
For each token ; do

Get the subtokens separated by 'innersep_list'.
Copy each subtoken to corresponding elements in 'arraylist'.
Cycle through the elements of 'array_name', and resetting to empty string each element's value.

12.9. File: button.tk

This file contains routines that are related to button widgets.

12.9.1.

buildCheckboxes()

```
buildCheckboxes {frame_name checkbutton_list orient spacing
                 {checkbox_labelstr ""} {place_buttonAssoc_right 1}
                 {place_labelstr_top 0} {button_assoc_groove_relief 0} }
```

Args:

frame_name	name of the frame where a collection of checkbuttons will be placed.
checkbutton_list	a list of checkbutton grouping specifications. A grouping specification is in itself another list where each entry specifies the name for the button, the label, and any outside widget association. Think of this as 2-deep: list of lists. { { {group1_button1Name group1_button1Label group1_button1Assoc} {group1_button2Name group1_button2Label group1_button2Assoc} {group1_button3Name group1_button3Label group1_button3Assoc} ... } { {group2_button1Name group2_button1Label group2_button1Assoc} {group2_button2Name group2_button2Label group2_button2Assoc} ... } ... }
orient	how the collection of buttons will be grouped - in a "column" or "grid".
spacing	distance between checkbuttons (in units or pixels)
checkbox_labelstr	single label string describing the collection of checkbuttons.
place_buttonAssoc_right	boolean value (0 or 1) that says to place the *button*Assoc to the right side of a check button instead of at the top.
place_labelstr_top	boolean value (0 or 1) that says to place the checkbox_labelstr at the top instead of at the left side.

button_assoc_groove_relief — boolean value (0 or 1) that says to enclose *button*_Assoc in a groove relief.

Returns

a list: frame_name, and widget names for all checkbuttons in the order specified in 'checkbutton_list'.

Control Flow:

```
Create and make visible the single checkbox label string. Place the string
either at the top or at the left side depending on whether or
not ' place_labelstr_top' is set.
Foreach group of declared checkbuttons in 'checkbutton_list'; do
 create and make visible the frame for each group
 foreach button in  each group; do
  if there's an associated widget for a button, then
   create a frame that will hold the button and the associated widget
   create the button
   make the button and the associated widget visible.
   append to buttonList the frame name holding the button and the associated
      widget.
  else
   create the button
   append to buttonList the name of the button
  endif
 done
done
pack buttonList
return list of widget names involved.
```

12.9.2.

buildCmdbuttons()

```
buildCmdButtons {frame_name cmdbutton_list orient spacing button_width
                 button_height {first_group_spacing 0}
                 {spread_out_buttons 1} {group_spacing "10m"}
                 {cmdButton_labelstr ""}}
```

Args:

frame_name — name of the frame where a collection of command buttons will be placed.

cmdbutton_list — a list of command button grouping specifications. A grouping specification is in itself another list where each entry specifies the name for the button, the label, and any outside widget association. Think of this as 2-deep: list of lists.

```
{ { {group1_button1Name group1_button1Label group1_button1Assoc}
   {group1_button2Name group1_button2Label group1_button2Assoc}
   {group1_button3Name group1_button3Label group1_button3Assoc}
   ...
  }
```

{ {group2_button1Name group2_button1Label group2_button1Assoc}
{group2_button2Name group2_button2Label group2_button2Assoc}
...
}
...
}

orient	how the collection of buttons will be grouped - in an "x" (horizontal), "y" (vertical), "xy" (gridded).
spacing	distance between command buttons (in units or pixels)
button_width	width for all buttons (in units or pixels)
button_height	height for all buttons (in units or pixels)
first_group_spacing	bolean value (0 or 1) that says to put a 'button_width' spacing before the 1st group of command buttons.
spread_out_buttons	bolean value (0 or 1) that says to spread out the distribution of the command buttons in 'frame_name'.
group_spacing	distance between groups of command buttons (in units or pixels)
cmdButton_labelstr	single label string describing the collection of command buttons.

Returns

a list: frame_name, and widget names for all command buttons in the order specified in 'checkbutton_list'. It will also return the single label string for the command buttons if one exists.

Control Flow:

Create and make visible the single command button label string.
Foreach group of declared command buttons in 'cmdbutton_list'; do
create the frame for each group
create the command buttons
append to buttonList the name of the buttons
done
Make visible all command buttons (i.e. pack buttonList)
Pack the frames for each group as follows:
(1) if first_group_spacing, puts an extra spacing before 1st group
(2) Put an amount of 'group_spacing" in between each group.
(3) Pack the frames in such a way that the correct orientation is followed.
return list of widget names involved.

12.9.3.

buildRadioboxes()

```
buildRadioboxes {frame_name radiobutton_list orient spacing
                 {radiobox_labelstr ""} {place_buttonAssoc_right 1}
                 {place_labelstr_top 0} {button_assoc_groove_relief 0} }
```

Args:

frame_name	name of the frame where a collection of radiobuttons will be placed.

radiobutton_list	a list of radiobutton grouping specifications. A grouping specification is in itself another list where each entry specifies the name for the button, the label, and any outside widget association. Think of this as 2-deep: list of lists. { { {group1_button1Name group1_button1Label group1_button1Assoc} {group1_button2Name group1_button2Label group1_button2Assoc} {group1_button3Name group1_button3Label group1_button3Assoc} ... } { {group2_button1Name group2_button1Label group2_button1Assoc} {group2_button2Name group2_button2Label group2_button2Assoc} ... } ... }
orient	how the collection of buttons will be grouped - in a "column" or "grid".
spacing	distance between radiobuttons (in units or pixels)
radiobox_labelstr	single label string describing the collection of radiobuttons.
place_buttonAssoc_right	boolean value (0 or 1) that says to place the *button*Assoc to the right side of a radio button instead of at the top.
place_labelstr_top	boolean value (0 or 1) that says to place the radiobox_labelstr at the top instead of at the left side.
button_assoc_groove_relief	boolean value (0 or 1) that says to enclose *button*_Assoc in a groove relief.

Returns

a list: frame_name, and widget names for all radiobuttons in the order specified in 'checkbutton_list'.

Control Flow:

```
Create and make visible the single radiobox label string. Place the string
either at the top or at the left side depending on whether or
not ' place_labelstr_top' is set.
Foreach group of declared radiobuttons in 'radiobutton_list'; do
 create and make visible the frame for each group
 foreach button in  each group; do
  if there's an associated widget for a button, then
   create a frame that will hold the button and the associated widget
   create the button
   make the button and the associated widget visible.
   append to buttonList the frame name holding the button and the associated
      widget.
  else
   create the button
   append to buttonList the name of the button
```

```
    endif
  done
done
pack buttonList
return list of widget names involved.
```

12.9.4.

disable_rcbutton()

```
disable_rcbutton{button_name}
```

Args:

button_name name of a radiobutton or checkbutton to disable.

Control Flow:

If button already disabled then skip the rest of this routine
Save the active color of the button in the global variable selColor.
Configure the button to be in a disbled state with color of the global variable disabledColor.

12.9.5.

enable_rcbutton()

```
enable_rcbutton{button_name}
```

Args:

button_name name of a radiobutton or checkbutton to enable.

Control Flow:

If can't find an active color for the button or the state of the button is already normal, then
skip the rest of this routine
Configure the button to be in a normal state with an active color taken from the global variable selColor.

12.9.6.

disable_button()

```
disable_button{button_name}
```

Args:

button_name name of a command button to disable.

Control Flow:

If command button already disabled, then skip the rest of this routine.
Configure the state of this button to be disabled.

12.9.7.

enable_button()

```
enable_button{button_name}
```

Args:

button_name name of a command button to enable.

Control Flow:

Configure the state of this button to be normal.

12.9.8.

disable_rcbuttons()

```
disable_rcbuttons{args}
```

Args:

args a list of radiobuttons/checkbuttons to disable.

Control Flow:

For each button in the argument list, call disable_rcbutton.

12.9.9.

enable_rcbuttons()

```
enable_rcbuttons{args}
```

Args:

args a list of radiobuttons/checkbuttons to enable.

Control Flow:

For each button in the argument list, call enable_rcbutton.

12.9.10.

invoke_rbutton()

```
invoke_rbutton {buttons}
```

Args:

buttons a list of radiobuttons to invoke if set.

Control Flow:

Foreach buttons; do

if button's value correspond to the ON state, then invoke button.
done

12.9.11.

invoke_cbutton()

```
invoke_cbutton {buttons}
```

Args:

buttons a list of checkbuttons to invoke if set.

Control Flow:

Foreach buttons; do
Force the button's state to be normal.
invoke button
if previous state of the button is disabled ; then
disable the button again
done

12.10. File: entry.tk

This file contains routines that are related to entry widgets.

12.10.1.

buildFullEntrybox()

```
buildFullEntrybox {frame_name label_width label entry_width
                   default_entryval scrollbar_placement all_button
                   {label_placement "left"}}
```

Args:

frame_name	the frame where a full entry widget will be placed.
label_width	the width (in units or pixels) for a full entry widget label.
label	the text for the full entry widget label.
entry_width	the width (in units or pixels) for an entry widget.
default_entryval	the default value for an entry widget.
scrollbar_placement	the location of the entry widget's scrollbar - left, right, bottom, top of the entry widget.
all_button	boolean value (0 or 1) that says to also include special "all" button.
label_placement	specifies where to place the accompanying label for the entry widget - left, right, bottom, top of the entry widget.

Returns

a list: frame_name, "all" button widget name, entry widget name, and scrollbar.

Control Flow:

Create the "all" button if 'all_button' is set.

Create the label for the entry widget.
Create the entry widget.
Insert a default value for the entry widget.
Create a scrollbar and associate it with the entry widget.
Make label, scrollbar, and entry widget visible by paying close attention to
scrollbar_placement, and label_placement.
return names of all widgets involved.

12.10.2.

compress_array()

```
compress_array {array_name}
```

Args:

array_name	an array to compress

Returns:
size of the newly-compressed array.

Control Flow:
Initialized the 'next' slot to be filled with non-empty data.
Go through all elements of 'array_name',
if element is non-empty,
insert element to the 'next' slot in array2.
increment 'next'
endif

Unset 'array_name'
Copy back all elements of 'array2' back to 'array_name' with all non-empty
entries now removed.
Return the size of the new array.

12.10.3.

disable_fullentry()

```
disable_fullentry{entryLabel entry entryScroll {allButton ""}}
```

Args:

entryLabel	label for an entry widget.
entry	entry widget to disable.
entryScroll	scrollbar for entry widget.
allButton	the special "all" for an entry widget.

Control Flow:
Disable entryLabel using disabledColor as color.
Disable scrollbar using disabledColor as color.
Disable the "all" button.
If the entry widget is not yet disabled, then

save the foreground color information of the entry widget to selColor
set the entry widget's state to disabled, and foreground color to
disabledColor
endif

12.10.4.

enable_fullentry()

```
disable_fullentry{entryLabel entry entryScroll {allButton ""}}
```

Args:

entryLabel	label for an entry widget.
entry	entry widget to enable.
entryScroll	scrollbar for entry widget.
allButton	the special "all" for an entry widget.

Control Flow:

Enable entryLabel.
Enable scrollbar.
Enable the "all" button.
Configure the state of the entry widget's state to normal, and set foreground color to active color.
endif

12.11. File: listbox.tk

This file contains routines that are related to listbox widgets.

12.11.1.

buildFullListbox()

```
buildFullListbox {frame_name ColxRow header_str scrollbarType
                  {all_button 1} {header_at_left 0} }
```

Args:

frame_name	the name of the frame widget to place the complete listbox.
ColxRow	# of columns (characters) and # of rows on the listbox.
header_str	the header string of the listbox.
scrollbarType	specifies the orientation of the listbox's scrollbar. Can only have the values: xscroll, yscroll, xyscroll, noscroll.
all_button	boolean value (0 or 1) that says to include the "Select All/Deselect All" buttons.
header_at_left	boolean value (0 or 1) that says to have the header to be at the left side of the listbox instead of at the top.

Returns

list: frame_name, label, all button, listbox widget name, and scrollbars.

Control Flow:

Create listbox header label if header_str is not empty. Make the header visible.
Create and make visible the "Select All/Deselect All" button if all_button is set. Make sure to orient header according to instructions from header_at_left

Create the listbox widget.
Create scrollbar and associate with listbox.
Make both listbox and scrollbar visible.

return widgets that have been created.

12.11.2.

get_keyvals()

```
get_keyvals{lbox key_list inner_sep outer_sep {type "all"}}
```

Args:

lbox	name of a listbox to get key values from.
key_list	list of field indices to listbox.
inner_sep	the string to put within key values of an entry in the resulting string.
outer_sep	the string to put between entries of the listbox in the resulting string.
type	choice value of either "select" or "all" referring to what type of listbox entries will be matched for key values.

Returns

a string containing selected field values of entries in listbox. 'inner_sep' separates field values, while entries are separated by 'outer_sep'.

Control Flow:

Get the listbox indices to extract field values from.
Extract the field values.
Construct the return string by separating field values with 'inner_sep', and separating each listbox entry with 'outer_sep'.
Remove any trailing 'outer_sep'.
Return key values string.

12.11.3.

strget_keyvals()

```
strget_keyvals{str keylist sep}
```

Args:

str	string to extract key values from.
keylist	list of field indices to str.
sep	the string to put between key values.

Returns
a string containing selected field values of string 'str', separated by 'sep'.

Control Flow:
foreach field in key_list
do
Extract the field values.
Construct the return string by separating field values with 'sep'.
done
Return key values string.

12.12. File: spinbox.tk

This file contains routines supporting the spinbox widget.

12.12.1.

buildSpinbox()

```
buildSpinbox{ frame_name cols list_discrete_values assocVarName
              assocVarElem {label_text ""} {label_placement "right"}
              {default_val ""} {dateFormat 0} }
```

Args:

frame_name	name of the frame where the spinbox widget will be placed.
cols	number of columns the spinbox entry occupies.
list_discrete_values	list of discrete values considered valid by the spinbox entry widget.
assocVarName	associated variable name for the spinbox entry widget.
assocVarElem	associated variable element for the spinbox entry widget.
label_text	the text string to label the spinbox.
label_placement	where to place 'label_text': right, left, top, bottom.
default_val	default value for the spin box entry.
dateFormat	boolean value (0 or 1) for declaring entry to be of date type: meaning numbers from 0 to 9 are to be given a 0 prefix.

Returns
a list: frame_name, spinbox entry widget, spinbox's scrollbar, and label.

Control Flow:
Build a full entry box. Give it a 'default_val' as value. If dateFormat is set, then prefix value with a 0 if between 0 and 9.
Make the spinbox entry visible.
Create a scrollbar and associate it with the entry widget.
Create a label for the entry.
Make scrollbar and label both visible, and arranged on the screen according to instruction of 'label_placement'.

Set 'list_discrete_values' to vlist.<spinbox_entry_name>.
Associate 'assocVarName(assocVarElem)' (or simply assocVarName) as textvariable to spinbox entry widget.
Register entry widget as a spinbox entry.

Return the names of the widgets involved.

12.12.2.

spincmd()

```
spincmd{ sbox dateFormat view_idx }
```

Args:

sbox	the name of the spinbox entry widget.
dateFormat	boolean variable (0 or 1) that says to prefix with a 0 if entry value is between 0 and 9.
view_idx	The index to entry widget to view.

Control Flow:

Call incr_spinbox if view_idx is -1; otherwise, call decr_spinbox.

12.12.3.

disable_spinbox()

```
disable_spinbox {spinEntry spinScroll {spinLabel ""}}
```

Args:

spinEntry	the name of the spinbox entry widget to disable.
spinScroll	the name of the spinbox scrollbar widget to disable.
spinLabel	the name of the spinbox label widget to disable.

Control Flow:

Disable the spinbox entry widget.
Disable the spinbox scrollbar.
Disable the spinbox label.

12.12.4.

enable_spinbox()

```
enable_spinbox {spinEntry spinScroll {spinLabel ""}}
```

Args:

spinEntry	the name of the spinbox entry widget to enable.
spinScroll	the name of the spinbox scrollbar widget to enable.
spinLabel	the name of the spinbox label widget to enable.

Control Flow:

Enable the spinbox entry widget.
Enable the spinbox scrollbar.

Enable the spinbox label.

12.12.5.

incr_spinbox()

```
incr_spinbox {spbox {dateFormat 0} }
```

Args:

spbox	name of the spinbox entry widget whose value will be incremented by 1.
dateFormat	boolean value (0 or 1) for declaring entry to be of date type: meaning numbers from 0 to 9 are to be given a 0 prefix.

Returns

new value of the spinbox entry widget.

Control Flow:

Get the list of values considered valid by the entry widget.
Get the current value of the entry widget.
If list of values is a range; then
 if entry value is currently Empty, then
  set the value to the minimum
 else
  set the value to the next number in the range. Don't increment anymore
  when the highest value is reached.
 endif
else
 if entry value is current Empty, then
  set the value to the first item in the list_discrete_values
 else
  set the value to next item in list_discrete_values. Don't do anything
  if we've run out of elements to access from list_discrete_values.
endif
Return new value of the entry widget.

12.12.6.

decr_spinbox()

```
decr_spinbox {spbox {dateFormat 0} }
```

Args:

spbox	name of the spinbox entry widget whose value will be decremented by 1.
dateFormat	boolean value (0 or 1) for declaring entry to be of date type: meaning numbers from 0 to 9 are to be given a 0 prefix.

Returns

new value of the spinbox entry widget.

Control Flow:

Get the list of values considered valid by the entry widget.

Get the current value of the entry widget.
If list of values is a range; then
 if entry value is currently Empty, then
  set the value to the maximum
 else
  set the value to the previous number in the range. Don't decrement anymore
  when the lowest value is reached.
 endif
else
 if entry value is current Empty, then
  set the value to the last item in the list_discrete_values
 else
  set the value to previous item in list_discrete_values. Don't do anything
  if we've run out of elements to access from list_discrete_values.
endif
Return new value of the entry widget.

12.12.7.

check_spinbox_value()

```
check_spinbox_value {spin_entry}
```

Args:

spin_entry	name of the spinbox entry widget whose value will be checked for validity.

Control Flow:

If spinbox's value list is a range of numbers, then
 blank out entry if spinbox value is not numeric, or not in the range.
else
 blank out entry of spinbox value does not match any of the values in
  value list.
endif

12.13. File: text.tk

This file contains routines supporting the text widget.

12.13.1.

buildFullTextbox()

```
buildFullTextbox{frame_name ColxRow scrollbarType {text_title ""}}
```

Args:

frame_name	name of the frame where the textbox will be placed.
ColxRow	number of columns and rows of the text widget.
scrollbarType	type of scrollbar to associate with the text widget - "yscroll" or "no-scroll".

text_title the text string for the label widget.

Returns
a list: frame_name, name of the text widget, and scrollbar.

Control Flow:
Create a user-friendly text widget with the specified dimensions.
Create a label for the text widget if text_title is set.
Make text and label widget visible.
Create and make visible a scrollbar. Associate the scrollbar with the text widget.

12.14. File: qsub.tk

This file contains routines supporting the Submit window.

12.14.1.

qsub()

```
qsub{}
```

Control Flow:
Create the submit dialog box.
Mark the Submit window active.
Populate the bottom part of the dialog box with command buttons:
confirm submit, interactive, cancel, reset options to default, and help.
Configure "reset options to default" button to set back to default values the widgets found in the Submit window only, and invoking as needed the StdoutRet, StderrRet, NotifyAbort, NotifyBegin, NotifyEnd, EnvList checkboxes, and the StdoutMerge, StderrMerge, and NoMerge radioboxes.
Configure "interactive" button to run cmdpath(QSUB) with a -I option in the background.
Configure "confirm submit" button to:
1) pre build the values of qsubv based on current widget values found in the Submit window and supporting windows.
2) if scriptName is set in the FILE entry box,
run "cmdpath(QSUB) <options> scriptName"
else
run "cmdpath(QSUB) <options> << {input taken from contents of the FILE text box}"
If command executed successfully; then get jobs Data.
Configure "cancel" button to destroy the qsub dialog box when clicked.
Configure "help" button to bring up the help page associated with the Submit window.

Populate the top and middle parts of the dialog box with the following widgets:
Create all the necessary frames to hold and group related widgets together.
Create the File text box.
Create the OPTIONS and OTHER OPTIONS labels.
Create the OTHER OPTIONS buttons: job dependency, file staging, and misc.
configure "job dependency" button to bring up the dependency dialog box.
configure "file staging" button to bring up the staging dialog box.

configure "misc" button to bring up the miscellaneous dialog box.
Create the SCRIPT label.
Create a user-friendly FILE entry box that also recognizes wildcards (*,~)
Create a user-friendly PBS FILE Prefix entry box.
Create the FILE "load" and "save" buttons.
Configure "load" button to:
(1) set busy cursor
(2) give an error if FILE entry is empty, or script FILE does not exist, or script FILE is not a regular file.
(3) run "PBS_SCRIPTLOAD_CMD -C <PBS FILE Prefix entry value> <script FILE entry value>"
(4) reset all widget values in the Submit window, dependency, staging, and misc dialog boxes.
(5) load new values to widgets using output obtained from (3). Invoke radiobuttons, checkbuttons in the Submit window only as needed.
(6) reset Prefix entry value to previous value before (5) was execute.
(7) Load the FILE text box with the execution lines (non-PBS lines) of the script FILE, which is currently saved in some buffer file.
(8) Remove the buffer file.
(9) Remove busy cursor.
Configure "save" button to:
(1) set busy cursor
(2) pre build the values of qsubv based on current widget values found in the Submit window and supporting windows.
(3) Open the file specified in the FILE entry widget. Use this file as the destination for PBS option lines (values taken from whatever widget is currently set), and execution lines as taken from the contents of the FILE text box. PBS directive is taken from the value of the Prefix entry box. The format is:
<PBS option lines>
<execution lines>
Also, bring up a confirmation box upon encountering a file that already exists.
(4) remove busy cursor.
(5) Bring up an Info box informing user of successful completion of (3) task.
Create a user-friendly Job Name entry box.
Create the priority spinbox.
Create the destination listbox. Make it single-selectable.
Load the listbox with values from the Queues listbox in the main xpbs window. Add a special default entry called "@hostsSelected" at the top of the listbox. This is assuming of course that hostsSelect contains only 1 entry. Select/highlight this default entry.
Build the "When to Queue" radiobuttons: NOW, and LATER at.
Configure the "LATER at" button to bring up the dateTime dialog box.
Configure the "NOW" button to reset the month, day, year, hh, min, and ss values to default when clicked.
Create a user-friendly "Account Name" entry box.
Create the "Hold Job" checkbutton.
Create the labels "Notify" and "when".
Create the "email addrs.." button.
Create the checkbuttons "job aborts", "job begins execution", and "job terminates".
- configure the checkbuttons in a way that the "email_addrs" button is

disabled when "job aborts", "job begins execution", and
"job terminates" buttons are currently deselected; otherwise, enable
"email_addrs" button.
Create the "Resources List" label.
Create the help button and associate with it help page on resources.
Create the resources=value box.
Create the "Merge to Stdout", "Merge to Stderr", and "Don't Merge"
radiobuttons.
Create the "Stdout in exec_host:" and "Stderr in exec_host:" checkbuttons.
Build the "Stdout File Name" entry box. Enable regular tabbing, and
overselect.
Build the "Stdout File Host Name" entry box. Enable regular tabbing, and
overselect.
Build the "Stderr File Name" entry box. Enable regular tabbing, and
overselect.
Build the "Stderr File Host Name" entry box. Enable regular tabbing, and
overselect.

Configure "Stdout in exec_host:" button to disable "Stdout File/Host Name"
widgets when invoked. Set focus to "Stderr File Name".
Configure "Stderr in exec_host:" button to disable "Stderr File/Host Name"
widgets when invoked. Set focus to "Stdout File Name".
Configure "Merge to Stdout" button to disable "Stderr File Name/Host Name"
widgets when invoked, and to enable "Stdout in exec_host:" button. Set
focus to "Stdout File Name".
Configure "Merge to Stderr" button to disable "Stdout File Name/Host Name"
widgets when invoked, and to enable "Stderr in exec_host:" button. Set
focus to "Stderr File Name".
Configure "Don't Merge" button to enable "Stdout in exec_host:" and
"Stderr in exec_host:" buttons, and if no button conflict, the
"Stdout File/Host Name" and/or "Stderr File/Host Name".

Create the "Environment Variables to Export" label.
Build environment variable=value box.
Build the checkbuttons "Current" and "Other Variables".
Configure the "Other Variables" button to enable the environment variable
entry boxes when the button is set, or disable the entry boxes when the
button is not set.

Make all widgets visible.
Set all widget values to default including those in the job dependency,
staging, and misc dialog boxes. Invoke appropriate button widgets as well
according to loaded values.
Wait for visibility of the Submit window before removing the busy cursor.
Wait until the Submit is destroyed.
Mark the Submit window inactive.
Free up the storage used up by the global variable qsubv.

12.14.2.

pre_build_qsub_opt()

```
pre_build_qsub_opt{}
```

Returns

0 if qsubv was successfully built; 1 otherwise.

Control Flow:

Set qsubv(destination) value based on what is currently selected in the destination listbox
Set qsubv(mail_option) based on values of the notify option checkbuttons. If none of the checkbuttons is selected, then set qsubv(mail_option) to "n".
Set qsubv(keep_args) based on what is set in "Stdout in exec_host:" and "Stderr in exec_host:" buttons.
Set qsubv(stdoutPath) to whatever non-default setting of the "Stdout File/Host Name" widgets.
Set qsubv(stderrPath) to whatever non-default setting of the "Stderr File/Host Name" widgets.
Set qsubv(res_args) to complete, non-default values of the resources=value entry boxes.
Set qsubv(env_args) to complete, non-default values of the environment=value entry boxes.
Call pre_build_depend_opt
Call pre_build_staging_opt
Call pre_build_misc_opt
Call pre_build_email_opt if qsubv(mail_option) is != n; otherwise, call init_qsub_email_argstr
Call pre_build_datetime_opt if qtime setting is not "NOW"; otherwise, call init_qsub_datetime_argst
return 0

12.14.3.

invoke_qsub_widgets()

```
invoke_qsub_widgets{}
```

Control Flow:

Invoke the checkbuttons: chkboxStdoutRet, chkboxStdoutRet, chkboxNotifyAbort, chkboxNotifyBegin, chkboxNotifyEnd, chkboxEnvList if set.
Invoke the radiobuttons: radioboxStdoutMerge, radioboxStderrMerge, radioboxNoMerge if set.

12.14.4.

confirmDelete()

```
confirmDelete{}
```

Control Flow:

Popup the "Save to File" dialog box.
Populate the top part of the dialog box with a message widget asking if user

wants an existing file to be overwritten.
Populate the bottom part of the dialog box with a Yes, No button.
Bind the <Return> key to invoke the No button.
Display the top and bottom parts of the dialog box.

12.14.5.

serverSelect()

```
serverSelect{}
```

Control Flow:
Popup the "Server Selection" dialog box.
Populate the top part of the dialog box with a message widget, and a listbox enumerating the list of hosts currently selected.
Populate the bottom part of the dialog box with an Ok button.
Bind the <Return> key to invoke the Ok button.
Display the top and bottom parts of the dialog box.

12.15. File: qalter.tk

This file contains routines supporting the Modify window.

12.15.1.

qalter()

```
qalter{}
```

Control Flow:
Create the Modify dialog box.
Mark the Modify window active.
Populate the bottom part of the dialog box with command buttons:
confirm modify, cancel, reset options to default, and help.
Configure "reset options to default" button to set back to default values the widgets found in the Modify window only, and invoking as needed the StdoutRet, StderrRet, NoRet, Notify, NotifyAbort, NotifyBegin, NotifyEnd checkboxes, and the StdoutMerge, StderrMerge, NoMerge radioboxes, and NoChange.
Configure "confirm modify" button to:
1) pre build the values of qalterv based on current widget values found in the Modify window and supporting windows:
Set qalterv(mail_option) based on values of the notify option checkbuttons. If none of the checkbuttons is selected, then set qalterv(mail_option) to "n".
Set qalterv(hold_args) based on values found on the hold job check buttons.
Set qalterv(keep_args) based on what is set in "Stdout in exec_host:" and "Stderr in exec_host:" buttons.
Set qalterv(stdoutPath) to whatever non-default setting of the "Stdout File/Host Name" widgets.

Set qalterv(stderrPath) to whatever non-default setting of the "Stderr File/Host Name" widgets.
Set qalterv(res_args) to complete, non-default values of the resources=value entry boxes.
2) if qalter options specified is Empty, then
issue a WARNING message.
else
run "cmdpath(QALTER) <options> <jobsSelected>
endif
If command executed successfully; then get jobs Data.
(3) destroy Modify window.
Configure "cancel" button to destroy the qalter dialog box when clicked.
Configure "help" button to bring up the help page associated with the Modify window.

Populate the top and middle parts of the dialog box with the following widgets:
Create all the necessary frames to hold and group related widgets together.
Create the OTHER ATTRIBUTES buttons: job dependency, file staging, and misc.
configure "job dependency" button to bring up the dependency dialog box.
configure "file staging" button to bring up the staging dialog box.
configure "misc" button to bring up the miscellaneous dialog box.
Create the Hold Types widgets: place user, place system, place other check buttons, and clear all checkbutton.
Create the "Modify job(s):" label.
Build the job id listbox making it non-selectable. The listbox contents are those currently selected/highlighted in the Jobs listbox of the main **xpbs** window.
Create the "ATTRIBUTES" label.
Create a user-friendly Job Name entry box.
Create the priority spinbox.
Build the "When to Queue" radiobuttons: NOW, and LATER at.
Configure the "LATER at" button to bring up the dateTime dialog box.
Configure the "NOW" button to reset the mon, day, year, hh, min, ss values to default when clicked.
Create a user-friendly "Account Name" entry box.
Create the labels "Notify" and "when".
Create the "email addrs.." button.
Create the checkbuttons "job aborts", "job begins execution", and "job terminates".
- configure the checkbuttons in a way that the "email_addrs" button is disabled when "job aborts", "job begins execution", and "job terminates" buttons are currently deselected; otherwise, enable "email_addrs" button.
Create the "Resources List" label.
Create the help button and associate with it help page on resources.
Create the resources=value box.
Create the "Merge to Stdout", "Merge to Stderr", and "Don't Merge", "No Change" radiobuttons.
Create the "Stdout in exec_host:" and "Stderr in exec_host:" checkbuttons.
Build the "Stdout File Name" entry box. Enable regular tabbing, and overselect.
Build the "Stdout File Host Name" entry box. Enable regular tabbing, and overselect.

Build the "Stderr File Name" entry box. Enable regular tabbing, and overselect.
Build the "Stderr File Host Name" entry box. Enable regular tabbing, and overselect.

Configure "Stdout in exec_host:" button to disable "Stdout File/Host Name" widgets when invoked. Set focus to "Stderr File Name".
Configure "Stderr in exec_host:" button to disable "Stderr File/Host Name" widgets when invoked. Set focus to "Stdout File Name".
Configure "Merge to Stdout" button to disable "Stderr File Name/Host Name" widgets when invoked, and to enable "Stdout in exec_host:" button. Set focus to "Stdout File Name".
Configure "Merge to Stderr" button to disable "Stdout File Name/Host Name" widgets when invoked, and to enable "Stderr in exec_host:" button. Set focus to "Stderr File Name".
Configure "Don't Merge" and "No Change" buttons to enable "Stdout in exec_host:" and "Stderr in exec_host:" buttons, and if no button conflict, the "Stdout File/Host Name" and/or "Stderr File/Host Name".

Register a default binding of "cancel" to the Modify window.
Make all widgets visible.
Set all widget values to default including those in the job dependency, staging, and misc dialog boxes. Invoke appropriate button widgets as well according to loaded values.
Wait for visibility of the Submit window before removing the busy cursor.
Wait until the Submit is destroyed.
Mark the Submit window inactive.
Free up the storage used up by the global variable qalterv.

12.15.2.

invoke_qalter_widgets()

```
invoke_qalter_widgets {assoc_array}
```

Args:

assoc_array array holding the values of the qalter widgets.

Control Flow:

Invoke any of radio buttons: radioboxPlace, radioboxClear, radiobox, radioboxStdoutMerge, radioboxStderrMerge, radioboxNoMerge, radioboxNoChange, rboxRetain, rboxNoRetain if set.
Invoke the chkboxNotify if set.

12.16. File: depend.tk

This file contains the routines supporting the Job Dependency dialog box.

12.16.1.

depend()

```
depend{callerDialogBox {qalter 0} }
```

Args:

callerDialogBox	the name of the dialog box that called this routine.
qalter	boolean value (0 or 1) that says to build options under the context of qalter instead of qsub.

Control Flow:

If Submit window called this dialog box,
  set the global input array to "qsubv"
  set the default array to "def_qsub"
else
  set the global input array to "qalterv"
  set default array to "def_qalter"
endif
Set busy cursor.
Create the dependency dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
  ok, reset options to default, and help.
Configure the ok button to call "pre_build_depend_opt" and destroy the
  depend Dialog box when clicked.
Configure the "reset options to default" button to set widget values to
  default values, and invoke appropriate widgets.
Configure the help button to bring up help page relating to job dependency.
Create the necessary frames to hold/group widgets.
Create the labels for "Concurrency Set", "Schedule THIS job after:",
  "THIS job must have:", and "THIS job is dependent on".
Create the "on other job(s)" spinbox.
Create the "synccount" spinbox.
Create the "sync with job(s)" entry box.
Create the radio buttons invoking the "synccount" spinbox and
  "sync with job(s)" entry as well as a "no concurrency" button.
  Configure the radiobutton invoking "synccount" spinbox so that it enables the
    spinbox and it disables the "sync with job(s)" entry box. Set focus on
    spinbox entry.
  Configure the radiobutton invoking "sync with job(s)" entry so that it
    enables the entry and it disables the "synccount" spinbox.
  Configure the "no concurrency" radio button so that it disables the
   "sync with job(s)" entry and "synccount" spinbox.
  Invoke whichever radiobutton was set.
Create the widgets for after, afterok,
  afternotok, afterany. Associate each one with a checkbutton. Configure each
  checkbutton so that it enables the appropriate box. Finally, toggle/invoke
  the set checkbutton.
Create the widgets for before, beforeok,
  beforenotok, beforeany. Associate each one with a checkbutton. Configure each
  checkbutton so that it enables the appropriate widget. Finally, toggle/invoke
  the set checkbutton.
Register "ok" as the default <Return> key action.
Make all the widgets visible.

Wait for the visibility of the depend dialog box before removing the busy cursor.
Wait for the dependency dialog box to disappear before marking the window inactive.
set input focus back to the calling dialog box.
set grab back to the calling dialog box.

12.16.2.

pre_build_depend_opt()

```
pre_build_depend_opt {array def_array}
```

Args:

array — array to add elements to corresponding to widget values suitable for processing by qsub and qalter.

def_array — array holding default values for the Dependency widgets.

Control Flow:

Initialize the depend argument string.
Set array(depend) value based on values found in the dependency widgets.

12.16.3.

invoke_depend_widgets()

```
invoke_depend_widgets {assoc_array}
```

Args:

assoc_array — array holding the values of the depend widgets.

Control Flow:

Invoke any of radio buttons: radioboxSyncNone, radioboxSynccount, and radioboxSyncwith if set.
Invoke the check buttons: chkboxAfter, chkboxAfterok, chkboxAfternotok, chkboxAfterany, chkboxBefore, chkboxBeforeok, chkboxBeforenotok, chkboxBeforeany if set.

12.17. File: staging.tk

This file contains routines supporting the File Staging dialog box.

12.17.1.

staging()

```
staging {callerDialogBox {qalter 0}}
```

Args:

callerDialogBox	name of the dialog box that called this routine.
qalter	boolean value (0 or 1) that says to build options under the context of qalter instead of qsub.

Control Flow:

If Submit window called this dialog box,
set the global input array to "qsubv"
else
set the global input array to "qalterv"
endif
set busy cursor
Create the File Staging dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
ok, reset options to default, and help.
Configure the ok button to call "pre_build_staging_opt" and destroy the staging Dialog box when clicked.
Configure the "reset options to default" button to set widget values to default values.
Configure the help button to bring up help page relating to file staging.
Create the necessary frames to hold/group widgets.
Create the Stagein box.
Create the Stageout box.
Make all widgets visible.
Register the "ok" action as default <Return>key binding.
Wait for the visibility of the staging dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
set input focus back to the calling dialog box.
set grab back to the calling dialog box.

12.17.2.

check_staging_input()

```
check_staging_input {host file}
```

Args:

host	in a stagein/stageout directive, this is the hostname part.
file	in a stagein/stageout directive, this is the filename part.

Control Flow:

Get the directory name for file, access the host, and check for the existence of the directory holding the file. If dir does not exist, return -1. Either use the RSH utility if file resides remotely, or the internal "file" utilities within TCL if file resides locally.

12.17.3.

pre_build_staging_opt()

```
pre_build_staging_opt {array def_idx}
```

Args:

array	array to add elements to which corresponds to widget values suitable for processing by qsub and qalter.
def_idx	default value index to the **default** array.

Control Flow:

Initialize argument strings associated with file staging.
Set array(stagein_filelist) value based on values found in the stagein widgets.
Set array(stageout_filelist) value based on values found in the stagout widgets.

12.18. File: misc.tk

This file contains routines that support the Miscellaneous dialog box.

12.18.1.

misc()

```
misc {callerDialogBox {qalter 0}}
```

Args:

callerDialogBox	name of the dialog box that called this routine.
qalter	boolean value (0 or 1) that says to build options under the context of qalter instead of qsub.

Control Flow:

If Submit window called this dialog box,
  set the global input array to "qsubv"
  set default array to "def_qsub"
else
  set the global input array to "qalterv"
  set default array to "def_qalter"
endif
set busy cursor
Create the File Staging dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
  ok, reset options to default, and help.
Configure the ok button to call "pre_build_misc_opt" and destroy the
  Miscellaneous Dialog box when clicked.
Configure the "reset options to default" button to set widget values to
  default values Invoke appropriate button widgets according to new values
  loaded.
Configure the help button to bring up help page relating to miscellaneous
  other PBS options.
Create the necessary frames to hold/group widgets.
Create the checkpoint interval minute spinbox.
Create the radiobutton invoking the checkpoint interval spinbox, as well as

buttons for "when host shuts down", "at host's default minimum time, and "do not checkpoint" attributes.
- Configure the button invoking checkpoint interval spinbox so that it enables the associated spinbox.
- Configure the "when host shuts down", "at host's default min time", and "do not checkpoint" buttons so that they disable the checkpoint interval spinbox.

Create the main checkbutton invoking widgets for specifying checkpoint attributes.
- Configure the checkbutton so that if it is set, then enable all the checkpoint widgets; otherwise, leave the widgets disabled.

Create the "rerunnable" and "not rerunnable" radiobuttons.
Create the checkbutton invoking the radiobuttons for specifying the rerunnable attribute of a job.
- Configure the checkbutton so that if it is set, then enable all the rerunnable widgets; otherwise, leave the widgets disabled.

Create the Shell path box.
Create the Group list box.
Create the User list box.
Make all widgets visible.
Register the "ok" action as default <Return>key binding.
Wait for the visibility of the staging dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Set input focus back to the calling dialog box.
Grab back the calling dialog box.

12.18.2.

pre_build_misc_opt()

```
pre_build_misc_opt {array def_array}
```

Args:

array	array to add elements to which corresponds to widget values suitable for processing by qsub and qalter.
def_array	array holding default values associated with misc widgets.

Control Flow:

Initialized the misc argument strings.
set array(checkpoint_arg) based on values found in checkpoint widgets.
set array(shell_args) based on values found in Shell path box.
set array(user_args) based on values found in User List box.
set array(group_args) based on values found in Group List box.

12.18.3.

invoke_misc_widgets()

```
invoke_misc_widgets {assoc_array}
```

Args:

assoc_array	array holding the values of the misc widgets.

Control Flow:

Invoke any of radio buttons: radioboxManual, radioboxAuto, radioboxDefault, and radioboxNoChkpnt if set.
Invoke the check buttons: chkboxChkpnt and chkboxMark if set.

12.19. File: email_list.tk

This file contains routines that support the Email Addresses dialog box.

12.19.1.

email_list()

```
email_list {callerDialogBox {qalter 0}}
```

Args:

callerDialogBox	name of the dialog box that called this routine.
qalter	boolean value (0 or 1) that says to build options under the context of qalter instead of qsub.

Control Flow:

If Submit window called this dialog box,
  set the global input array to "qsubv"
else
  set the global input array to "qalterv"
endif
set busy cursor
Create the Email Addresses dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
  ok, reset options to default, and help.
Configure the ok button to call "pre_build_email_opt" and destroy the
  Email Addresses Dialog box when clicked.
Configure the "reset options to default" button to set widget values to
  default values. Invoke appropriate button widgets according to new values
  loaded.
Configure the help button to bring up help page relating to email addresses.
Create the necessary frames to hold/group widgets.
Create the **numMisc** entry boxes for the Email addresses box.
Make all widgets visible.
Register the "ok" action as default <Return>key binding.
Wait for the visibility of the email addresses dialog box before removing busy
cursor.
 Wait for window to be destroyed before marking the dialog box as
inactive.
Set input focus back to the calling dialog box.
set a grab back to the calling dialog box.

12.19.2.

pre_build_email_opt()

```
pre_build_email_opt {array}
```

Args:

array	array to add elements to corresponding to widget values suitable for processing by qsub and qalter.

Control Flow:

set array(email_args) based on values found in Email Addresses box.

12.20. File: datetime.tk

Initialize email addresses argument string. This file contains routines that support the Date/Time dialog box.

12.20.1.

dateTime()

```
dateTime {callerDialogBox {qalter 0}}
```

Args:

callerDialogBox	name of the dialog box that called this routine.
qalter	boolean value (0 or 1) that says to build options under the context of qalter instead of qsub.

Control Flow:

If Submit window called this dialog box,
set the global input array to "qsubv"
else
set the global input array to "qalterv"
endif
If current values for mon, day, year, hour, minutes, and seconds is set to
default, then set date/time to current.
set busy cursor
Create the Date/Time dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to check validy of entered date/time value,
call "pre_build_datetime_opt", and destroy the
Date/Time Dialog box when clicked.
Configure the help button to bring up help page relating to date/time.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the date/time spinbox widgets.
Make all widgets visible.
Register the "ok" action as default <Return> key binding.

Wait for the visibility of the date/time dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Set input focus back to the calling dialog box.

12.20.2.

pre_build_datetime_opt()

```
pre_build_datetime_opt {array}
```

Args:

array — array to add elements to corresponding to widget values suitable for processing by qsub and qalter.

Control Flow:

initialize the date/time argument string.
set array(exec_time) based on values found in date/time widgets.

12.21. File: qterm.tk

This file contains routines that support the Terminate Server dialog box.

12.21.1.

qterm()

```
qterm{}
```

Control Flow:

set busy cursor
Create the Terminate Server dialog box.
Mark the window as active.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the "Terminate server(s):" label.
Create the servername(s) listbox and make it read-only. Insert the values found highlighted/selected in the Hosts listbox of the main **xpbs** window.
Create the type of shutdown radiobuttons - "immediate" and "delay".
Set the default shutdown type.
Make all widgets visible.
Populate the bottom part of the dialog box with the following buttons: terminate, cancel, and help.
Configure the terminate button to run:
"cmdpath(QTERM) -t <shutdown_type> <hostsSelected>"
If command execution was a success, then get new data for servers, queues, and jobs, and destroy the Terminate Server dialog box.
Configure the "cancel" button so that it destroys the Terminate Server dialog box when clicked.
Configure the help button to bring up help page relating to qterm.
Register the "cancel" action as default <Return> key binding.
Wait for the visibility of the Terminate Server dialog box before removing busy

cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.22. File: qdel.tk

This file contains routines that support the Delete Job dialog box.

12.22.1.

qdel()

`qdel{}`

Control Flow:
set busy cursor
Create the Delete dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
delete, cancel, and help.
Configure the delete button to run:
"cmdpath(QTERM) -W <delay_secs> <jobsSelected>"
If command execution was a success, then get new data for jobs only, and
destroy the Delete dialog box.
Configure the "cancel" button so that it destroys the Delete dialog
box when clicked.
Configure the help button to bring up help page relating to qdel.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the "Delete job(s)" label.
Create the jobid(s) listbox and make it read-only. Insert the values found
highlighted/selected in the Jobs listbox of the main **xpbs** window.
Create the "For running job(s), send kill signal after" label.
Create the delay_signal spinbox.
Set the default delay_signal.
Register the "cancel" action as default <Return> key binding.
Make all widgets visible.
Wait for the visibility of the Delete Job dialog box before removing busy
cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Free up the storage occupied by qdelv.

12.23. File: qhold.tk

This file contains routines that support the Hold Job dialog box.

12.23.1.

qhold()

`qhold{}`

Control Flow:
set busy cursor
Create the Hold Job dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
hold, cancel, and help.
Configure the hold button to run:
"cmdpath(QHOLD) -h <not_default_hold_types_string> <jobsSelected>"
If command execution was a success, then get new data for jobs only, and
destroy the Hold Job dialog box.
Configure the "cancel" button so that it destroys the Hold Job dialog
box when clicked.
Configure the help button to bring up help page relating to qhold.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the jobid(s) listbox and make it read-only. Insert the values found
highlighted/selected in the Jobs listbox of the main **xpbs** window.
Create the hold types checkbuttons - user, other, and system.
Set the default checkbuttons' values.
Make all widgets visible.
Register the "cancel" action as default <Return> key binding.
Wait for the visibility of the HOld Job dialog box before removing busy
cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Free up the storage occupied by qholdv.

12.24. File: qrls.tk

This file contains routines that support the Release Job dialog box.

12.24.1.

qrls()

```
qrls{}
```

Control Flow:
set busy cursor
Create the Release Job dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
release, cancel, and help.
Configure the release button to run:
"cmdpath(QRLS) -h <not_default_hold_types_string> <jobsSelected>"
If command execution was a success, then get new data for jobs only, and
destroy the Release Job dialog box.
Configure the "cancel" button so that it destroys the Release Job dialog
box when clicked.
Configure the help button to bring up help page relating to qrls.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the jobid(s) listbox and make it read-only. Insert the values found
highlighted/selected in the Jobs listbox of the main **xpbs** window.

Create the hold types checkbuttons - user, other, and system.
Set the default checkbuttons' values.
Make all widgets visible.
Register the "cancel" action as default <Return> key binding.
Wait for the visibility of the Release Job dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Free up the storage occupied by qrlsv.

12.25. File: qsig.tk

This file contains routines that support the Signal Job dialog box.

12.25.1.

qsig()

```
qsig{}
```

Control Flow:
set busy cursor
Create the Signal Job dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons: signal, cancel, and help.
Configure the signal button to run:
"cmdpath(QSIG) -s <not_default_signal_name> <jobsSelected>"
If command execution was a success, then get new data for jobs only, and destroy the Signal Job dialog box.
Configure the "cancel" button so that it destroys the Signal Job dialog box when clicked.
Configure the help button to bring up help page relating to qsig.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the "to RUNNING job(s):" label.
Create the jobid(s) listbox and make it read-only. Insert the values found highlighted/selected in the Jobs listbox of the main **xpbs** window.
Create the "other signal" entry widget.
Create the radio buttons HUP, INT, KILL, TERM, and OTHER with the last one invoking the "other signal" entry widget.
- Configure all the radiobuttons (except OTHER) to disable the "other entry" widget when the buttons are invoked.
Set the default radiobuttons' value. Based on the default values, invoke the appropriate radiobutton.
Make all widgets visible.
Register the "cancel" action as default <Return> key binding.
Wait for the visibility of the Signal Job dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Free up the storage occupied by qsigv.

12.26. File: qmsg.tk

This file contains routines that support the Send Message to Running Job dialog box.

12.26.1.

qmsg()

```
qmsg{}
```

Control Flow:

set busy cursor
Create the Send Message dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
Send Message, cancel, and help.
Configure the Send Message button to run:
"cmdpath(QMSG) <Eflag> <Oflag> <message> <jobsSelected>"
If command execution was a success, then get new data for jobs only, and
destroy the Send Message dialog box.
Configure the "cancel" button so that it destroys the Send Message dialog box when clicked.
Configure the help button to bring up help page relating to qmsg.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the jobid(s) listbox and make it read-only. Insert the values found highlighted/selected in the Jobs listbox of the main **xpbs** window.
Create the message entry box.
Create the checkbuttons Stdout, and Stderr.
Set the default values for the checkbuttons.
Create the "to" and "of RUNNING job(s)" labels.
Make all widgets visible.
Register the "cancel" action as default <Return> key binding.
Wait for the visibility of the Send Message dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Free up the storage occupied by qmsgv.

12.27. File: qmove.tk

This file contains routines that support the Move Job dialog box.

12.27.1.

qmove()

```
qmove{}
```

Control Flow:

set busy cursor

Create the Move Job dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
move, cancel, and help.
Configure the move button to run:
"cmdpath(QMOVE) <queue_selected> <jobsSelected>"
If command execution was a success, then get new data for jobs only, and
destroy the Move Job dialog box.
Configure the "cancel" button so that it destroys the Move Job dialog
box when clicked.
Configure the help button to bring up help page relating to qmove.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the jobid(s) listbox and make it read-only. Insert the values found
highlighted/selected in the Jobs listbox of the main **xpbs** window.
Create the destination listbox. and make it single-selectable. Load it with
the current values in the in the queuesListbox of the main
xpbs window. Select/highlight the firs entry.
Create the "to queue (select one):" and "Move job(s):" labels.
Make all widgets visible.
Register the "cancel" action as default <Return> key binding.
Wait for the visibility of the Move Job dialog box before removing busy
cursor.
Wait for window to be destroyed before marking the dialog box as inactive.
Free up the storage occupied by qmove.

12.28. File: owners.tk

This file contains routines that support the Select Owners dialog box.

12.28.1.

owners()

`owners{}`

Control Flow:
set busy cursor
Create the Select Owners dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select owners_list argument after
checking for completeness of user@hostname arguments. Also, destroy the
Select Owners dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to owners selection.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the user@hostname box.
Create the radiobuttons for invoking the user@hostname box, and for
invoking the wildcard "ANY" owners input. Configure the former button to
enable the box when invoked, and configure the latter to disable the
box when invoked.

Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Owners dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.29. File: state.tk

This file contains routines that support the Select Job States dialog box.

12.29.1.

state()

```
state{}
```

Control Flow:
set busy cursor
Create the Select Job States dialog box.
Mark the window as active.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the checkbuttons corresponding to state selection: R, Q, W, H, E, T.
Make all widgets visible.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select states argument after checking for presence of states value. Also, destroy the Select Job States dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to states selection.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select States dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.30. File: jobname.tk

This file contains routines that support the Select Job Name dialog box.

12.30.1.

jobname()

```
jobname{}
```

Control Flow:
set busy cursor
Create the Select Job Name dialog box.
Mark the window as active.

Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select jobname argument after checking for presence of jobname argument. Also, destroy the Select Job Name dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to job name selection.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create job name entry box.
Create the radiobuttons for invoking the job name entry box, and for invoking the wildcard "ANY" job name input. Configure the former button to enable the entry when invoked, and configure the latter to disable the entry when invoked.
Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Job Name dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.31. File: hold.tk

This file contains routines that support the Select Hold Types dialog box.

12.31.1.

hold()

```
hold{}
```

Control Flow:
set busy cursor
Create the Select Hold Types dialog box.
Mark the window as active.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the checkbuttons corresponding to hold types: user, other, system.
Create the radiobuttons for invoking the checkbuttons, and for invoking the wildcard "ANY" hold types input. Configure the former button to enable the checkbuttons when invoked, and configure the latter to disable the checkbuttons when invoked.
Set default values for the various widgets.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select hold types argument. If no hold types checkbutton is selected, then set the argument to "n". Destroy the Select Hold Types dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to hold types selection.
Register the "ok" action as default <Return> key binding.
Make all widgets visible.
Wait for the visibility of the Select Hold Types dialog box before removing busy cursor.

Wait for window to be destroyed before marking the dialog box as inactive.

12.32. File: acctname.tk

This file contains routines that support the Select Account Name dialog box.

12.32.1.

acctname()

```
acctname{}
```

Control Flow:

set busy cursor
Create the Select Account Name dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select jobname argument after
checking for presence of account name argument. Also, destroy the
Select Job Name dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to account name
selection.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create account name entry box.
Create the radiobuttons for invoking the account name entry box, and for
invoking the wildcard "ANY" account name input. Configure the former button
to enable the entry when invoked, and configure the latter to disable the
entry when invoked.
Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Account Name dialog box before removing
busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.33. File: checkpoint.tk

This file contains routines that support the Select Checkpoint Attribute dialog box.

12.33.1.

checkpoint()

```
checkpoint{}
```

Control Flow:

set busy cursor
Create the Select Checkpoint Attribute dialog box.
Mark the window as active.

Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select checkpoint argument after checking for presence of an operator argument. Also, destroy the Select Checkpoint Attribute dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to checkpoint attribute selection.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create operator spin box.
Create the checkpoint interval spin box.
Create the radiobuttons for invoking the checkpoint interval spinbox, and for invoking "n", "s", "c", "u", and the wildcard "ANY" checkpoint attribute.
Configure the former button to enable the checkpoint interval spinbox when invoked, and to consider as valid operators: =, !=, >=, >, <=, <.
Configure the "n", "s", "c" buttons to disable the checkpoint interval spinbox, and to consider as valid operators: =, !=, >=, >, <=, <.
Configure the "u" button to disable the checkpoint interval spinbox, and to consider as valid operators: !=, =. Load the operator spinbox with the default "=" value if current value is either "!=" or "=".
Configure the "ANY" button to disable the checkpoint interval spinbox, and to consider as valid operators: =. Load the operator spinbox with the default "=" value.
Set default values for the various widgets.
Make all widgets visible.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Checkpoint attribute dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.34. File: qtime.tk

This file contains routines that support the Select Queue Time dialog box.

12.34.1.

qtime()

```
qtime{}
```

Control Flow:
set busy cursor
Create the Select Queue Time dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select date/time argument after checking for completeness and validity of the date/time argument. Also, destroy the Select Queue Time dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to queue time selection.

Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create operator spin box. Give it values: =, !=, >=, >, <=, <.
Create the date/time spin boxes.
Create the radiobuttons for invoking the date/time spinboxes, and for invoking the wildcard "ANY" date/time spinbox.
Configure the former button to enable the date/time spinboxes when invoked, and to consider as valid operators: =, !=, >=, >, <=, <.
Configure the "ANY" button to disable the date/time spinboxes, and to consider as valid operators: =. Load the operator spinbox with the default "=" value.
Create the label "Queue Time".
Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Execution Time attribute dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.35. File: res.tk

This file contains routines that support the Select Resource Attributes dialog box.

12.35.1.


```
res{}
```

Control Flow:
set busy cursor
Create the Select Resource Attributes dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons: ok, and help.
Configure the ok button to create the select resource attributes argument after checking for completeness of a resources and operator argument. Also, destroy the Select Resources dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to resource attributes selection.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the res<op>val box.
Create the radiobuttons for invoking the box, and for invoking the wildcard "ANY" resource attributes input.
Configure the former button to enable the box when invoked, and configure the "ANY" button to disable the box.
Create the label "Resource Attribute".
Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Resources attribute dialog box before

removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.36. File: priority.tk

This file contains routines that support the Select Priority Criteria dialog box.

12.36.1.

priority()

```
priority{}
```

Control Flow:
set busy cursor
Create the Select Priority dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select priority argument after
checking for presence of priority entry value eand operator argument. Also,
destroy the Select Priority dialog box after argument has been successfully
created.
Configure the help button to bring up help page relating to priority attribute
selection.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the operator spin box.
Create the priority spinbox. Load as valid values: -1024 to 1023.
Create the radiobuttons for invoking the priority spinbox, and for
invoking the wildcard "ANY" priority input.
Configure the former button to enable the priority spinbox when
invoked and to consider as valid operator spinbox values: =, !=, >=, >,
<=, <, and configure the "ANY" button to disable the priority spinbox and
to consider as valid operator spinbox values: =.
Create the label "Priority".
Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Priority attribute dialog box before
removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.37. File: rerun.tk

This file contains routines that support the Select Rerun Attribute dialog box.

12.37.1.

rerun()

```
rerun{}
```

Control Flow:
set busy cursor
Create the Select Rerun attribute dialog box.
Mark the window as active.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the radiobuttons: y, n, ANY.
Populate the bottom part of the dialog box with the following buttons:
ok, and help.
Configure the ok button to create the select rerun attribute argument. Destroy the Select Rerun dialog box after argument has been successfully created.
Configure the help button to bring up help page relating to rerun attribute selection.
Create the label "Rerunnable".
Make all widgets visible.
Set default values for the various widgets.
Register the "ok" action as default <Return> key binding.
Wait for the visibility of the Select Rerunnable attribute dialog box before removing busy cursor.
Wait for window to be destroyed before marking the dialog box as inactive.

12.38. File: trackjob.tk

This file contains routines that support the Track Job feature.

12.38.1.

trackjob()

```
trackjob{}
```

Control Flow:
set busy cursor
Create the Track Job dialog box.
Mark the window as active.
Populate the bottom part of the dialog box with the following buttons:
start/reset tracking, stop, close window, and help.
Configure the start/reset tracking button to:
call trackjob_rstart TCL procedure
Configure the stop tracking button to:
- simply increment the TRACKJOB_UPDATE_SEQ causing any previous tracking to quit on its next polling activity.
- change the background color of the trackjob button back to normal color.
- inform user via an Info box that the job tracking was stopped.
Configure the close window button to simply destroy the Track Job dialog box when the button is clicked.
Configure the help button to bring up help page relating to track job feature.
Populate the top and middle parts of the dialog box with the following widgets:
Create the necessary frames to hold/group widgets.
Create the message widget containing the label "Periodically check for completion of jobs for user(s):".
Create the username box.
Create the trackjob interval spinbox. Load 1-9999 as valid values.

Create the radiobuttons for invoking the RSH entry widget, as well as for
telling xpbs that the output files are "local".
Set the default value for the radio buttons.
Create the "Jobs Found Completed" listbox. Make it single-selectable, and
register it as a special trackjob listbox.
If trackjob_output contains entries, then
load those entries into the listbox
endif
Make widgets visible.
Wait for visibility of the Track Job dialog box before removing busy cursor.
Register the "close window" action as default <Return> key binding.
Wait for window to be destroyed before marking the dialog box as inactive.
compress the username box array.

12.38.2.

trackjob_auto_update()

```
trackjob_auto_update{update_seq}
```

Args:

update_seq some tracking sequence number

Control Flow:

if <tracking sequence number> does not match <active sequence number> ; then
quit out of this routine
endif
call trackjob_auto_update_seq again after **trackjob_mins** of time.
Check for returned output files (call trackjob_check).

12.38.3.

trackjob_rstart()

```
trackjob_rstart{}
```

Control Flow:

Check to make sure we have a valid **trackjob_mins** value
run "PBS_QSTATDUMP_CMD -T -u <user_name(s)>" and dump output to trackjob_array.
NOTE: trackjob_array will contain one entry for each jobid found, and the
entries list the job owner, output file and error file names.

if <tracking sequence number> does not match <active sequence number> ; then
quit out of this routine
endif
call trackjob_auto_update_seq again after **trackjob_mins** of time.
Check for returned output files (call trackjob_check).
Delete all entries from the trackjob listbox.
Change the color of the trackjob button to normal color.
Free up the storage used by trackjob_output.

12.38.4.

trackjob_check()

```
trackjob_check{}
```

Control Flow:

```
if trackjob_array does not exit; then
  skip the rest of this routine
endif
if the Track Job dialog box exists, then
  delete all entries from the trackjob listbox
endif
Free up storage used by trackjob_output.
Send message to InfoBox that says we're looking for returned output files.
Foreach job in trackjob_array
do
  get the output file/path name of the job
  get the error file/path name of the job

  if files are local, then
    check for the existence of at least ONE of them.
    if the trackjob dialog box exists, then
      add the jobid to the trackjob listbox
    endif
    Also dd the jobid to the trackjob_output buffer array
  else
    check for existence of at least ONE of the files by running:
      "trackjob_rsh_command <outputfile/errorfile> test -f <file> && echo 1"
    if the trackjob dialog box exists, then
      add the jobid to the trackjob listbox
    endif
    Also add the jobid to the trackjob_output buffer array
  endif
done
Set the color of the trackjob button to signalColor if the trackjob_output
  buffer array contains at least one entry.
Send "done" message to InfoBox.
```

12.38.5.

trackjob_show()

```
trackjob_show{jobid}
```

Args:

jobid The jobid whose output file and error file will be shown.

Control Flow:

```
set busy cursor
```

Create the Job Output Dialog Box.
Mark the window active.
Get the output path/file name for jobid.
Get the error path/file name for jobid.
Create the necessary frames to hold/group widgets.
Create the output path label message panel.
Create the error path label message panel.
Create the output contents text box. Make it read-only. Associate a scrollbar.
Create the error contents text box. Make it read-only. Associate a scrollbar.
At the bottom of the dialog box, create an "ok" button. Configure this button so that when clicked, it destroys the show Dialog box.
Load the text boxes with values:
for local files, insert contents of output file into output textbox, and error file into error textbox using open/read.
for remote files, insert contents of output file into output textbox, and error file into error textbox using "trackjob_rsh_command <host> cat <file>"
Register "ok" as the default action for <Return> key.
Make all widgets visible.
Wait for the visibility of the Job Output dialog box before removing busy cursor.
Wait for the dialog box to be destroyed before marking it inactive.
Send "done" message to InfoBox.

12.39. File: auto_upd.tk

This file contains routines supporting the data auto updating feature in xpbs.

12.39.1.

auto_upd()

```
auto_upd{}
```

Control Flow:
set busy cursor
Create the Auto Update Dialog Box.
Mark the window active.
At the bottom of the dialog box, create the buttons: start updating, stop updating, close window, and help.
Configure "start updating" button so that it gets a new DATA_UPDATE_SEQ (a new active sequence), execute in **auto_update_mins** the TCL procedure data_auto_update, and inform user via a Help box that auto updating will start in **auto_update_mins**.
Configure the "stop updating" button so that it gets a new DATA_UPDATE_SEQ without running a data update.
Configure the "close window" button to simply destroy the Auto Update Dialog box when button is clicked.
Configure the "help" button to bring up a help page on auto updating of data.
Create the necessary frames to hold/group widgets.
Create the "mins" label.
Create the auto_update_mins spinbox. Give it values between 1-9999.
Register "cancel" as the default action for <Return> key presses.
Wait for the visiblity of the Auto Update dialog box before removing busy

cursor.
Wait for the dialog box to be destroyed before marking it inactive.

12.40. File: pref.tk

This file contains routines supporting the Preferences dialog box.

12.40.1.

pref()

```
pref{}
```

Control Flow:
set busy cursor
Create the Preferences Dialog Box.
Mark the window active.
Create the necessary frames to hold/group widgets.
Create the server hosts entry box.
At the bottom of the dialog box, create the buttons: ok, help.
Configure "ok" button so that it destroys the Preferences dialog box.
onfigure the "help" button to bring up a help page on user preferences.
Register "ok" as the default action for <Return> key presses.
Wait for the visiblity of the Preferences dialog box before removing busy cursor.
Wait for the dialog box to be destroyed before marking it inactive.

12.41. File: prefsave.tk

This file contains routines supporting the Preferences Save confirmation dialog box.

12.41.1.

prefsave()

```
prefsave{}
```

Control Flow:
Create the Save Preference Settings Dialog Box.
Create the message frame.
At the bottom of the dialog box, create the buttons: yes, no.
Configure "yes" button so that it calls the PrefSave function and then destroys the Preferences Save dialog box.
Configure "no" button to simply destroy the PrefSave dialog box.
Register "yes" button as the default action for <Return> key presses.

12.42. File: preferences.tcl

This file contains routines supporting the setting of user preferences.

12.42.1.

Pref_Init()

```
Pref_Init{ userDefaults appDefaults }
```

Args:

userDefaults name of the user preferences file.

appDefaults name of the application preferences file.

Control Flow:

PrefReadFile the application preferences file.
If the user preferences file exists, then PrefReadFile that too.

12.42.2.

PrefReadFile()

```
PrefReadFile{basename level}
```

Args:

basename name of a preferences file

level priority level of the resources defined in the file.

Control Flow:

Read/load the resources in the <basename> preferences file.
If color model is color, then read/load the resources in the <basename>-color file (if it exists).
If color model is monochrome, then read/load the resources in the <basename>-mono file (if it exists).

12.42.3.

PrefVar()

```
PrefVar{ item }
```

Args:

item name of a preferences item.

Control Flow:

Return the variable to associate with an X resource.

12.42.4.

PrefXRes()

```
PrefXRes{ item }
```

Args:

item name of a preferences item.

Control Flow:

Return the X resource name associated with a preference item.

12.42.5.

PrefDefault()

```
PrefDefault{ item }
```

Args:

item name of a preferences item.

Control Flow:

Return the default value associated with a preference item.

12.42.6.

PrefComment()

```
PrefComment{ item }
```

Args:

item name of a preferences item.

Control Flow:

Return the comment field of a preference item.

12.42.7.

PrefHelp()

```
PrefHelp{ item }
```

Args:

item name of a preferences item.

Control Flow:

Return the Help field of a preference item.

12.42.8.

Pref_Add()

```
Pref_Add { prefs }
```

Args:

prefs a list of preference items

Control Flow:

Foreach item in prefs;
do
if Xresources item's value is Empty; then
give the resource a default value
set the variable associated with the item to have the resource's value
else
set the variable associated with the item to have the resource's value
endif
done

12.42.9.

PrefValue()

```
PrefValue{ varName xres }
```

Args:

varName name of a variable associated with the X resource given by 'xres'.

xres name of an X resources to get value from.

Control Flow:

If varName already exists, simply return its value.
else get the value of the X resource associated with the varName. Assign this value to varName.
endif

12.42.10.

PrefValueSet()

```
PrefValueSet{ varName value }
```

Args:

varName name of X resource variable.

value value of some X resource.

Control Flow:

set varName to have the 'value'.

12.42.11.

PrefSave()

```
PrefSave{}
```

Control Flow:

Get the X resource values from user preferences file.
Create a <user preferences filename>.new and:
- put old resources value into this file.
- add a line "!!! Lines below here are automatically added" to the file.
- puts new resources values after the above line.

Now do a "mv <user preferences filename>.new <user preferences filename>".

12.42.12.

prefDoIt()

```
prefDoIt{}
```

Control Flow:

Get the initial values for xpbs resources.
Get the current values for the xpbs resources.
If corresponding values do not match, then return 1; otherwise, return 0.

12.43. Program: xpbs_datadump

The **xpbs_datadump** command is mainly used by **xpbs** to obtain information about servers, queues, and jobs within a single execution. By turning on certain command line options, this program can be instructed to (1) list only those jobs which meet a list of selection criteria, (2) list only job information (-J), (3) list only output file/path names of jobs for tracking purposes (-T). This program combines the features of qstat and qselect.

12.43.1. Overview

Parse the options on the execute line and build up an attribute list. Depending on what options are set, give the status of the server, queues, and jobs for each of the hosts listed in the execute line.

12.43.2. External Interfaces

Upon successful processing of all the operands presented to the xpbs_datadump command, the exit status will be a value of zero.

If the xpbs_datadump command fails to process any operand, the command exits with a value greater than zero.

12.43.3. File: xpbs_datadump.c

This file contains the main routine and some other functions related to server, queue, and jobs status as well as job selection. All other functions are in the PBS commands library.

12.43.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

argc The number of arguments on the command line.

argv The **argv** array contain the following arguments:
[−a [op]date_time] [−A account_string][−c [op]interval]
[−h hold_list] [−l resource_list] [−N name] [−p [op]priority]
[−q destination] [−r y | n] [−s states] [−u user_name] [−J]
[−T] [−t wait_timeout_secs] server_name..

envp The **envp** array contains environment variables for this process. None are used by the main routine, but some may be used by the library.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

```
Use getopt to get each option
      Build the attribute list for the select job request
If destination is given Then
      Determine the queue and server name
Foreach server listed in the execute line
do
 Connect to the server. Send alarm signal after timeout

 if "-J" option AND "-T" options are not set; then
  Send the status server request
  Print the server status returned
 endif

 if "-J" option AND "-T" options are not set; then
  Send the status queue request
  Print the queue status returned
 endif

 Send the select job status request
 if "-T" option is not set; then
  Print the job status for identifiers returned
 else
  Print information about jobs output file/host names, error file/host names
 endif
 Disconnect from the server
done
```

12.43.3.2.

set_attrop()

```
void set_attrop(struct attropl **list, char *name, char *resource,
                char *value, enum batch_op op)
```

Args:

list	The attribute list.
name	The name part of the attribute.
resource	The resource part of the attribute.
value	The value part of the attribute.
op	The operation part of the attribute.

Control Flow:

Allocate the memory for an attribute structure
If name is defined Then
    Allocate the memory for the name part
    Copy the name part
If resource is defined Then
    Allocate the memory for the resource part
    Copy the resource part
If value is defined Then
    Allocate the memory for the value part
    Copy the value part
Set the operation part
Add the attribute structure to the beginning of the attribute list

12.43.3.3.

check_op()

```
void check_op(char *opstring, enum batch_op *op, char *value)
```

Args:

opstring	The operator and value string from the command line.
op	The operator part of the string turned into an enum batch_op. The operator defaults to EQ if none is given.
value	The value part of the string.

Control Flow:

Set the operator to EQ
If opstring contains an operator Then
    Find out which operator was used
Copy the value part

12.43.3.4.

check_res_op()

```
int check_res_op(char *resources, char *name, enum batch_op *op, char *value, char **posit
```

Args:

resourcesThe comma delimited list of resources. The list looks like

name op value, ...

name The resource name.

op The operator.

value The value.

position The next position in the resource list to parse.

Returns:

Zero, if the resource list is parsed correctly, one otherwise.

Control Flow:

Scan for the resource name
Find out which operator was used
Scan for the resource value
Set the next character position

12.43.3.5.

istrue()

```
int istrue(char *string)
```

Args:

string Is this string some textual form of TRUE?

Returns:

True, if the strings represents true, false otherwise.

Control Flow:

Does the string match TRUE
Does the string match True
Does the string match true
Does the string match 1

12.43.3.6.

states()

```
void states(char *string, char *q, char *r, char *h, char *w, char *t, char *e
, int len)
```

Args:

string The string that holds the count of jobs in each state from the server.

q The number of queued jobs.

r The number of running jobs.

h The number of held jobs.

w	The number of waiting jobs.
t	The number of jobs in transit.
e	The number of exiting jobs.

Control Flow:

While the string is not empty Do
Scan for the next word
If it is Queued Then set the output pointer to q
If it is Running Then set the output pointer to r
If it is Held Then set the output pointer to h
If it is Waiting Then set the output pointer to w
If it is Transit Then set the output pointer to t
If it is Exiting Then set the output pointer to e
Copy the next word to where the output pointer is pointing

12.43.3.7.

display_statjob()

```
void display_statjob(struct batch_status *status, int prtheader, int full,
                     char *server_name)
```

Args:

status	A list of information about each job returned by the server.
prtheader	True, if the header is to be printed, false otherwise.
full	True, if a full display is requested, false for a normal display.
server_name	full name of the server associated with the jobs to be displayed.

Control Flow:

If not full and header Then
Print the header
While there is an item in the status list Do
If full Then
Print a full job display of all the attributes
Else
Print a normal display of the attributes listed in the ERS
Get the next item in the list

12.43.3.8.

display_statque()

```
void display_statque(struct batch_status *status, int prtheader, int full,
                     char *server_name)
```

Args:

status	A list of information about each queue returned by the server.
prtheader	True, if the header is to be printed, false otherwise.

full True, if a full display is requested, false for a normal display.

server_name full server name associated with the list of queues to be displayed.

Control Flow:

If not full and header Then
Print the header
While there is an item in the status list Do
If full Then
Print a full queue display of all the attributes
Else
Print a normal display of the attributes listed in the ERS
Get the next item in the list

12.43.3.9.

display_statserver()

```
void display_statserver(struct batch_status *status, int prtheader, int full)
```

Args:

status A list of information about the server returned by the server.

prtheader
True, if the header is to be printed, false otherwise.

full True, if a full display is requested, false for a normal display.

Control Flow:

If not full and header Then
Print the header
While there is an item in the status list Do
If full Then
Print a full server display of all the attributes
Else
Print a normal display of the attributes listed in the ERS
Get the next item in the list

12.43.3.10.

display_trackstatjob()

```
void display_trackstatjob(struct batch_status *status, char *server_name)
```

Args:

status A list of information about the jobs returned by the server.

server_name full server name associated with a job.

Control Flow:

While there is an item in the status list Do
print the full jobid, owner, output file/host name, error file/host name
Get the next item in the list
Done

12.44. Program: xpbs_scriptload

Command that reads in a PBS job script and returns a list of "job attribute = value" lines based on PBS directive lines found in the script. A special line called "Buffer File = <filename>" is also displayed where <filename> will contain non-PBS directive lines found in the script.

12.44.1. Overview

Parse the execute line and get the PBS prefix string to look for when scanning a script for PBS options. Get the script and check for embedded operands. Return "job attribute = value" lines based on operands found in the script. Also, include the line "Buffer_File = <filename>" which refers to the file that holds non-PBS directive lines found in the script.

12.44.2. External Interfaces

Upon successful processing of all the operands presented to the the xpbs_scriptload command, the exit status will be a value of zero.

If the xpbs_scriptload command fails to process any operand, the command exits with a value greater than zero.

12.44.3. File: xpbs_scriptload.c

This file contains the main routine and some other functions related to job script parsing only. All other functions are in the library.

12.44.3.1.

main()

```
main(int argc, char **argv, char **envp)
```

Args:

- argc — The number of arguments on the command line.
- argv — The **argv** array contain the following arguments:
  [-C directive_prefix] script
- envp — The **envp** array contains environment variables for this process. Not used in this program.

Returns:

Zero, if no errors are detected. Positive, otherwise.

Control Flow:

Use getopt to get each option
Get the script and any options embedded in the script
Print "job attribute = value" lines.
Print "Buffer_File = <filename>" line.

12.44.3.2.

set_dir_prefix()

```
char *set_dir_prefix(char *prefix)
```

Args:

prefix The directive prefix supplied by the user, if given.

Returns:

The directive prefix.

Control Flow:

If prefix has something in it Then
Use prefix
Else If the environment variable PBS_DPREFIX is defined Then
Use PBS_DPREFIX
Else
Use the default PBS_DPREFIX_DEFAULT

12.44.3.3.

isexecutable()

```
int isexecutable(char *line)
```

Args:

line A line of the script file.

Returns:

True, if the line is not a comment, false otherwise.

Control Flow:

Is the first non-blank character a #?

12.44.3.4.

ispbsdir()

```
int ispbsdir(char *line)
```

Args:

line A line from the script file.

Returns:

True, if the line is a PBS directive, false otherwise. If it is a directive, it returns the starting address of line string.

Control Flow:

Does the first part of the line match the PBS directive prefix?

12.44.3.5.

get_script()

```
int get_script(FILE *file, char *script, char *prefix)
```

Args:

file The file descriptor of the script.

script The name of the copy that is made of the script that contains only non-PBS directive lines.

prefix The PBS directive prefix.

Returns:

Zero, if the script was copied okay and PBS options correctly parsed, non-zero otherwise.

Control Flow:

```
Create a temporary file
While there is a line left in the script file Do
    If no executable statements yet and this is a PBS directive Then
        Continuation is TRUE
        While Continuation D0
            Check if this line is continued (ends in \n) and
              if it is, Get the next line in the script and
              append to current directive line
        Done
       Parse the PBS directives
    Else
     Write the line to the temporary file
     If this an executable statement Then
        stop processing anymore PBS directive lines
     endif
```

12.44.3.6.

make_argv()

```
void make_argv(int *argc, char *argv[], char *line)
```

Args:

argc The number of PBS directives found in the line.

argv The individual PBS directives.

line The PBS directives line from the script.

Control Flow:

```
Set argv[0] to qsub
While the line is not empty Do
    If the next character is a quote Then
        Find the matching quote
        Make it a blank
    Scan for the next blank
    Allocate memory for the word
    Copy the word
    Put the word's address into the argv array
    Increment the number of things in argv
```

12.44.3.7.

do_dir()

```
int do_dir(char *line)
```

Args:

line A PBS directives line from the script.

Returns:

The value returned from processing the directives (see process_opts).

Control Flow:

If the first time through Then
    Clear out the array that will hold the words of the line
Parse the line into words
Process the word list

12.44.3.8.

process_opts()

```
int process_opts(int argc, char **argv, int pass)
```

Args:

argc The number of arguments in argv.

argv The command line or PBS directives line arguments.

pass Zero, if a command line argument list, positive if a PBS directive argument list.

Control Flow:

If pass is greater than zero Then
    Start at the beginning of the argument list
While getopt Do
    For each option, print corresponding "job attribute = value" string.

Note that the following rules are enforced:

1. Option argument values supplied on the command line take precedence over values for the same option supplied in script directives.
2. If an option is repeated on the command line (or in the script, subject to rules 1), the argument value for the last occurrence:
   - replaces the prior value if the option is singled valued (integer or string).
   - is appended to the prior value(s) if the option is list valued (comma separated elements).

12.45. xpbsmon Packaging

The main file called **xpbsmon** contains the main() section of the GUI; it starts up appropriate routines on its event loop to respond to actions like mouse presses. Related procedures, callback functions are grouped together in a file. Files with the ".tk" suffix contain Tk-related procedures while those with ".tcl" suffix contain non-Tk related routines. Bitmap files used by the GUI are located in the **bitmaps** directory. Help files accessed by the GUI are in the

help directory.

12.46. File: xpbsmon

This file contains the main event loop.

12.46.1.

main()

```
main{argc, argv}
```

Args:

argc The number of arguments on the command line.

argv The **argv** list contain the arguments

Returns:

Zero, if no command line syntax errors are detected. Positive, otherwise.

Control Flow:

set appropriate Tcl/Tk library directories, program version number, and program paths.

Load xpbsmon resource values as supplied from the X resources files: global and user's .xpbsmonrc file.
Set default values for unset xpbsmon resources.
Set the colors of various widgets based on xpbs color resources.
Save the initial values of the xpbs resources.

Also, for sites information, load it.

Set the mainWindow path and make it visible.

Build the main display of xpbsmon containing the necessary widgets.
Set properties involving the window manager.

if autoUpdate is turned on, then
    schedule an update of nodes data.
fi

12.47. File: node.tk

This file contains routines that creates a node box which represent a node or execution host on display.

12.47.1.

nodeCreate()

```
nodeCreate( nodeframe, nodename, nodelabel, clusterframe, nodeType, viewType )
```

Args:

nodeframe	a node abstraction
nodename	hostname of the node represented by 'nodeframe'
nodelabel	display label of the node represented by 'nodeframe'
clusterframe	the frame where 'nodeframe' belongs to
nodeType	type of node in terms of whether or not it is managed by a server and running a MOM (MOM_SNODE), managed by a server but is not running a MOM (NOMOM_SNODE), not managed by a server but is running a MOM (MOM), and finally, not managed by a server and is not running a MOM (NOMOM). NOMOM_SNODE is the default.
viewType	how nodeframe is going to be displayed: FULL, MIRROR, ICON.

Returns:

{$width $height} - the resulting width and height.

Control Flow:

Depending on the requested viewType, appropriately set the font types and maxWidth and maxHeight of the display box to use.

create the main nodeframe.
create the frame that will hold the nodelabel.
create the frame that will hold the canvas.
create the frame that will hold the x scrollbar.
create the frame that will hold the y scrollbar.

For the nodelabel, truncate it if the viewType is ICON. Calculate its width and height.

create the canvas widget. Update the nodeframe's node status to its current status if its nodeType is MOM and viewType is not MIRROR or if its nodeType is not MOM and viewType is not MIRROR. The default types are NOINFO for a node that is running a MOM, and UP for a node that is not running a MOM.
is not MIRROR. calculate its frame width and height.

create the Xscrollbar. Calculate its height.

create the Yscrollbar. Calculate its width.

Update the various attributes appropriately of the nodeframe's structure. The display width of the nodeframe will be adjusted according to the width of the nodelabel.

If the the display width is > maxWidth, adjust things so that they all fit in maxWidth. Do the same for display height.

If the viewType is ICON, then cover the contents of the canvas. Otherwise, display the canvas contents.

12.47.2.

nodeReCreate()

```
nodeReCreate( nodeframe )
```

Args:

nodeframe a node abstraction

Returns:

{$width $height} - the resulting width and height.

Control Flow:

Depending on the requested viewType, get values again for font types and maxWidth and maxHeight of the display box to use.

redisplay the nodelabel text.
reconfigure the nodeframe's canvas' width and height and scroll regions.
update the node's status when needed.
reconfigure the xscrollbar and the yscrollbar and redisplaying them or removing them from view when needed.

Finally, resize the display width and height depending on the new sizes of maxWidth and maxHeight.

If viewType is ICON, then cover the contents of the canvas.

12.47.3.

nodeAddWidth()

```
nodeAddWidth( nodeframe, incr )
```

Args:

nodeframe a node abstraction

incr amount to add to the nodeframe' width

Control Flow:

Adds 'incr' amount to the nodeframe's display width, and if the resulting value is still within the limits, then extend the width of the nodeframe's canvas, as well its cluster frame.

12.47.4.

nodeRepack()

```
nodeRepack( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:
redisplay a nodeframe,

if line scale width is > canvas width and is still
within the nodeMaxWidth boundary limit, then
set canvas width to that of line scale width,
and adjust node distances appropriately
fi
Also, show xscroll bar or yscroll when needed.

12.47.5.

nodeCoverCanvas()

```
nodeCoverCanvas( nodeframe )
```

Args:
nodeframe a node abstraction

Control Flow:
Puts a rectangular blanket over the canvas of a nodeframe.

12.47.6.

nodeUnCoverCanvas()

```
nodeUnCoverCanvas( nodeframe )
```

Args:
nodeframe a node abstraction

Control Flow:
Removes the rectangular blanket from the canvas of a nodeframe.

12.47.7.

nodeReCoverCanvas()

```
nodeReCoverCanvas( nodeframe )
```

Args:
nodeframe a node abstraction

Control Flow:
Like nodeCoverCanvas except the rectangular blanket is resized.

12.47.8.

nodeRefreshGet()

```
nodeRefreshGet( nodeframe }
```

Args:

nodeframe
a node abstraction

Control Flow:
get the refresh flag attribute value.

12.47.9.

nodeRefreshPut()

```
nodeRefreshPut ( nodeframe, flag )
```

Args:

nodeframe a node abstraction

flag new flag value

Control Flow:
update the refresh attribute value to 'flag'.

12.47.10.

nodeNameGet()

```
nodeNameGet( nodeframe }
```

Args:

nodeframe a node abstraction

Control Flow:
get the name attribute value.

12.47.11.

nodeNamePut()

```
nodeNamePut ( nodeframe, name )
```

Args:

nodeframe a node abstraction

name new nodename

Control Flow:

update the name attribute value to 'name'.

12.47.12.

nodeLabelFrameGet()

```
nodeLabelFrameGet ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

return the frame on which nodeframe's label widget sits on.

12.47.13.

nodeLabelFramePut()

```
nodeLabelFramePut ( nodeframe, frame )
```

Args:

nodeframe a node abstraction

frame a frame to put label widget

Control Flow:

set the frame on which nodeframe's label widget sits on to 'frame'.

12.47.14.

nodeLabelGet()

```
nodeLabelGet( nodeframe }
```

Args:

nodeframe a node abstraction

Returns:

Control Flow:

returns the pathname to the widget labeling the nodeframe.

12.47.15.

nodeLabelPut()

```
nodeLabelPut ( nodeframe, label )
```

Args:

nodeframe	a node abstraction
name	label widget pathname

Returns:

Control Flow:
makes 'label' the pathname to the nodeframe's label.

12.47.16.

nodeTypeGet()

```
nodeTypeGet( nodeframe }
```

Args:

nodeframe	a node abstraction

Returns:

Control Flow:
returns the type attribute value of nodeframe.

12.47.17.

nodeTypePut()

```
nodeTypePut ( nodeframe, type )
```

Args:

nodeframe	a node abstraction
type	new MOM type.

Control Flow:
updates the type attribute of nodeframe to 'type'.

12.47.18.

nodeViewTypeGet()

```
nodeViewTypeGet( nodeframe }
```

Args:

nodeframe a node abstraction

Control Flow:
returns the display/view type of nodeframe.

12.47.19.

nodeViewTypePut()

```
nodeViewTypePut ( nodeframe, type )
```

Args:
nodeframe a node abstraction
type new display type.

Returns:

Control Flow:
updates the display/view type of nodeframe to 'type'.

12.47.20.

nodeCanvasFrameGet()

```
nodeCanvasFrameGet ( nodeframe )
```

Args:
nodeframe a node abstraction

Control Flow:
returns the frame on which nodeframe's canvas sits on.

12.47.21.

nodeCanvasFramePut()

```
nodeCanvasFramePut ( nodeframe, frame )
```

Args:
nodeframe a node abstraction
frame a frame

Returns:

Control Flow:
set the frame on which nodeframe's canvas sits on to 'frame'.

12.47.22.

nodeCanvasGet()

```
nodeCanvasGet( nodeframe }
```

Args:
nodeframe a node abstraction
Control Flow:
returns the pathname to the canvas widget of 'nodeframe'.

12.47.23.

nodeCanvasPut()

```
nodeCanvasPut ( nodeframe, type )
```

Args:
nodeframe a node abstraction
canvas new canvas widget.
Control Flow:
makes 'canvas' the nodeframe's canvas widget.

12.47.24.

nodeXscrollFrameGet()

```
nodeXscrollFrameGet ( nodeframe )
```

Args:
nodeframe a node abstraction
Control Flow:
return the frame on which nodeframe's xscrollbar sits on.

12.47.25.

nodeXscrollFramePut()

```
nodeXscrollFramePut ( nodeframe, frame )
```

Args:

nodeframe a node abstraction

frame an xscroll frame

Control Flow:

set the frame on which nodeframe's xscrollbar sits on to 'frame'.

12.47.26.

nodeXscrollGet()

```
nodeXscrollGet( nodeframe }
```

Args:

nodeframe a node abstraction

Control Flow:

returns the pathname to the nodeframe's xscroll widget.

12.47.27.

nodeXscrollPut()

```
nodeXscrollPut ( nodeframe, xscroll )
```

Args:

nodeframe a node abstraction

xscroll xscrollbar pathname.

Returns:

Control Flow:

makes 'xscroll' pathname the nodeframe's xscroll widget.

12.47.28.

nodeYscrollFrameGet()

```
nodeYscrollFrameGet ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

return the frame on which nodeframe's yscrollbar sits on.

12.47.29.

nodeYscrollFramePut()

```
nodeYscrollFramePut ( nodeframe, frame )
```

Args:

nodeframe	a node abstraction
frame	a yscroll frame

Control Flow:
set the frame on which nodeframe's yscrollbar sits on to 'frame'.

12.47.30.

nodeYscrollGet()

```
nodeYscrollGet( nodeframe }
```

Args:

nodeframe	a node abstraction

Control Flow:
returns the pathname to the nodeframe's xscroll widget.

12.47.31.

nodeYscrollPut()

```
nodeYscrollPut ( nodeframe, yscroll )
```

Args:

nodeframe	a node abstraction
yscroll	yscrollbar pathname.

Control Flow:
makes 'yscroll' pathname the nodeframe's yscroll widget.

12.47.32.

nodeScrollRegionWidthGet()

```
nodeScrollRegionWidthGet( nodeframe }
```

Args:

nodeframe a node abstraction

Control Flow:
returns the current scrollregion's width in nodeframe's canvas.

12.47.33.

nodeScrollRegionWidthPut()

```
nodeScrollRegionWidthPut ( nodeframe, width )
```

Args:

nodeframe a node abstraction

width width of the scrollregion.

Control Flow:
records the scrollregion's width in nodeframe's canvas.

12.47.34.

nodeScrollRegionHeightGet()

```
nodeScrollRegionHeightGet( nodeframe }
```

Args:

nodeframe a node abstraction

Control Flow:
returns the scrollregion's height in nodeframe's canvas.

12.47.35.

nodeScrollRegionHeightPut()

```
nodeScrollRegionHeightPut ( nodeframe, height )
```

Args:

nodeframe a node abstraction

height height of the scrollregion.

Control Flow:
records the scrollregion's height in nodeframe's canvas.

12.47.36.

nodeDisplayWidthGet()

```
nodeDisplayWidthGet( nodeframe }
```

Args:

nodeframe	a node abstraction

Control Flow:
returns the displayWidth attribute value of nodeframe.

12.47.37.

nodeDisplayWidthPut()

```
nodeDisplayWidthPut ( nodeframe, width )
```

Args:

nodeframe	a node abstraction
width	width of display

Control Flow:
set the displayWidth's attribute value of nodeframe to 'width'.

12.47.38.

nodeDisplayHeightGet()

```
nodeDisplayHeightGet( nodeframe }
```

Args:

nodeframe	a node abstraction

Returns:

Control Flow:
returns the displayHeight attribute value of nodeframe.

12.47.39.

nodeDisplayHeightPut()

```
nodeDisplayHeightPut ( nodeframe, height )
```

Args:

nodeframe	a node abstraction
height	height of display

Control Flow:
set the displayHeight's attribute value of nodeframe to 'height'.

12.47.40.

nodeCanvasWidthGet()

```
nodeCanvasWidthGet( nodeframe }
```

Args:
nodeframe a node abstraction
Returns:

Control Flow:
returns the canvasWidth attribute value of nodeframe.

12.47.41.

nodeCanvasWidthPut()

```
nodeCanvasWidthPut ( nodeframe, width )
```

Args:
nodeframe a node abstraction
width width of canvas
Control Flow:
set the canvasWidth's attribute value of nodeframe to 'width'.

12.47.42.

nodeCanvasHeightGet()

```
nodeCanvasHeightGet( nodeframe }
```

Args:
nodeframe a node abstraction
Control Flow:
returns the canvasWidth attribute value of nodeframe.

12.47.43.

nodeCanvasHeightPut()

```
nodeCanvasHeightPut ( nodeframe, height )
```

Args:

nodeframe	a node abstraction
height	height of canvas

Control Flow:

set the canvasHeight's attribute value of nodeframe to 'height'.

12.47.44.

nodeClusterFrameGet()

```
nodeClusterFrameGet ( nodeframe )
```

Args:

nodeframe	a node abstraction

Control Flow:

return the cluster frame on which nodeframe sits on.

12.47.45.

nodeClusterFramePut()

```
nodeClusterFramePut ( nodeframe, frame )
```

Args:

nodeframe	a node abstraction
frame	a cluster frame

Control Flow:

set the cluster frame on which nodeframe sits on to 'frame'.

12.47.46.

nodeXposGet()

```
nodeXposGet ( nodeframe )
```

Args:

nodeframe	a node abstraction

Control Flow:

return the nodeframe's X coordinate.

12.47.47.

nodeXposPut()

```
nodeXposPut ( nodeframe, pt )
```

Args:

nodeframe	a node abstraction
pt	the x coordinate

Control Flow:
sets the nodeframe's X coordinate value to 'pt'.

12.47.48.

nodeYposGet()

```
nodeYposGet ( noderame )
```

Args:

nodeframe	a node abstraction

Control Flow:
return the nodeframe's Y coordinate.

12.47.49.

nodeYposPut()

```
nodeYposPut ( nodeframe, pt )
```

Args:

nodeframe	a node abstraction
pt	the y coordinate

Control Flow:
sets the nodeframe's Y coordinate value to 'pt'.

12.47.50.

nodeOffsetWidthGet()

```
nodeOffsetWidthGet ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:
returns the amount of pixels a node needs to move to the right during a redisplay.

12.47.51.

nodeOffsetWidthPut()

```
nodeOffsetWidthPut ( nodeframe, width )
```

Args:

nodeframe a node abstraction

width a width in pixels

Control Flow:
sets the amount of pixels to 'width' that a a node needs to move to the right.

12.47.52.

nodeNextGet()

```
nodeNextGet ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:
In an array of displayed nodeframes, this returns the its neighbor frame to the right.

12.47.53.

nodeNextPut()

```
nodeNextPut ( nodeframe, frame )
```

Args:

nodeframe a node abstraction

frame a neighboring frame

Control Flow:
sets nodeframe's neighbor frame to 'frame'.

12.47.54.

nodeMainFrameGet()

```
nodeMainFrameGet ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

returns the main frame where canvas, scrollbar of nodeframe sits.

12.47.55.

nodeMainFramePut()

```
nodeMainFramePut ( nodeframe, frame )
```

Args:

nodeframe a node abstraction

frame a frame to place canvas, scrollbars

Control Flow:

sets the main frame where canvas, scrollbar of nodeframe sits to 'frame'.

12.47.56.

nodeGroupXCGet()

```
nodeGroupXCGet ( nodeframe, group )
```

Args:

nodeframe a node abstraction

group a grouping idea within a nodeframe's canvas

Control Flow:

returns the X coordinate position of this group of stuff in nodeframe's canvas.

12.47.57.

nodeGroupXCPut()

```
nodeGroupXCPut ( nodeframe, group, xc )
```

Args:

nodeframe	a node abstraction
group	a grouping idea within a nodeframe's canvas
xc	the X coordinate of this group of stuff

Control Flow:

sets the X coordinate position of this group of stuff in nodeframe's canvas to 'xc'.

12.47.58.

nodeFindXCs()

```
nodeFindXCs ( nodeframe )
```

Args:

nodeframe	a node abstraction

Control Flow:

In nodeframe's canvas, find all the X coordinates per group of items.

Go through every line of data in nodeframe's canvas,
    foreach groups of data in a line,
        get X coordinate value for the current group,
        get maximum width value for the current group,

Go through each group,
    set the X coordinate of the "next" to be
      max(current X coordinate value for next group,
    (X coordinate for current group + max width value for current group)

Save the max X coordinate value for each group.

12.47.59.

nodeAdjustDisplay()

```
nodeAdjustDisplay ( nodeframe )
```

Args:

nodeframe	a node abstraction

Control Flow:

Call nodeFindXCs to get the aligned X coordinates for each group (column) of stuff in nodeframe's canvas.

Go through each item in nodeframe's canvas,
    find the group they belong to and see what their X coordinates
        supposed to be. Calculate an offset value of the item's

current X coordinate with the group X coordinate value. Then move the item appropriately using the newly-calculated offset.

Also, as you go through the loop, find the total height of the contents of the canvas as well as the total width. These will be used to update line scale width value as well scroll regions which would introduced the appropriate x scroll or yscroll.

12.47.60.

nodePrint()

```
nodePrint ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

prints the values of all the attributes of 'nodeframe'.

12.47.61.

nodeDelete()

```
nodeDelete( nodeframe )
```

Args:

nodeframe

Control Flow:

removes from view nodeframe and deallocates all memory storage associated with it.

12.47.62.

nodeLineGet()

```
nodeLineGet ( nodeframe, lineno, group )
```

Args:

nodeframe a node abstraction

lineno a line # in nodeframe's canvas

group a grouping of stuff in nodeframe's canvas

Control Flow:

return the item-id at (lineno,groupno) in nodeframe's canvas.

12.47.63.

nodeLinePut()

```
nodeLinePut ( nodeframe, lineno, group, tagOrId )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
group	a grouping of stuff in nodeframe's canvas
tagOrId	the item-id

Control Flow:

set the item-id at (lineno,groupno) in nodeframe's canvas to 'tagOrId'

12.47.64.

nodeAddText()

```
nodeAddText ( nodeframe, lineno, groupno, text )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
group	a grouping of stuff in nodeframe's canvas
text	text widget to add to nodeframe's canvas

Control Flow:

Based on the view type, set the text font to use appropriately.
The idea is to separate text widgets into different groups. Each group of text is identified by some item id or tag. This id will be saved for future manipulation.

Indent the text by text_width and text_height if it is the 1st group of text in the line. Simple start the text after the previous line if one exists; otherwise, simple add to the current line.

12.47.65.

nodeAddLineText()

```
nodeAddLineText ( nodeframe lineno listOfText )
```

Args:

nodeframe 12 a node abstraction

lineno	a line # in nodeframe's canvas
listOfText	list of text to add

Control Flow:
adds a "listOfText" as one line in nodeframe's canvas.

12.47.66.

nodeMatchItemTag()

```
nodeMatchItemTag( nodeframe, itemid, tag )
```

Args:

nodeframe	a node abstraction
itemid	an id in nodeframe's canvas
tag	tag to match

Returns:
1 if matching "tag" is found 0 otherwise

Control Flow:
searches for a tag in 'itemid' that matches "tag" and returns 1 if found one, 0 otherwise.

12.47.67.

nodeModText()

```
nodeModText ( nodeframe, lineno, groupno, text )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
group	a grouping of stuff in nodeframe's canvas
text	text widget to add to nodeframe's canvas

Returns:
1 if a modify action took place 0 otherwise

Control Flow:
Get the id @lineno,groupno in nodeframe's canvas. If its tag is a text, then modify its text value if != "text"; otherwise, delete this current text and call "nodeAddText" to add a new text.

12.47.68.

nodeRemLineEntry()

```
nodeRemLineEntry(nodeframe, lineno, groupno)
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
group	a grouping of stuff in nodeframe's canvas

Control Flow:

Starting at @lineno,groupno, text will be deleted until the end of line, and any textid saved somewhere for the text will be thrown away.

12.47.69.

nodeModLineText()

```
nodeModLineText ( nodeframe, lineno, listOfText )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
listOfText	list of text to modify

Control Flow:

like nodeModText except that modification applies to a line of text. This will completely replace all text found @lineno.

12.47.70.

nodeRemLines()

```
nodeRemLines ( nodeframe, lineno )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas

Control Flow:

This removes all the lines in nodeframe's canvas starting @lineno.

12.47.71.

nodeScaleCreate()

```
nodeScaleCreate ( nodeframe lineno groupno value )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
group	a grouping of stuff in nodeframe's canvas
value	scale value

Control Flow:

create the text value of the scale. Position it appropriately.
create the rectangle for the max value.
if value > max value; then
create 2 filled rectangles. One for the scale value of up to max value. The other is for any of remaining value in excess of the max value.
else
create 1 rectangle only for the scale value.

Mark all parts of the scale as "$lineno$groupno$nodeframe" and save this tag for future collective manipulation.

12.47.72.

nodeScaleReCreate()

```
nodeScaleReCreate ( nodeframe lineno groupno newvalue )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
group	a grouping of stuff in nodeframe's canvas
newvalue	scale value

Control Flow:

This will itemconfigure the label for the value of the widget. All the other rectangles are appropriately recreated.

12.47.73.

nodeAddLineScale()

```
nodeAddLineScale ( nodeframe, lineno, param )
```

Args:

nodeframe	a node abstraction
lineno	a line # in nodeframe's canvas
param	{label valueLabel}

Control Flow:

creates a line containing a scale widget given by 'param'.

12.47.74.

nodeModLineScale()

```
nodeModLineScale ( nodeframe, lineno, newContent )
```

Args:

nodeframe a node abstraction

lineno a line # in nodeframe's canvas

newContent new Content of the form {label valueLabel}

Control Flow:

This recreates the line that has a scale widget using 'newContent' as specification.

12.47.75.

nodeDown()

```
nodeDown ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

This changes the background color of the node canvas to $canvas(nodeColorDown)

12.47.76.

nodeUp()

```
nodeUp ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

This changes the background color of the node canvas to $canvas(nodeColorUP)

12.47.77.

nodeOffline()

```
nodeOffline ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

This changes the background color of the node canvas to $canvas(nodeColorOFFL)

12.47.78.

nodeReserved()

```
nodeReserved ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

This changes the background color of the node canvas to $canvas(nodeColorRSVD)

12.47.79.

nodeInUse()

```
nodeInUse ( nodeframe )
```

Args:

nodeframe a node abstraction

Control Flow:

This is a little less straightforward procedure because different unique
jobs can be assigned different colors.
First, it looks at the nodeInfo of the nodename represented by 'nodeframe'. If
the nodeInfo contains a NODEJOB, that means the node is INUSE. Then get the
userinfo = { {user1 {j11 j12 ... j1n}}, {user2 { j21 j22 ... j2n} ...}, create
jobs = {user1.j11 user1.j12 ... user1.j1n user2.j21 user2.j22 ... user2.j2n ..}.
if nodestatus is INUSE-EXCLUSIVE and length(jobs) <= 1, then
go head and assign a new INUSE color
else if length(jobs) > 1, then
set INUSE color to be $canvas(nodeColorINUSEshared)

change the background color of the nodeframe's canvas to whatever the chosen
color if one exists.

12.47.80.

nodeUpdateStat()

```
nodeUpdateStat ( sysframe, nodeid, status2, defstat )
```

Args:

sysframe	a site/system abstraction
nodeid	name of node in a site
status2	new node status
defstat	default node status (=NOINFO)

Control Flow:

This basically sets the status of node described by nodeid @ sysframe to 'defstat'. It also does other bookkeepings like:

save oldstatus

if status is "" ; then
    set status to $defstat
fi

save status by calling the function systemNodeStatusPut()

To do the following, call the function clusterStatsUpdate():

status	action
OFFLINE	update the clusterOfflinePool count
DOWN	update the clusterDownPool count
FREE	update the clusterAvailPool count
INUSE	update the clusterUsePool count
RESERVED	update the clusterReservedPool count

oldstatus	action
OFFLINE	decrement the clusterOfflinePool count by 1
DOWN	decrement the clusterDownPool count by 1
FREE	decrement the clusterAvailPool count by 1
INUSE	decrement the clusterUsePool count by 1
RESERVED	decrement the clusterReservedPool count by 1

12.47.81.

nodeDisplayInfo()

```
nodeDisplayInfo (nodeframe, queryInfo, create)
```

Args:

nodeframe	a nodeframe abstraction
queryInfo	information to be displayed
create	flag to signal whether to create the frame or modify contents of existing frame (default = 0)

Control Flow:

set i 0
foreach elem in queryInfo
do
    if elem's type is TEXT, then

```
            if create is TRUE
                add ith-line of text featuring elem's header and result
            else
                mod ith-line of text featuring elem's header and result
        else if elem's type is SCALE, then
            if create is TRUE
                add ith-line of scale featuring elem's header and result
            else
                mod ith-line of scale featuring elem's header and result
        else if elem's type is NODEJOB, then
            if the i-th line is empty, then
                add ith-line of text featuring elem's header
            else
                mod ith-line of text featuring elem's header
            fi
    cluster.tk
            set k = $i + 1
    cluster.tk
            foreach of the user:jobs info,
            do
                if the k-th line of text is empty, then
                    add kth-line of text indented
                else
                    modify the kth-line of text indented
                fi
                incr k
            done
        fi

        increment i
    done
    if no lines of text containing NODEJOB information,
        reset k to current value of i

    remove remaining k lines of nodeframe

    adjust the display of nodeframe to see if xscroll and yscroll bars are needed
    cover the contents if nodeframe's view type is ICON; otherwise, uncover the
    contents.
```

12.48. File: cluster.tk

This file contains routines for displaying the server box.

12.48.1.

clusterAddWidth()

```
clusterAddWidth( clusterf, incr )
```

Args:

clusterf a cluster frame abstraction

incr amount to add to the clusterf' width

Control Flow:

Adds 'incr' amount to the clusterf's display width, and if the resulting value is still within the limits, then extend the width of the clusterf's canvas, and the enclosing system frame. Update the clusterf's scroll region width appropriately.

12.48.2.

clusterPropagateOffset()

```
clusterPropagateOffset( clusterf diff )
```

Args:

clusterf
a cluster frame abstraction

diff offset value to be propagated

Control Flow:

Updates the offset values of all the cluster frames following 'clusterf'.

12.48.3.

clusterDelete()

```
clusterDelete ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

removes from view clusterf and deallocates all memory storage associated with it.

12.48.4.

clusterNamePut()

```
clusterNamePut ( clusterframe, name )
```

Args:

clusterframe a cluster frame abstraction

name name for the cluster

Control Flow:

set the name attribute of cluster to 'name'.

12.48.5.

clusterNameGet()

```
clusterNameGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of clusterframe's name attribute.

12.48.6.

clusterCanvasFramePut()

```
clusterCanvasFramePut ( clusterframe, frame )
```

Args:

clusterframe a cluster frame abstraction

frame canvas frame widget

Control Flow:

set the canvasFrame attribute of cluster to 'frame'.

12.48.7.

clusterCanvasFrameGet()

```
clusterCanvasFrameGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Returns:

Control Flow:

returns the value of clusterframe's canvasFrame attribute.

12.48.8.

clusterCanvasPut()

```
clusterCanvasPut ( clusterframe, canvas )
```

Args:

clusterframe	a cluster frame abstraction
canvas	canvas frame widget

Control Flow:
set the canvas frame attribute of cluster to 'canvas'.

12.48.9.

clusterCanvasGet()

```
clusterCanvasGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the value of clusterframe's canvas attribute.

12.48.10.

clusterRefreshPut()

```
clusterRefreshPut ( clusterframe, flag )
```

Args:

clusterframe	a cluster frame abstraction
flag	do a refresh? flag

Control Flow:
set the refresh flag attribute of cluster to 'flag'.

12.48.11.

clusterRefreshGet()

```
clusterRefreshGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Returns:

Control Flow:
returns the value of clusterframe's refresh attribute.

12.48.12.

clusterLabelFramePut()

```
clusterLabelFramePut ( clusterframe, frame )
```

Args:

clusterframe	a cluster frame abstraction
frame	name of a frame

Control Flow:
set the labelFrame widget pathname of clusterframe to 'frame'.

12.48.13.

clusterLabelFrameGet()

```
clusterLabelFrameGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the labelFrame widget pathname of clusterframe.

12.48.14.

clusterLabelPut()

```
clusterLabelPut ( clusterframe, label )
```

Args:

clusterframe	a cluster frame abstraction
label	label widget pathname

Control Flow:
set the label widget pathname of clusterframe to 'label'.

12.48.15.

clusterLabelGet()

```
clusterLabelGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:
returns the label widget pathname of clusterframe.

12.48.16.

clusterLabelTextPut()

```
clusterLabelTextPut ( clusterframe, text )
```

Args:

clusterframe a cluster frame abstraction

text text of clusterframe's label widget

Control Flow:
set the text for the label widget of clusterframe to 'text'.

12.48.17.

clusterLabelGet()

```
clusterLabelGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:
returns the text of clusterframe's label widget.

12.48.18.

clusterFooterHeaderPut()

```
clusterFooterHeaderPut ( clusterframe, label )
```

Args:

clusterframe a cluster frame abstraction

label text of clusterframe's footer label

Control Flow:
set the text for the footer label widget of clusterframe to 'label'.

12.48.19.

clusterFooterHeaderGet()

```
clusterFooterHeaderGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the text of clusterframe's footer label widget.

12.48.20.

clusterStatusBarFramePut()

```
clusterStatusBarFramePut ( clusterframe, frame )
```

Args:

clusterframe a cluster frame abstraction

frame frame widget

Control Flow:

set the statusBarFrame attribute of cluster to 'frame'.

12.48.21.

clusterStatusBarFrameGet()

```
clusterStatusBarFrameGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of clusterframe's statusBarFrame attribute.

12.48.22.

clusterStatusBarPut()

```
clusterStatusBarPut ( clusterframe, statusBar )
```

Args:

clusterframe a cluster frame abstraction

statusBar footer label widget pathname

Control Flow:

set the footer label widget pathname of clusterframe to 'statusBar'.

12.48.23.

clusterStatusBarGet()

```
clusterStatusBarGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the footer label widget pathname.

12.48.24.

clusterXscrollPut()

```
clusterXscrollPut ( clusterframe, xscroll )
```

Args:

clusterframe a cluster frame abstraction

xscroll x scrollbar widget pathname

Control Flow:

set the xscrollbar widget pathname of clusterframe to 'xscroll'.

12.48.25.

clusterXscrollGet()

```
clusterXscrollGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the xscrollbar widget pathname of clusterframe.

12.48.26.

clusterYscrollPut()

```
clusterYscrollPut ( clusterframe, yscroll )
```

Args:

clusterframe a cluster frame abstraction

yscroll y scrollbar widget pathname

Control Flow:
set the yscrollbar widget pathname of clusterframe to 'yscroll'.

12.48.27.

clusterYscrollGet()

```
clusterYscrollGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:
returns the yscrollbar widget pathname of clusterframe.

12.48.28.

clusterXscrollFramePut()

```
clusterXscrollFramePut ( clusterframe, frame )
```

Args:

clusterframe a cluster frame abstraction

frame frame widget

Control Flow:
set the xscrollFrame attribute of cluster to 'frame'.

12.48.29.

clusterXscrollFrameGet()

```
clusterXscrollFrameGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:
returns the value of clusterframe's xscrollFrame attribute.

12.48.30.

clusterYscrollFramePut()

```
clusterYscrollFramePut ( clusterframe, frame )
```

Args:

clusterframe	a cluster frame abstraction
frame	canvas frame widget

Control Flow:
set the yscrollFrame attribute of cluster to 'frame'.

12.48.31.

clusterYscrollFrameGet()

```
clusterYscrollFrameGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the value of clusterframe's yscrollFrame attribute.

12.48.32.

clusterDisplayWidthPut()

```
clusterDisplayWidthPut ( clusterframe, width )
```

Args:

clusterframe	a cluster frame abstraction
width	width of display

Control Flow:
set the displayWidth attribute of clusterframe to 'width'.

12.48.33.

clusterDisplayWidthGet()

```
clusterDisplayWidthGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the value of the displayWidth attribute of clusterframe.

12.48.34.

clusterDisplayHeightPut()

```
clusterDisplayHeightPut ( clusterframe, height )
```

Args:

clusterframe	a cluster frame abstraction
height	height of display

Control Flow:
set the displayHeight attribute of clusterframe to 'height'.

12.48.35.

clusterDisplayHeightGet()

```
clusterDisplayHeightGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the value of the displayHeight attribute of clusterframe.

12.48.36.

clusterCanvasWidthPut()

```
clusterCanvasWidthPut ( clusterframe, width )
```

Args:

clusterframe	a cluster frame abstraction
width	width of canvas

Control Flow:
set the canvasWidth attribute of clusterframe to 'width'.

12.48.37.

clusterCanvasWidthGet()

```
clusterCanvasWidthGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the value of the canvasWidth attribute of clusterframe.

12.48.38.

clusterCanvasHeightPut()

```
clusterCanvasHeightPut ( clusterframe, height )
```

Args:

clusterframe	a cluster frame abstraction
height	height of canvas

Control Flow:

set the canvasHeight attribute of clusterframe to 'height'.

12.48.39.

clusterCanvasHeightGet()

```
clusterCanvasHeightGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:

returns the value of the canvasHeight attribute of clusterframe.

12.48.40.

clusterScrollRegionWidthPut()

```
clusterScrollRegionWidthPut ( clusterframe, width )
```

Args:

clusterframe	a cluster frame abstraction
width	width of scrollRegion

Control Flow:

set the scrollRegionWidth attribute of clusterframe to 'width'.

12.48.41.

clusterScrollRegionWidthGet()

```
clusterScrollRegionWidthGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of the scrollRegionWidth attribute of clusterframe.

12.48.42.

clusterScrollRegionHeightPut()

```
clusterScrollRegionHeightPut ( clusterframe, height )
```

Args:

clusterframe a cluster frame abstraction

height height of scrollRegion

Control Flow:

set the scrollRegionHeight attribute of clusterframe to 'height'.

12.48.43.

clusterScrollRegionHeightGet()

```
clusterScrollRegionHeightGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of the scrollRegionsHeight attribute of clusterframe.

12.48.44.

clusterXposPut()

```
clusterXposPut ( clusterframe, pos )
```

Args:

clusterframe a cluster frame abstraction

pos width of scrollRegion

Control Flow:

set the Xpos attribute of clusterframe to 'pos'.

12.48.45.

clusterXposGet()

```
clusterXposGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of the Xpos attribute of clusterframe.

12.48.46.

clusterYposPut()

```
clusterYposPut ( clusterframe, pos )
```

Args:

clusterframe a cluster frame abstraction

height height of scrollRegion

Control Flow:

set the Ypos attribute of clusterframe to 'pos'.

12.48.47.

clusterYposGet()

```
clusterYposGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of the Ypos attribute of clusterframe.

12.48.48.

clusterNextGet()

```
clusterNextGet ( nodeframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

In an array of displayed clusterframes, this returns its neighbor frame to the right.

12.48.49.

clusterNextPut()

```
clusterNextPut ( clusterframe, frame )
```

Args:

clusterframe	a cluster frame abstraction
frame	a neighboring frame

Control Flow:
sets clusterframe's neighbor frame to 'frame'.

12.48.50.

clusterOffsetWidthPut()

```
clusterOffsetWidthPut ( clusterframe, width )
```

Args:

clusterframe	a cluster frame abstraction
width	the offset width

Control Flow:
set the offset width to 'width' of clusterframe.

12.48.51.

clusterOffsetWidthGet()

```
clusterOffsetWidthGet ( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the offset width of the clusterframe which is the value to move the clusterframe to the right when a refreshDisplay is done.

12.48.52.

clusterMainFramePut()

```
clusterMainFramePut ( clusterframe, frame )
```

Args:

clusterframe a cluster frame abstraction

frame main frame widget

Control Flow:

set the mainFrame attribute of cluster to 'frame'.

12.48.53.

clusterMainFrameGet()

```
clusterMainFrameGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of clusterframe's mainFrame attribute.

12.48.54.

clusterSystemFramePut()

```
clusterSystemFramePut ( clusterframe, frame )
```

Args:

clusterframe a cluster frame abstraction

frame system frame widget

Control Flow:

set the systemFrame attribute of cluster to 'frame'.

12.48.55.

clusterSystemFrameGet()

```
clusterSystemFrameGet ( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the value of clusterframe's systemFrame attribute.

12.48.56.

clusterNodesListPut()

```
clusterNodesListPut( clusterframe, nlist )
```

Args:

clusterframe	a cluster frame abstraction
nlist	list of nodenames in clusterframe

Control Flow:
set the nodeslist of clusterframe to 'nlist'.

12.48.57.

clusterNodesListGet()

```
clusterNodesListGet( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the nodeslist attribute of clusterframe.

12.48.58.

clusterTotPoolPut()

```
clusterTotPoolPut( clusterframe, totpool )
```

Args:

clusterframe	a cluster frame abstraction
totpool	# of nodes in the pool of clusterframe

Control Flow:
set totpool of clusterframe to 'totpool'.

12.48.59.

clusterTotPoolGet()

```
clusterTotPoolGet( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:
returns the # of nodes in the pool of clusterframe.

12.48.60.

clusterUsePoolPut()

```
clusterUsePoolPut( clusterframe, usepool )
```

Args:

clusterframe	a cluster frame abstraction
usepool	# of used nodes in the pool of clusterframe

Control Flow:

set usepool attribute of clusterframe to 'usepool'.

12.48.61.

clusterUsePoolGet()

```
clusterUsePoolGet( clusterframe )
```

Args:

clusterframe	a cluster frame abstraction

Control Flow:

returns the # of used nodes in the pool of clusterframe.

12.48.62.

clusterAvailPoolPut()

```
clusterAvailPoolPut( clusterframe, availpool )
```

Args:

clusterframe	a cluster frame abstraction
availpool	# of available nodes in the pool of clusterframe

Control Flow:

set availpool attribute of clusterframe to 'availpool'.

12.48.63.

clusterAvailPoolGet()

```
clusterAvailPoolGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the # of available nodes in the pool of clusterframe.

12.48.64.

clusterOfflinePoolPut()

```
clusterOfflinePoolPut( clusterframe, offlpool )
```

Args:

clusterframe a cluster frame abstraction

offlpool # of offline nodes in the pool of clusterframe

Control Flow:

set offlpool attribute of clusterframe to 'offlpool'.

12.48.65.

clusterOfflinePoolGet()

```
clusterOfflinePoolGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the # of offline nodes in the pool of clusterframe.

12.48.66.

clusterDownPoolPut()

```
clusterDownPoolPut( clusterframe, downpool )
```

Args:

clusterframe a cluster frame abstraction

downpool # of down nodes in the pool of clusterframe

Control Flow:

set downpool attribute of clusterframe to 'downpool'.

12.48.67.

clusterDownPoolGet()

```
clusterDownPoolGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the # of down nodes in the pool of clusterframe.

12.48.68.

clusterReservedPoolPut()

```
clusterReservedPoolPut( clusterframe, rsrvpool )
```

Args:

clusterframe a cluster frame abstraction

rsrvpool # of reserved nodes in the pool of clusterframe

Control Flow:

set rsrvpool attribute of clusterframe to 'rsrvpool'.

12.48.69.

clusterReservedPoolGet()

```
clusterReservedPoolGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the # of reserved nodes in the pool of clusterframe.

12.48.70.

clusterUnkPoolPut()

```
clusterUnkPoolPut( clusterframe, unkpool )
```

Args:

clusterframe a cluster frame abstraction

unkpool # of unknown status nodes in the pool of clusterframe

Control Flow:

set unkpool attribute of clusterframe to 'unkpool'.

12.48.71.

clusterUnkPoolGet()

```
clusterUnkPoolGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the # of unknown status nodes in the pool of clusterframe.

12.48.72.

clusterCpusAssnPut()

```
clusterCpusAssnPut( clusterframe, cpus_assn )
```

Args:

clusterframe a cluster frame abstraction

cpus_assn # of cpus assigned in clusterframe

Control Flow:

set cpus_assn attribute of clusterframe to 'cpus_assn'.

12.48.73.

clusterCpusAssnGet()

```
clusterCpusAssnGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the # of cpus assigned in clusterframe.

12.48.74.

clusterCpusMaxPut()

```
clusterCpusMaxPut( clusterframe, cpus_max )
```

Args:

clusterframe a cluster frame abstraction

cpus_max max # of cpus in clusterframe

Control Flow:

set cpus_max attribute of clusterframe to 'cpus_max'.

12.48.75.

clusterCpusMaxGet()

```
clusterCpusMaxGet( clusterframe )
```

Args:

clusterframe a cluster frame abstraction

Control Flow:

returns the max # of cpus in clusterframe.

12.48.76.

clusterPrint()

```
clusterPrint ( clusterframe )
```

Args:

clusterframe a cluster abstraction

Control Flow:

prints the values of all the attributes of 'clusterframe'.

12.48.77.

clusterCreate()

```
clusterCreate (frame, clusterName, clusterLabel, nlist, footerHeader, viewType)
```

Args:

frame a frame to create the cluster on

clusterName name for the cluster

clusterLabel display label

nlist list of nodes under the cluster

footerHeader the footer string for the cluster

viewType the viewType {ICON, FULL, MIRROR}

Control Flow:

foreach node in nlist
do

get nodename, nodetype, nodelabel
update the cluster status counts
create the node
if there's a previous node, set that node's next neighbor to the newly-created node.
keep positioning the nodes in one row of the cluster until you hit either one of the following conditions:
i. $canvas(clusterNumBoxesPerRow) has been reached
ii. if adding the new node will result in the $canvas(clusterMaxWidth) to be reached
Hitting conditions i or ii means to start a new column in the cluster frame.
Record the Xpos and Ypos position of all the nodes
done

Create the Xscrollbar
Create the Yscrollbar
Create the footer label, frame

This would shrink the display view if total width is > clusterMaxWidth

Initialize various attribute values of cluster
pack xscroll if needed
pack yscroll if needed
pack the rest of clusterframe's parts to create a view

12.48.78.

clusterReCreate()

```
clusterReCreate ( frame )
```

Args:

frame the cluster frame to recreate

Control Flow:

relabel the display label of the cluster
relabel the footer Label

recreate the nodes of the cluster, by packing in the same as in clusterCreate.

12.48.79.

clusterStatsUpdate()

```
clusterStatsUpdate ( clusterf, status, oper )
```

Args:

clusterfthe cluster frame to update status counts

status the status in question

oper the operator (i.e. +, -)

Control Flow:

Based on the current status, whether be OFFLINE, DOWN, FREE, INUSE_SHARED, INUSE-EXCLUSIVE, RESERVED, NOINFO, keep track of the various status counts internally by applying 'oper'.

12.48.80.

clusterRepack()

```
clusterRepack (clusterframe)
```

Args:

clusterframe frame to repack

Control Flow:

Repacking a frame means removing the current frame in view, and redisplaying it making appropriate adjustments to widths and heights. xscrollbar and yscrollbar will show up as needed.

12.49. File: system.tk

This file contains the routines for displaying the site box.

12.49.1.

systemAddWidth()

```
systemAddWidth( systemf, incr )
```

Args:

systemf a system frame abstraction

incr amount to add to the systemf' width

Control Flow:

Adds 'incr' amount to the systemf's display width, and if the resulting value is still within the limits, then extend the width of the systemf's canvas. Update the systemf's scroll region width appropriately.

12.49.2.

systemRefreshPut()

```
systemRefreshPut( systemframe, flag )
```

Args:

systemframe a system frame abstraction

flag the refresh flag

Control Flow:

sets the systemframe's refresh flag to 'flag'.

12.49.3.

systemRefreshGet()

```
systemRefreshGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

return the systemframe's refresh flag.

12.49.4.

systemNamePut()

```
systemNamePut( systemframe, name )
```

Args:

systemframe a system frame abstraction

name name associated with systemframe

Control Flow:

sets the systemframe's name to 'name'.

12.49.5.

systemNameGet()

```
systemNameGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

return the systemframe's name attribute.

12.49.6.

systemNodeFramePut()

```
systemNodeFramePut ( systemframe, nodename, frame, frametype )
```

Args:

systemframe	a system frame abstraction
nodename	a node enclosed inside a systemframe
frame	frame widget pathname
frametype	type of frame (ICON, FULL, MIRROR)

Control Flow:
sets the frame enclosed in systemframe for nodename to 'frame' with 'frametype'.

12.49.7.

systemNodeFrameGet()

```
systemNodeFrameGet ( systemframe, nodename, frametype  )
```

Args:

systemframe	a system frame abstraction
nodename	a node enclosed inside a systemframe
frametype	type of frame (ICON, FULL, MIRROR)

Control Flow:
returns the frame enclosed in systemframe for nodename of 'frametype'.

12.49.8.

systemNodeFrameUnset()

```
systemNodeFrameUnset ( systemframe, nodename, frametype )
```

Args:

systemframe	a system frame abstraction
nodename	a node enclosed inside a systemframe
frametype	type of frame (ICON, FULL, MIRROR)

Control Flow:
deallocates storage used for saving the 'frame' of 'nodename' on 'systemframe'.

12.49.9.

systemNodeStatusPut()

```
systemNodeStatusPut ( systemframe, nodename, stat )
```

Args:

systemframe a system frame abstraction

nodename a node enclosed inside a systemframe

stat status (i.e. UP, DOWN, OFFLINE, RESERVED, NOINFO, INUSE)

Control Flow:

sets the status of nodename under systemframe to 'stat'.

12.49.10.

systemNodeStatusGet()

```
systemNodeStatusGet ( systemframe, nodename )
```

Args:

systemframe a system frame abstraction

nodename a node enclosed inside a systemframe

Control Flow:

returns the status of nodename under systemframe.

12.49.11.

systemNodeStatusUnset()

```
systemNodeStatusUnset ( systemframe, nodename )
```

Args:

systemframe a system frame abstraction

nodename a node enclosed inside a systemframe

Control Flow:

deallocates the storage used for saving the status information for nodename at systemframe.

12.49.12.

systemNodeNamesGet()

```
systemNodeNamesGet ( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the list of nodes known under 'systemframe'.

12.49.13.

systemNodeInfoPut()

```
systemNodeInfoPut ( systemframe, nodename, info )
```

Args:

systemframe	a system frame abstraction
nodename	a name of the node enclosed by systemframe
info	information about the node

Control Flow:

sets the information about the node known by 'nodename' in 'systemframe' to 'info'.

12.49.14.

systemNodeInfoAppend()

```
systemNodeInfoAppend ( systemframe, nodename, info )
```

Args:

systemframe	a system frame abstraction
nodename	a name of the node enclosed by systemframe
info	information about the node

Control Flow:

appends the information 'info' for the node known by 'nodename' in 'systemframe'.

12.49.15.

systemNodeInfoGet()

```
systemNodeInfoGet ( systemframe, nodename )
```

Args:

systemframe	a system frame abstraction
nodename	a name of the node enclosed by systemframe

Control Flow:

returns information about the node 'nodename' under 'systemframe'.

12.49.16.

systemNodeInfoUnset()

```
systemNodeInfoUnset ( systemframe, nodename )
```

Args:

systemframe a system frame abstraction

nodename a name of the node enclosed by systemframe

Control Flow:

deallocates the storage used for saving the information for nodename at systemframe.

12.49.17.

systemNodeInfo2Append()

```
systemNodeInfo2Append ( systemframe, nodename, info )
```

Args:

systemframe a system frame abstraction

nodename a name of the node enclosed by systemframe

info information about the node

Control Flow:

appends the information 'info' for the node known by 'nodename' in 'systemframe'.

12.49.18.

systemNodeInfo2Get()

```
systemNodeInfo2Get ( systemframe, nodename )
```

Args:

systemframe a system frame abstraction

nodename a name of the node enclosed by systemframe

Control Flow:

returns information about the node 'nodename' under 'systemframe'.

12.49.19.

systemNodeInfo2Unset()

```
systemNodeInfo2Unset ( systemframe, nodename )
```

Args:

systemframe	a system frame abstraction
nodename	a name of the node enclosed by systemframe

Control Flow:

deallocates the storage used for saving the information for nodename at systemframe.

12.49.20.

systemNodeTypePut()

```
systemNodeTypePut ( systemframe, hostname, type )
```

Args:

systemframe	a system frame abstraction
hostname	a name of a host
type	type of host/node

Control Flow:

sets the hostname at systemframe's type to 'type'.

12.49.21.

systemNodeTypeGet()

```
systemNodeTypeGet ( systemframe, hostname )
```

Args:

systemframe	a system frame abstraction
hostname	a name of a host

Control Flow:

returns the 'type' of hostname at systemframe.

12.49.22.

systemClusterFramePut()

```
systemClusterFramePut( systemframe, name, frame )
```

Args:

systemframe	a system frame abstraction
name	a name of a node

frame frame widget

Control Flow:
sets the node name under systemframe to 'frame'.

12.49.23.

systemClusterFramGet()

```
systemClusterFramGet( systemframe, name )
```

Args:

systemframe a system frame abstraction

name a name of a node

Control Flow:
returns the cluster frame of node named by 'name' under systemframe.

12.49.24.

systemClusterFrameUnset()

```
systemClusterFrameUnset ( systemframe, name )
```

Args:

systemframe a system frame abstraction

name a name of a node

Control Flow:
deallocates the storage used to save the cluster frame of node named 'name' under 'systemframe'.

12.49.25.

systemClusterNamesGet()

```
systemClusterNamesGet ( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:
returns the list of names for the clusterframe known to 'systemframe'.

12.49.26.

systemCanvasPut()

```
systemCanvasPut( systemframe, frame )
```

Args:

systemframe a system frame abstraction

frame frame widget pathname

Control Flow:

sets the canvas attribute of 'systemframe' to 'frame'.

12.49.27.

systemCanvasGet()

```
systemCanvasGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the canvas attribute of 'systemframe'.

12.49.28.

systemDisplayWidthPut()

```
systemDisplayWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:

sets the displayWidth attribute of 'systemframe' to 'width'.

12.49.29.

systemDisplayWidthGet()

```
systemDisplayWidthGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the displayWidth attribute of 'systemframe'.

12.49.30.

systemDisplayHeightPut()

```
systemDisplayHeightPut( systemframe, height)
```

Args:

systemframe a system frame abstraction

height height in pixels

Control Flow:

sets the displayHeight attribute of 'systemframe' to 'height'.

12.49.31.

systemDisplayHeightGet()

```
systemDisplayHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the displayHeight attribute of 'systemframe'.

12.49.32.

systemScrollRegionWidthPut()

```
systemScrollRegionWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:

sets the scrollRegionWidth attribute of 'systemframe' to 'width'.

12.49.33.

systemScrollRegionWidthGet()

```
systemScrollRegionWidthGet( systemframe )
```

Args:

systemframe
a system frame abstraction

Returns:

Control Flow:
returns the value to the scrollRegionWidth attribute of 'systemframe'.

12.49.34.

systemScrollRegionHeightPut()

```
systemScrollRegionHeightPut( systemframe, height)
```

Args:

systemframe
a system frame abstraction

heightheight in pixels

Returns:

Control Flow:
sets the scrollRegionHeight attribute of 'systemframe' to 'height'.

12.49.35.

systemScrollRegionHeightGet()

```
systemScrollRegionHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:
returns the value to the scrollRegionHeight attribute of 'systemframe'.

12.49.36.

systemCanvasFrameWidthPut()

```
systemCanvasFrameWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:
sets the canvasFrameWidth attribute of 'systemframe' to 'width'.

12.49.37.

systemCanvasFrameWidthGet()

```
systemCanvasFrameWidthGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the canvasFrameWidth attribute of 'systemframe'.

12.49.38.

systemCanvasFrameHeightPut()

```
systemCanvasFrameHeightPut( systemframe, height)
```

Args:

systemframe a system frame abstraction

height height in pixels

Control Flow:

sets the canvasFrameHeight attribute of 'systemframe' to 'height'.

12.49.39.

systemCanvasFrameHeightGet()

```
systemCanvasFrameHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the canvasFrameHeight attribute of 'systemframe'.

12.49.40.

systemCanvasWidthPut()

```
systemCanvasWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:
sets the canvasWidth attribute of 'systemframe' to 'width'.

12.49.41.

systemCanvasWidthGet()

```
systemCanvasWidthGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:
returns the value to the canvasWidth attribute of 'systemframe'.

12.49.42.

systemCanvasHeightPut()

```
systemCanvasHeightPut( systemframe, height)
```

Args:

systemframe a system frame abstraction

height height in pixels

Control Flow:
sets the canvasHeight attribute of 'systemframe' to 'height'.

12.49.43.

systemCanvasHeightGet()

```
systemCanvasHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Returns:

Control Flow:
returns the value to the canvasHeight attribute of 'systemframe'.

12.49.44.

systemScrollWidthPut()

```
systemScrollWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:

sets the scrollWidth attribute of 'systemframe' to 'width'.

12.49.45.

systemScrollWidthGet()

```
systemScrollWidthGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the scrollWidth attribute of 'systemframe'.

12.49.46.

systemScrollHeightPut()

```
systemScrollHeightPut( systemframe, height)
```

Args:

systemframe a system frame abstraction

height height in pixels

Returns:

Control Flow:

sets the scrollHeight attribute of 'systemframe' to 'height'.

12.49.47.

systemScrollHeightGet()

```
systemScrollHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the scrollHeight attribute of 'systemframe'.

12.49.48.

systemLabelWidthPut()

```
systemLabelWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:

sets the labelWidth attribute of 'systemframe' to 'width'.

12.49.49.

systemLabelWidthGet()

```
systemLabelWidthGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the labelWidth attribute of 'systemframe'.

12.49.50.

systemLabelHeightPut()

```
systemLabelHeightPut( systemframe, height)
```

Args:

systemframe a system frame abstraction

height height in pixels

Returns:

Control Flow:

sets the labelHeight attribute of 'systemframe' to 'height'.

12.49.51.

systemLabelHeightGet()

```
systemLabelHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the labelHeight attribute of 'systemframe'.

12.49.52.

systemFooterWidthPut()

```
systemFooterWidthPut( systemframe, width )
```

Args:

systemframe a system frame abstraction

width width in pixels

Control Flow:

sets the footerWidth attribute of 'systemframe' to 'width'.

12.49.53.

systemFooterWidthGet()

```
systemFooterWidthGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the footerWidth attribute of 'systemframe'.

12.49.54.

systemFooterHeightPut()

```
systemFooterHeightPut( systemframe, height)
```

Args:

systemframe
a system frame abstraction

heightheight in pixels

Control Flow:

sets the footerHeight attribute of 'systemframe' to 'height'.

12.49.55.

systemFooterHeightGet()

```
systemFooterHeightGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the value to the footerHeight attribute of 'systemframe'.

12.49.56.

systemXscrollPut()

```
systemXscrollPut( systemframe, xscroll )
```

Args:

systemframe a system frame abstraction

xscroll xscrollbar widget pathname

Control Flow:

sets systemframe's Xscroll attribute to 'xscroll'.

12.49.57.

systemXscrollGet()

```
systemXscrollGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the systemframe's Xscroll attribute.

12.49.58.

systemYscrollPut()

```
systemYscrollPut( systemframe, yscroll )
```

Args:

systemframe a system frame abstraction

xscroll xscrollbar widget pathname

Control Flow:

sets systemframe's Yscroll attribute to 'yscroll'.

12.49.59.

systemYscrollGet()

```
systemYscrollGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the systemframe's Yscroll attribute.

12.49.60.

systemServerNamesPut()

```
systemServerNamesPut( systemframe, serverl )
```

Args:

systemframe a system frame abstraction

serverl list of servers

Control Flow:

sets systemframe's servers attribute to 'serverl'.

12.49.61.

systemServerNamesGet()

```
systemServerNamesGet( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:

returns the systemframe's servers attribute.

12.49.62.

systemPrint()

```
systemPrint ( systemframe )
```

Args:

systemframe a system frame abstraction

Control Flow:
prints the values of all the attributes of 'systemframe'.

12.49.63.

systemNodesCreate()

```
systemNodesCreate( systemframe, systemName )
```

Args:

systemframe a system frame abstraction

systemName name assigned to the system

Control Flow:

foreach server under systemName
do
get server cluster label, nodeslist

create the server cluster frame
if there's a previous cluster, set that cluster's next neighbor to the newly-created cluster.
keep positioning the nodes in one row of the cluster until you hit either one of the following conditions:
i. $canvas(systemNumBoxesPerRow) has been reached
ii. if adding the new cluster will result in the $canvas(systemMaxWidth) to be reached
Hitting conditions i or ii means to start a new column in the cluster frame.
Record the Xpos and Ypos position of each cluster
done
call systemServerNamesPut() for the list of servers that at least has one node.

Create the Xscrollbar
Create the Yscrollbar
Create the footer label, frame

This would shrink the display view if total width is > clusterMaxWidth
Also, it will try to adjust the display to fit all labelWidth and/or footerWidth up to clusterMaxWidth

Initialize various attribute values of system
pack xscroll if needed
pack yscroll if needed
pack the rest of systemframe's parts to create a view

12.49.64.

systemNodesReCreate()

```
systemNodesReCreate( systemframe )
```

Args:

systemframe a system abstraction

Control Flow:

redisplay the nodelabel text.
reconfigure the nodeframe's canvas' width and height and scroll regions.
update the node's status when needed.
reconfigure the xscrollbar and the yscrollbar and redisplaying them or removing them from view when needed.

Finally, resize the display width and height depending on the new sizes of maxWidth and maxHeight.

If viewType is ICON, then cover the contents of the canvas.

12.49.65.

systemRepack()

```
systemRepack ( frame )
```

Args:

frame the system frame to redisplay

Control Flow:

relabel the display label of the system

display x scroll and yscroll and recalculate appropriate widths and heights.

12.49.66.

systemDelete()

```
systemDelete( sysframe )
```

Args:

sysframe a system frame abstraction.

Control Flow:

destroys the 'sysframe' and deallocates all storage associated with 'sysframe'.

12.49.67.

systemAdjustNodesDistances()

```
systemAdjustNodesDistances ( sysframe )
```

Args:

sysframe a system frame abstraction.

Control Flow:

foreach nframe enclosed under sysframe,
do
move the nframe by offset width (as carried in the nodeframe's list of properties)
update the Xpos property of nodeframe
reset nodeframe's offsetWidth to 0
done

foreach clusterframe enclosed under sysframe,
do
move the clusterframe by offset width (as carried in the nodeframe's list of prorties)
update the Xpos property of clusterframe
reset clusterframe's offsetWidth to 0
done

12.49.68.

systemRefreshDisplay()

```
systemRefreshDisplay( sysframe )
```

Args:

sysframe a system frame abstraction.

Control Flow:

foreach nodeframe that belongs to sysframe,
do
if nodeframe's refresh flag is set to TRUE, then
do a nodeRepack
set clusterRefresh flag of the clusterframe where nodeframe belongs to to TRUE
set systemRefresh flag of the systemframe where nodeframe belongs to to TRUE
reset nodeframe's refresh flag to 0
fi

done
foreach clusterframe that belongs to sysframe,
do
if clusterframe's refresh flag is set to TRUE, then
do a clusterRepack
set systemRefresh flag of the systemframe where clusterframe belongs to to TRUE
reset clusterframe's refresh flag to 0
fi
done

if systemframe's refresh flag is set to TRUE, then

do a systemRepack
fi

12.49.69.

systemDisplayClusterStatus()

```
systemDisplayClusterStatus (sysframe, cname)
```

Args:

sysframe	a system frame abstraction.
cname	cluster name that belongs to sysframe

Control Flow:

displays in the cluster identified by cname the values for usepool, availpool, totpool, offlpool, downpool, rsvpool, unkpool.

12.49.70.

systemAddWidth()

```
systemAddWidth( systemf, incr )
```

Args:

systemf	a system frame abstraction
incr	amount to add to the clusterf' width

Control Flow:

Adds 'incr' amount to the systemf's display width and canvas width, and if the resulting values are still within the systemMaxWidth, then update the systemf's scroll region width appropriately.

12.49.71.

systemUpdateInUse()

```
systemUpdateInUse( systemf, n, jobslist )
```

Args:

systemf	a system frame abstraction
n	the name of the node
jobslist	format: { {{....} {Jobs: X USER: Y JID:} NODEJOB} }

Control Flow:

Get the Y part in 'jobslist',
if its value is empty,
set it to 0
fi

if Y valus is 1, then update node status to INUSE-EXCLUSIVE,
else if Y value > 1, then update node status to INUSE-SHARED.

12.49.72.

systemGetJobsInfo()

```
systemGetJobsInfo (sysframe, server_name)
```

Args:

sysframe a system frame abstraction

server_name name of the server to get jobs information from

Control Flow:

create a lookup table of all the nodes/execution hostnames known to 'sysframe'.

get the clusterframe of 'server_name'
issue a pbsconnect to the server
if connect was successful, then
post a pbsstatserv
if got back "resources_max.ncpus", then
save value to cpusMax
property of clusterframe
fi

post a pbsstatjob
foreach job
do
initialize user, state, hostlist, ncpus buffers

if job_state is not RUNNING then go to the next job
if got back "Job_Owner", then save value to user buffer
if got back "job_state", then save value to state buffer
if got back "exec_host", then save value to hostlist buffer
if got back "resources_used.nodes", then save list of nodes
to nodeslist buffer
if got back "Resource_List.ncpus", then save value to ncpus
buffer,
increment cpus_assn by ncpus value
set list of execution hosts assigned to the job to
nodeslist value (yes, this takes precedence) if it
exists; otherwise use the hostlist value

go through each of the execution hosts assigned to the job,
and build the jobs table:
jobs(hostname@user) <list of jobids>
done

disconnect from server_name
fi

```
foreach elem in jobs table,
do
     build the nodejobs table:
            nodejobs(hostname)
                "user1 <list of jobids>","user2 <list of jobids>",...
     build the arrays:
           njobs(hostname) <number of jobs on hostname>
           nusers(hostname) <number of users assigned with hostname>
done

foreach hostname in nodejobs table,
do

     append the node information of hostname with the info:
           nodeinfo(hostname):
              $nodejobs(hostname): "list of (user, jobids) on hostname"
              header: "nusers(hostname) USER(s) njobs(hostname) JIDs: "
              type: NODEJOB

     update the node INUSE status of hostname if its type is != MOM_SNODE
        because in this case, its associated server will update the
        status.
done

if clusterframe of server_name exists, then
     save cpus_assn value to cpusAssn property of clusterframe
fi
```

12.49.73.

systemPopulateNodesWithInfo()

```
systemPopulateNodesWithInfo( sysframe, create )
```

Args:

sysframe	a system frame abstraction
create	the create flag

Returns:

Control Flow:

```
unset node INUSE colors mapping
unset nodeinfo, nodeinfo2 saved values for each node known to the system

get a new list of nodes from each of the server

foreach server_names known to sysframe,
do
     if server has known nodes in it; then
        get the jobs information from it,
     fi
done
```

```
foreach node known to sysframe
do
      get nodejob info, nodestat, nodetype

      if nodetype is NOMOM, then
            mark node FREE if current stat is INUSE-EXCLUSIVE or
                        INUSE-SHARED but no jobs running on it.
            put 'nodeinfo2' (i.e. static attributes) as the node's info
              to be displayed
      else if nodetype is NOMOM_SNODE, then
            put 'nodeinfo2' (i.e. static attributes) as the node's info
              to be displayed
            *** no need to update its status; server's statnodes will
            *** update this
      else if nodetype is MOM, then
            mark node NOINFO if current stat is INUSE-EXCLUSIVE or
                        INUSE-SHARED but no jobs running on it.

            open a connection to the node's MOM
            if unsuccessful opening the connection, mark node as DOWN
            if successful, then
                  get status of this time-sharing node which should
                  return one of {DOWN, FREE, NOINFO}
                     if status is FREE,
                     then
                        send queries and
                        save the results in the nodeInfo property of
                        node at sysframe
                     else if status is NOINFO,
                     then
                        simply update the node's INUSE status depending
                        on whether is job is running on it or not.
                     fi
                  close fd to MOM
            fi
      else if nodetype is MOM_SNODE, then
            open a connection to the node's MOM
            if successful connection, then
                  get status of this node which should
                  return one of {DOWN, FREE, NOINFO}
                     if status is FREE,
                     then
                        send queries and
                        save the results
                     else if status is NOINFO,
                     fi
                  close fd to MOM
            fi
            put in as 'nodeinfo' property to be displayed the static
              attribute values, results of the queries sent (if any),
              and the list of jobs running on it.
      fi

      get the node information (via the nodeInfo property),
```

```
        if something exists, then
            display it on the fullnodeFrame, and mirrornodeFrame
        else
            simply remove all the items currently displayed on the node
            canvas
    done

    display the cluster status information for each of the clusterframe's known
    to sysframe
```

12.50. File: pbs.tk

This file contains routines that access some of the functionalities of PBS.

12.50.1.

getNodesList()

```
getNodesList( sitename, host, nodesq )
```

Args:

sitename	name of a particular site (actually a system name).
host	host that holds some nodes/inuse file
nodesq	the query to send that returns a list of nodes.

Control Flow:

```
    get the list of nodes from host using the query 'nodesq', and using the openrm,
    addreq, getreq, closeerm calls to obtain the result.
```

12.50.2.

TSgetStatus()

```
TSgetStatus( fd, sysframe, nodename, update )
```

Args:

fd	descriptor to MOM
sysframe	system frame that holds the frame of nodename
nodename	a node's name
update	a flag signalling whether or not to update the 'nodename's internal status flag

Control Flow:

```
    get the status of nodename that is of type time-sharing.
    Basically, an "arch" query is sent to the node's MOM.
    If sending the query was unsuccessful, then we assume
        node is DOWN.

    If the query was sent successfully, get its result.
```

i. If the result is bad (empty string returned), then
node is DOWN
ii. if the result is caused by the MOM not recognizing the query, then
node has NOINFO
iii. If neither i and ii, then mark node as FREE.
end

12.50.3.

sendTSQueries()

```
sendTSQueries (fd, sitename, nodename)
```

Args:

fd	the port to MOM
sitename	name of a site where node sending query belongs to
nodename	name of the node

Control Flow:

foreach elem in queryTable($sitename, $nodename),

foreach operand in an elem's query expression,
if it is a query string, then
save the "operand, row, col" information to queryIdxList
sends the operand as query request to port $fd
end
increment col count
done
increment row count

done
return queryIdxList

12.50.4.

recvResponses()

```
recvResponses (fd, sitename, nodename, querylist)
```

Args:

fd	port to MOM
sitename	name of the site where node running MOM belongs to
nodename	name of the node
querylist	list of queries (and additional info) to get response from

Control Flow:

create a mirror queryTable.

foreach elem querylist

```
do
    get the corresponding result from the fd
    if the result is invalid, set result to ""

    foreach (row, col) indices in the current element,
    do
        get the row-th element of the mirror queryTable. From this,
          obtain the qexpr, header, type
        replace the row-th element's col value to the result of the
              query
        recreate the mirror table with the modified element value
    done
done

return the mirror queryTable
```

12.51. File: expr.tk

This file contains routines that support query expression evaluation when data has been gathered from appropriate PBS moms.

12.51.1.

isNumber()

```
isNumber ( str, number )
```

Args:

str a string representation of a size number that can contain the characters kmgtp-KMGTP]?[bwBW]?

number the extracted number from the string.

Returns:

1 if str is a number and also returns in 'number' the value not containing the units [kmgtpKMGTP]?[bwBW]?; 0 otherwise

12.51.2.

isFloat()

```
isFloat( str )
```

Args:

str a string representation of a size number that can contain the characters, kmgtp-KMGTP]?[bwBW]?.

Returns:

1 if str is a float number (contains a .); 0 otherwise.

12.51.3.

isSingleOp()

```
isSingleOp ( str )
```

Args:

str a string representation of a TCL operator

Returns:

1 if is a single-character operator name;
2 if the single-char operator name could potentially be an incomplete name (i.e. < for <=),
0 if not a single-operator at all!
Single operators: -,~,*,/,%,+,^,(,)
incomplete opers: <,>|,&,=,,|

12.51.4.

isDoubleOp()

```
isDoubleOp ( str )
```

Args:

str a string representation of a TCL operator

Returns:

1 if 'str' represents a double character operator;
0 otherwise

Double character operators: <<, <=, >>, >=, ==, !=, &&, ||

12.51.5.

isQueryString()

```
isQueryString( str )
```

Args:

str a string representation of a TCL operator

Control Flow:

A query string is anything that is not an operator and it is not a constant number (real or int).

12.51.6.

queryExprCreate()

```
queryExprCreate( str )
```

Args:

str a query expression

Returns:

Control Flow:

Parses a str (e.g. "(loadave/ncpus * 100)") and returns a list containing each of the tokens of the expression.

expr - holds the return value for this procedure
val - holds the current character value
sval - holds set-of-chars that represents operators that are yet-to-be completed
hval - a buffer that holds items that are yet to be completed
foreach char in 'str';

if char is a space (" "), then
if there's something in the hval buffer, then append them to the expr list, clear hval
go to the next iteration
end

append char to val which holds the current set-of-chars)

if current set-of-chars is a single character operator, then
if there's something in the hval buffer, then append them to the expr list, clear hval
append this current set-of-chars char (which should only contain 1 char) into expr list, clear the val buffer which holds the current-set-of-chars
else if current set-of-chars is a single char that represents an operator that is yet to be completed, then
if there's something in the hval buffer, then append them to the expr list, clear hval
save the 1 character into the sval buffer
else if current set-of-chars represents a double-character operator,then
append these set-of-chars to expr list
clear all buffers: hval, val, sval
else (we have a set-of-chars that is not an operator), then,
if there's something in the yet-to-be-completed operator sval, then
append them to expr list
clear sval
set hval to 2nd char of val since the first character had already been saved to expr list via sval
else
append current set-of-chars to hval

```
            clear val buffer
        end
    end

    process remaining items: append hval to expr list if not empty

    return expr list
```

12.51.7.

fltround()

```
fltround (val,precision)
```

Args:

val	a floating point value
precision	number of places after the "."

Control Flow:

This rounds 'val' so that at most 'precision' number of digits would appear after ".".

12.51.8.

evaluateExpr()

```
evaluateExpr( expr )
```

Args:

expr a query expression

Returns:

Given a query expression, evaluate it, substitute query values for query strings, and return the new expression.

Control Flow:

```
if the # of items in expr is <= 1; then
    simply return the expr

Get the operands of the expression and saved it in 'operand'

while operand (a char in the expression) != "";
do
    if operand is "/" ; then  ensure that no fraction parts are lost by:
        insert "double "(" after "/"
        go through the rest of the expr, and try to match "(", and ")"
            and stop matching when all pending matches have been
            satisfied.
        insert matching ")"
        else if operand is a number ; then
```

replace the operand's value in expr to a proper value
(i.e. remove size suffixes, etc...)
if it is a floating point number, then
set hasFloat flag

get next operand (from where this loop started so yes, some of the
elements may be parsed over again)
done

Now evaluate (execute) the parsed expr and save result in val

if execution was successful,
return val ("rounded float") directly if hasFloat
otherwise, return round(ceil($val))
if execution was unssuccessful,
return an empty string

12.52. File: common.tk

This file ocntains general-purpose routines used by the **xpbsmon** utility.

12.52.1.

listcomp()

```
listcomp( list1, list2 )
```

Args:

list1 TCL list 1

list2 TCL list 2

Control Flow:

if lengths of list1 and list2 don't match, then return 1

matches each element of list1 to corresponding element at list2 and if they're the same, return 0.
Otherwise, return 1.

12.52.2.

InfoBox_flush()

```
InfoBox_flush( start_line )
```

Args:

start_line starting line of the info listbox

Control Flow:

deletes lines starting at 'start_line' for info listbox.

12.52.3.

stackPush()

```
stackPush( element )
```

Args:

element an item to add to stack

Control Flow:

This is a local implementation of the push operator of a stack ADT.

12.52.4.

stackPop()

```
stackPop()
```

Control Flow:

This is a local implementation of the pop operator of a stack ADT.

12.52.5.

isStackEmpty()

```
isStackEmpty()
```

Returns:

1 if stack is empty; 0 otherwise

12.52.6.

stackClear()

```
stackClear()
```

Control Flow:

Deletes all the items in a stack ADT.

12.52.7.

stackPrint()

```
stackPrint()
```

Returns:

Control Flow:
Prints the elements of a stack.

12.52.8.

addLlist()

```
addLlist (llist, key, row, col)
```

Args:

llist	a list of lists
key	a list's key element
row	a row value of a list element in llist
col	a key value of a list element in llist

Returns:
1 if a new {key row col} was added; 0 if an existing element was modified.

Control Flow:
NOTE: This modifies the original 'llist'.
Basically, an llist element looks like:

{key row1 col1 row2 col2 ... ...}

This adds the new elements {row col} if an element of the list matches the 'key'; otherwise, append {key row col} to llist.

12.52.9.

cleanstr()

```
cleanstr(str)
```

Args:

str	a string of characters.

Control Flow:
removes any characters in string that could be a problem under TCL like "[]" which signifies an execute action, as well as the global sysinfo(rcSiteInfoDelimeterChar) input separator.

12.53. File: color.tk

This file contains functions supporting the color bar.

12.53.1.

getNextNodeColorInUse()

`getNextNodeColorInUse()`

Control Flow:

Returns:

the next color to use to mark a single-user INUSE node canvas.

get current value of canvas(nodeColorINUSE_index), increment it by 1, and get the modulus over the # of colors in canvas(nodeColorINUSEexclusive). The effect is that after the last color on the list has been returned, then it will go back and reselect colors from the beginning of the list.

12.53.2.

assignNodeColorInUse()

`assignNodeColorInUse (job, defcolor)`

Args:

joblist	job info in the form "user.jobid"
defcolor	default color

Returns:

Control Flow:

if a color has already been assigned to job, then set retcolor to it
otherwise,

set retcolor to defcolor if one exists; otherwise, getNextColorInUse
assign this new retcolor to job

fi

update the colorCnt of the job

12.53.3.

unsetNodeColorInUseMapping()

`unsetNodeColorInUseMapping()`

Returns:

Control Flow:

unset all the color assignments to jobs, and their color counts
reset the colorINUSE_index back to -1.

12.53.4.

colorBarPopulate()

```
colorBarPopulate(startx, starty, maplist, tag)
```

Args:

startx	starting x position of the display
starty	starting y position of the display
maplist	what things to be mapped on the colorbar with elements of the form: {color1 label} {color2 label} ...
tag	tag to be assigned to canvas widgets created as a result of this call

Control Flow:

given a widget in canvas identified by coordinates (x1,y1,x2,y2)
foreach {color, label} in maplist
do
    if 1st element then
        x1 = startx
    else
        x1 = previous widget's x2 + add some space paddings

    y1 = starty
    x2 = x1 + smallTextFontWidth
    y2 = y1 + smallTextFontHeight
    create rectangle (x1,y1,x2,y2), colored with 'color' with tag 'tag'

    x1 = rectangle's x2 + space padding
    y1 = rectangle's y1

    create text(x1, y1) with tag 'tag' and text 'label'
done

12.53.5.

colorBarCreate()

```
colorBarCreate( frame_name )
```

Args:

frame_name	a frame abstraction

Control Flow:

create the color bar containing rectangles of colors and their color names.

create the color bar canvas
create the scrollbar

call colorBarPopulate to create colors info for FREE, DOWN, OFFL, RSVD, NOINFO, INUSE-TIMESHARED with tag 'fixed'.

12.53.6.

colorBarUpdate()

```
colorBarUpdate()
```

Control Flow:

create a color table containing the list of jobs for each assigned colors.

use this color table to create the inuse table for associating the color name and the display label with the latter being the <jid.user.#ofnodes>.

get the coordinates of the "fixed" items on the canvas, and call colorBarPopulate with the new items being placed just below the "fixed" items.

pack or unpack the colorbar's scrollbar as appropriate.

12.53.7.

pref()

```
pref( callerDialogBox, focusBox )
```

Args:

callerDialogBox	the dialog box abstraction that called this function.
focusBox	the dialog box to return focus to upon return from this function.

Control Flow:

set a PREFLCK
set busy cursor

create a pref dialog window

create 2 boxes within this window: box1 and box2

box1 is for specifying the site names and their view types, while box2 is for adding the server names and labels. Add values to box1 by calling siteAdd() while add values to box2 by calling serversPut(). Delete items from box1 by calling siteDelete() while for box2, call serversDelete(). For the "set nodes" button of box2, bring up the "Server Preferences" window supplied with the info about the currently selected server name. Load values to box1 by calling sitesGet()

display the row of buttons: "done redisplay view", "done don't redisplay view", "help".

display all the widgets created.

remove busy cursor

unset PREFLCK

When the dialog window is unmapped, call boxUnset().

12.53.8.

prefComplete1()

```
prefComplete1( callerDialogBox )
```

Args:

callerDialogBox the dialog box that called this procedure.

Control Flow:

destroy callerDialogBox
invoke "System.." menu button
fi

12.53.9.

prefComplete2()

```
prefComplete2( callerDialogBox )
```

Args:

callerDialogBox the dialog box that called this procedure.

Control Flow:

Same algorithm as prefComplete1() except "System.." menu button is not invoked.

12.53.10.

siteNamesGet()

```
siteNamesGet()
```

Returns:

the list of site names known to the system by consulting the global sysview array.

12.53.11.

siteNamesPrint()

```
siteNamesPrint()
```

Control Flow:
Print to stdout the names of the sites known to the system.

12.53.12.

siteAdd()

```
siteAdd( siteName, boxframe )
```

Args:

siteName name of a site

boxframe a box abstraction

Control Flow:
create a sysview entry with 'siteName' as index and boxframe's entryval 1 as the value.

12.53.13.

siteDelete()

```
siteDelete( siteName )
```

Args:

siteName name of a site

Control Flow:
unset sysview($siteName)
Delete siteName's entry on the "System.." menu.

12.53.14.

serverDelete()

```
serverDelete( serverid )
```

Args:

serverid "siteName,serverName"

Control Flow:
call unset sysnodes($serverid)

12.53.15.

queryTableGet()

```
queryTableGet( sitename, nodename, type )
```

Args:

sitename name of the site that holds the query table

nodename node name associated with a particular queryTable

type type of node represented by 'nodename'

Control Flow:

An internal table called "queryTable" is maintained to keep track of the list of queries (their display labels, and display type). This returns a query information line from the queryTable based on the "display type".

12.53.16.

queryTableDelete()

```
queryTableDelete( nodeid )
```

Args:

nodeid some node identifier (it could be a nodename, or a supernodename,node)

Control Flow:

unset queryTableDelete(nodeid)

12.53.17.

queryTableSave()

```
queryTableSave( sitename, boxframe )
```

Args:

sitename
name of a site

boxframe a box abstraction

Control Flow:

set host to the box's title

unset queryTable(sitename,host)

go through each row of input of 'boxframe', and use them as input to queryTable(sitename,host)

12.53.18.

queryTableLoad()

```
queryTableLoad( sitename, boxframe )
```

Args:

sitename name of a site

boxframe a box abstraction

Control Flow:

set host to the title of 'boxframe'.

if queryTable(sitename,host) does not exist, then return

go through each entry of queryTable(sitename,host), and use each one as input to the 1st, 2nd, and 3rd columns of the box in 'boxframe'.

12.53.19.

queryTablePrint()

```
queryTablePrint(sitename)
```

Args:

sitename name of a site

Control Flow:

prints the queryTable information for 'sitename'.

12.53.20.

sitesGet()

```
sitesGet( boxframe )
```

Args:

boxframe a box abstraction

Control Flow:

get the entry in boxframe. Delete its content.

get all the site names and corresponding view types known to the system, and load them as entries to box.

12.53.21.

sitesPut()

```
sitesPut( boxframe )
```

Args:

boxframe a box abstraction

Control Flow:
Go through each element of box in 'boxframe' and add a corresponding radiobutton to the "System.." menu.

12.53.22.

serverNamesGet()

```
serverNamesGet( siteName )
```

Args:
siteName name of a site

Returns:
list of server names that are under 'siteName'.

Control Flow:
This function makes use of the global sysnodes array.

12.53.23.

statNodes()

```
statNodes( server_name, sysframe )
```

Args:
server_name name of a server
sysframe associated system frame

Returns:
list of nodes managed by 'server_name'.

Control Flow:
do a pbsconnect to 'server_name'
post a 'pbsstatnode' query. Get the results.
From the results,
(1) add node's name and NOMOM_SNODE type to the nodes list. Update the node's type internally.
(2) do the following actions:
state => nodeUpdateStat()
properties => systemNodeInfo2Append()

issue a pbsdisconnect
return the nodes list

12.53.24.

statNodesStateMap()

```
statNodesStateMap( state )
```

Args:

state state info to map

Returns:

the state info recognizeable to xpbsmon

Control Flow:

```
IF    state    => RETURN
      FREE, free   => FREE
      OFFLINE, offline  => OFFLINE
      DOWN, down => DOWN
      RESERVED, reserve  => RESERVED
      INUSE-EXCLUSIVE, job-exclusive => INUSE-EXCLUSIVE
      INUSE-SHARED, job-sharing => INUSE-SHARED
      NOINFO => NOINFO
      <default> => ""
```

12.53.25.

nodesListMerge()

```
nodesListMerge( nlist1, nlist2, frame )
```

Args:

nlist1 nodes list 1

nlist2 nodes list 2

frame system node frame to update information on

Control Flow:

```
Load nlist1.
Check nlist2:
   if it encounters a nodename that is in 'nlist1',
      update the corresponding 'nlist1' entry with values from
      'nlist2' and set the node type to 'MOM_SNODE'.
   else
      add to nlist1.
   update the node's type

return the new list of nodes (with information) in the order that they
    were specified in the argument list.
```

12.53.26.

serverNamesSorted()

```
serverNamesSorted( systemName, servers, nodesp, frame )
```

Args:

systemName	name of a site
servers	list of server names
nodesp	an array to hold the nodes for each of the server listed in 'servers'
frame	frame of 'systemName'

Returns:

list of server names that are under 'systemName' sorted according to the increasing number of nodes each one holds.

Control Flow:

for each of the server name in 'servers',
get the nodes list by statnodes-ing the server. Save the results in nlist1.
get the nodes list specified by user in the Pref dialog box. Save these
results in nlist2.
Merge 'nlist2' with 'nlist1' and place result in 'nodesp' array.
create a new servers list called 'newservers', and add items to it depending
on increasing # of nodes.
done
return 'newservers'

12.53.27.

serversPut()

```
serversPut( boxframe, siteName )
```

Args:

boxframe	a box abstraction
sitename	name of a site

Control Flow:

if 'siteName' is an empty string, then return

get the box from 'boxframe'
go through each element of box (server_name, server_label)), and
if global array entry sysnodes($siteName,$server_name) is set,
then update its server label value to 'server_label'
else
initialize: sysnodes($siteName,$server_name) -> $server_label.

12.53.28.

serversGet()

```
serversGet( boxframe, siteName )
```

Args:

boxframe a box abstraction

siteName name of a site

Control Flow:

get the server, label boxes from 'boxframe'. Delete all their entries. Update the nrows count of Servers box to 0.

get the entry widgets associated with the Servers box and delete all the characters that it is holding.

Set the site box's titlelabel to "Servers@siteName"

go through each of the server names to 'siteName', inserting the server_name into server box, and the server_label into the label box, and updating its nrows count.

select the first entry of the Servers box.

12.53.29.

sysnodesGet()

```
sysnodesGet( sitename, boxframe )
```

Args:

sitename name of a site.

boxframe a box abstraction

Returns:

Control Flow:

set host to the title of the boxframe

if sysnodes(siteName,host) does not exist, then
return
fi

get the node box (box that holds nodenames) and the nodetype box.

delete all the entries of node box.

for the other entries of sysnodes(siteName,host), input them to node box and type box, updating the box's nrows counter.

select the first entry of box if one exists.

12.53.30.

sysnodesPut()

```
sysnodesPut( sitename, serverName, entry, box )
```

Args:

sitename	name of a site
serverName	name of a server
entry	name of an entry widget
box	a box abstraction

Returns:

Control Flow:

Reset the sysnodes(sitename,serverName) value to contain only the server_label.

And for the rest of the entries of sysnodes(sitename,serverName), fill them with inputs from the 1st and 2nd column of 'box'.

12.53.31.

sysnodesPrint()

```
sysnodesPrint(sitename)
```

Args:

sitename	name of a site

Control Flow:

prints all the nodes information contained under 'sitename'.

12.53.32.

prefServerComplete()

```
prefServerComplete( boxframe, callerDialogBox )
```

Args:

boxframe	a box abstraction
callerDialogBox	the dialog box that called this procedure

Returns:

Control Flow:

if 'boxframe' has 0 rows, then issue an error message.

Otherwise, simply destroy callerDialogBox

12.53.33.

prefServer()

```
prefServer(siteName, serverName, callerDialogBox, focusBox)
```

Args:

siteName	name of a site
serverName	name of a server
callerDialogBox	the dialog box that called this procedure
focusBox	the dialog box to return focus to upon return from this function

Returns:

Control Flow:

enable busy_cursor
create top part and bottom part of the dialog box.
create the server display label dialog box.

create the server box to be filled with Nodes information.
for the update button of server box, have it so that it calls "prefQuery()".

build the row of command buttons {ok help}. For the ok button, create an action that will update the sysnodes global variable when the button is pressed (via call to sysnodesPut)

display the widgets.
load default values for the server box.

remove busy cursor

Upon return, put a grab on callerDialogBox, and set focus to focusBox.

12.53.34.

prefQuery()

```
prefQuery (siteName, nodeName, nodeType, callerDialogBox, focusBox)
```

Args:

siteName	name of a site
nodeName	name of a node
nodeType	type of node
callerDialogBox	the dialog box that called this procedure
focusBox	the dialog box to return focus to upon return from this function

Returns:

Control Flow:

Return if nodeType is not MOM.

enable busy_cursor
create top part and bottom part of the dialog box.

create the Query box to be filled with query expressions, display label, and output type information. Load it with default values by calling queryTableLoad().

build the row of command buttons {ok help}. For the ok button, create an action that will update the queryTable global variable when the button is pressed (via call to queryTableSave)

display the widgets.

remove busy cursor

Upon return, put a grab on callerDialogBox, and set focus to focusBox.

12.54. File: preferences.tcl

This file contains procedures for loading, saving parameters into the xpbsmonrc file.

12.54.1.

prefLoadSitesInfo()

`prefLoadSitesInfo()`

Control Flow:

foreach info in sitesinfo global list
do
from the info, get the:
col1: sitename
col2: site's VIEW type
col3: servername
col4: serverlabel
col5: nodename
col6: nodetype
col7: querylist

if no 'nodename' has been specified, then simply update the site's view type, and server information and continue to the next iteration of this loop.
return immediately if one of the ff:

1. # of cols in info is != 7
2. site's VIEW type is not ICON or FULL
3. nodetype is not MOM or NOMOM
4. querytype is not SCALE, TEXT

based on site's VIEW type, update sysview for site
update sysnodes for sitename,servername making sure no duplicates in the "nodename nodetype" value.

update queryTable for sitename,nodename using querylist and ensuring no duplicates.
done

12.54.2.

prefSaveSitesInfo()

```
prefSaveSitesInfo()
```

Control Flow:
using the global variables sysview, sysnodes, queryTable, recreate the global siteinfo list with the following format:

<sitename>sep<sitevtype>sep<svrname>sep<srvlabel>sep<nodename>sep<nodetype>sep<querylist>

where sep is sysinfo(rcSiteInfoDelimeterChar)

12.55. File: main.tk

This file contains procedures for building the main application window.

12.55.1.

iconView()

```
iconView( force )
```

Args:

Control Flow:
force - set to TRUE if to force an icon View of the system

if !force and the view of the current system is already in ICON, then
return

if system already exists, then
recreate system nodes in ICON view
else
create system nodes from scratch in ICON view
schedule a new cycle of populateNodesWithInfo
fi

12.55.2.

fullView()

```
fullView( force )
```

Args:

Control Flow:
force - set to TRUE if to force an icon View of the system

if !force and the view of the current system is already in FULL, then
return

if system already exists, then
recreate system nodes in FULL view
else
create system nodes from scratch in FULL view
schedule a new cycle of populateNodesWithInfo
fi

12.55.3.

build_main_window()

```
build_main_window(mainWindow)
```

Args:

mainWindowa frame abstraction

Control Flow:

create 3 frames for holding the menubar, main frame, and status bar.
display them on screen.

fill menubar with info
fill main frame with info
fill statusbar with info.

12.55.4.

fillMainFrame()

```
fillMainFrame(widget_name)
```

Args:

widget_name a widget abstraction

Control Flow:

displayView for current systemName (siteName)

12.55.5.

fillStatusbarFrame()

```
fillStatusbarFrame(widget_name)
```

Args:

widget_name a widget abstraction

Control Flow:
create the "INFO" labe
create the info bar listbox

display the widgets.

12.55.6.

displayView()

```
displayView(frame, sitename, init)
```

Args:

frame a frame abstraction

sitename name of a site

init flag whether it is the first time

Control Flow:
delete the previous frame in view

display icon view if current system's view is ICON
display full view if current system's view is FULL

12.55.7.

fillMenubarFrame()

```
fillMenubarFrame(widget_name)
```

Args:

widget_name a widget abstraction

Control Flow:
create a menubutton called "System.."

foreach sitename known to the system,
do
add a corresponding radio button to the menubutton above
done

build the row of command buttons:
Pref.. AutoUpdate.. Help About.. Close <minimize> <maximize>

adjust various sizes of various button and set bindings:

Pref ..... call pref()
AutoUpdate ..... call auto_upd()
Help ..... call xpbs_help()
About ..... call about()
<minimize> ..... call iconView()

<maximize> ..... call fullView()
Close ..... call prefSaveSitesInfo(), prefsave()

12.56. File: listbox.tk

This file contains routines supporting a complete listbox widget.

12.56.1.

lboxvalue_isUnique()

```
lboxvalue_isUnique(listbox, value)
```

Args:

listbox the list box

value a value string

Returns:

1 if 'value' is unique (that is, not found in listbox); 0 if 'value' is not unique (is found in listbox)

Control Flow:

go through each element of listbox,
if element is the same as 'value' then return 0 (meaning not unique)

return 1 (meaning value is unique)

12.56.2.

lcomp()

```
lcomp(lbox1, lbox2)
```

Args:

lbox1 listbox 1

lbox2 listbox 2

Returns:

0 if lbox1 and lbox2 contain the same elements; 1 if not

Control Flow:

return 1 if the 2 listboxes don't match in sizes

go through each element of the 2 listboxes, and if at least 2 elements on
the same position don't match, then return 1.
Otherwise, return 0.

NH 2 File: box.tk

This file contains functions that support the building of a box widget which is a multi-column listbox and items are added to it via accompanying entry widgets.

12.56.3.

box()

```
box( frame_name, args )
```

Args:

frame_name frame to place box widget on.

args list of parameters for the box:

-title <title_of_the_box>

title returns title of the box

-class <class_name>

class returns the class of the box

titlelabel returns the label widget of the box.

-key <keyval)key value of the box (index to field containing unique values).

key returns the key of the box

-entrylabels <listOfEntryLabels>

entrylabels returns list of entry labels

-lboxlabels <listOfListBoxLabels>

lboxlabels returns list of listbox labels.

-lboxwidths <listOfWidths>

lboxwidths returns list of listbox widths

-lboxheights <listOfHeights> lboxheights
returns the list of listbox heights

-orient <orientation of box: x, y>

orient returns the orientation of the box

-grabBox <widget_name>

grabBox returns the widget name that is grabbed after the box widget has come up.

-selindex <index_whose_value_to_be_highlighted>

selindex returns the index whose value is currently highlighted.

-vscroll <scrollbar_pathname>

vscroll returns the vertical scrollbar pathname

ncols returns the # of cols of the box

-nrows <# of rows of the box>

nrows returns the number of rows of the box.

lbox <index>returns the listbox widget pathname at col <index> of box.

entry <index>
returns the entry widget pathname at col <index> of box.

-entryval <index> <value>

entryval <index>
returns the value of the entry widget at <index>

-entryvalDeleted <value>

entryvalDeleted
returns the last entry value that was deleted from the box.

-noUpdateButton <true|false>

noUpdateButton
returns the flag value of whether or not the box should have an update button.

-addCmd <cmd_func>

addCmd returns name of command to execute after the add button of the box has been clicked.

-remCmd <cmd_func>

remCmd returns the name of command to execute after the delete button of the box has been clicked.

updateButtonreturns the updateButton of the box.

create if the box is to be newly created.

unset unset all storage associated with box.

getBoxArray return the storage array of the box.

Control Flow:

A box is made of a list of listboxes, where a row on each listbox is collectively defined together, and input to each row is provided by a list of entryboxes with one entry for each listbox.

create one frame to hold everything. add the frame_name to list of sysinfo boxes.

create the top frame
create the bottom frame
create the bottom frame's entry frame
create the bottom frame's listbox labels frame
create the bottom frame's box frame

puts title label on the top frame
create the add button on the bottom frame's entry frame

get the parameters for list of entry info, listbox widths, and listbox heights.

```
set k 0
foreach entryinfo in list of entry infos,
do
        if entryinfo's type is MENU_ENTRY, then create a menu-ed
            entry
        otherwise,
            create an full entrybox complete label specified in
        entryinfo
        increment k
done
```

set the number of columns counter to k
pack all the entry widgets or menu-ed entry widgets created.

set the tabbing sequence for the box as it traverses the entry widgets.

On the bottom frame's box frame,
create the list of listboxes based on what's specified for the
the lboxlabels.
pack all the listbox widgets.

set the scrolling mechanism for the box.

create the accompanying listboxes' command buttons:
<delete> if noUpdateButton; otherwise,
<delete> <update>

for the <delete> button, bind the remCmd to it.

depending on orientation, display the box horizontally if orient is 'x',
otherwise, vertically, if orient is 'y'.

12.56.4.

boxesUnset()

`boxesUnset()`

Control Flow:
go through each of the boxes known to system and unset them.

12.56.5.

boxAdd()

`boxAdd( frame_name, addfunc )`

Args:
frame_namea box abstraction
addfunc the TCL expression to execute after adding entries to box.
Returns:
Control Flow:
get the keylist (list of indices that collectively defines a unique entry
of the box), it's of the format:
{ keyindex1 (<keyindex1>:<keyval1>) ... (<keyindexN>:<keyvalN>) }

go through each column of box,
do
keep track of column index
cleanup the entry values of the box, removing any ";", "]", and "["
which can be problematic under TCL

if keylist contains a "(^|)<keyindex>", then this must be a
special key that must have a non-empty value, and that
its value must be unique if it matches <keyval>. Record this
info.
fi
create a keyval_string

done

go through each row of box,
do
go through each col of box,
do
set rowvalue_string to the values
of the row whose col index match the keylist
issue an error message if a col is one of the keys and its
value is one of the values that must be unique,
and the value is not unique.
done

issue another error message if rowval_string matches keyval_string
(a duplicate)

done

Finally, insert keyval_string to the appropriate columns of the box.
increment box's row count
select the added item

execute addfunc

12.56.6.

boxDelete()

`boxDelete(frame_name, remfunc)`

Args:

frame_name a box abstraction

remfunc name of a TCL function to execute when a box is deleted.

Control Flow:

go through each col of box,
do
get the listbox at that column
get the selected index at that column
save the value of the selected index
delete the selected element
select the next item on the listbox
done

decrement the row count

execute the remfunc

12.56.7.

boxGetCurSelect()

```
boxGetCurSelect( frame_name, index )
```

Args:

frame_name a box abstraction

index column index of box

Returns:

the selected item at column 'index' of box.

12.56.8.

boxSelect()

```
boxSelect( frame_name, index )
```

Args:

frame_name a box abstraction

index column index of box

Control Flow:

if index is -1 (nothing to select), simply execute the 1st listbox's "cmd"

go through each column of of box,
do
get the listbox at column.
select the 'index' of listbox.

whatever value is selected, load it as input to the accompanying input entry widgets.
if this is the 1st column, set the focus to the corresponding entry widget, and select this entry's value.

if the listbox has an accompanying "cmd", execute it.
done

12.56.9.

boxSetScroll()

```
boxSetScroll( frame_name )
```

Args:

frame_name a box abstraction

Control Flow:

if no vscroll of box, then return

go through each column of box,
do
get the listbox at column, and set its yscrollcommand to set vscroll of box
done

configure vscroll to adjust all views of the listboxes of box at the same row.

12.56.10.

boxSetTabbing()

```
boxSetTabbing( f )
```

Args:

f a box abstraction

Control Flow:

go through each entry of box, and set tabbing mechanism so that when the <tab> key is hit, it will go to the next (forward) adjacent entry box, and when <cntrl-p> is hit, then it go to the previous adjacent entry box.

12.56.11.

boxAdjScrollView()

```
boxAdjScrollView( args )
```

Args:

args input parameter: <list of listboxes> <scroll parameters>

Returns:

Control Flow:

go through each of the <list of listboxes>,
and issue "yview <scroll parameters" to each one.

12.57. File: bindings.tk

This contains procedures that attaches bindings to widgets as a result of a mouse click.

12.57.1.

bind_button1()

```
bind_button1( win )
```

Args:

win pathname of a widget

Control Flow:

when the mouse button 1 is clicked on 'win', record the x and y coordinates.

12.57.2.

bind_canvas()

```
bind_canvas( canvasw )
```

Args:

canvasw pathname to a canvas widget

Control Flow:

when button 1 is clicked on 'canvasw', then save X and Y coordinates of pointer, and popup a node info box.

12.58. File: entry.tk

This file contains routines that are realated to an entry widget.

12.58.1.

menuEntry()

```
menuEntry( frame_name, args )
```

Args:

frame_name a frame where the menu entry widget is to be placed.

args parameters to this new widget which can be:

menubutton
: returns the button widget inside the entry box.

-menuvalues <listOfpossibleValues>

menuvalues
: returns list of possible values to menu entry.

-title <titlelabel>

title returns the title assigned to the menu entry.

-textvariable <NameOftextVariableToHoldResult>

createcreate a new menu entry widget

getMenuArray
: returns the storage used for the menu entry.

Returns:

Control Flow:

a menuEntry is an entry widget which when clicked, will give you the list of possible values, and by using the mouse, you can select the value for the entry.

create the label containing the <titlelabel>

create the menubutton with <textvariable> used to hold results. Fill this button with popup sub-menus using as <menuvalues> as labels. Invoke the 1st entry on the list.

display the widgets.

12.59. File: auto_upd.tk

This file contains routines related to the Auto Update dialog.

12.59.1.

data_auto_update()

```
data_auto_update()
```

Returns:

Control Flow:

same function as in xpbs except the function systemPopulateNodesWithInfo() is called instead.

12.60. File: dialog.tk

This file contains routines that are related to the building of dialog boxes.

12.60.1.

popupNodeInfoBox()

```
popupNodeInfoBox(callerDialogBox, nodeframe, nodename, nodeType, clusterSrc,
                 focusBox)
```

Args:

callerDialogBox - the dialog box when this procedure was called

nodeframe	a node abstraction
nodename	name associated with 'nodeframe'
nodeType	type associated with 'nodeframe'
clusterSrc	frame where 'nodeframe' resides
focusBox	where to set focus upon return from this call

Control Flow:

if nodeInfoBox already exists, then retun immediately

popup a dialog box.

if nodename is of compound form (supernode,nodename), then use only the
nodename portion as label.

create a new nodeframe with 'nodename', 'nodeType', 'clusterSrc', and MIRROR
type.

get the node information for 'nodename' under systemframe that is under
nodeframe. If one exists, then display it on the
MIRROR nodeframe. Otherwise, just color the nodeframe.

create the command button: <ok> to be placed at the bottom of the dialog
box. Register default action to this button.

Upon returning from this dialog window,
be sure to set focus back to focusBox,
to grab callerDialogBox,
do nodeDelete

[This page is blank.]

Index of File Names and Functions