GNUBatch Release 1

API Reference Manual

Table of Contents

Chapter 1

Introduction to GNUBatch API

The **GNUBatch** API enables a C or C++ programmer to access **GNUBatch** facilities directly from within an application. The application may be on a Unix host or on a Windows workstation.

Communication takes place using a TCP connection between the API running on a Windows or Unix machine and the server process xbnetserv running on the Unix host in question. The same application may safely make several simultaneous conversations with the same or different host.

The user may submit, change, delete and alter the state of jobs or variables to which he or she has access, and may receive notification about changes which may require attention. In addition, the user access control parameters may be viewed and if permitted, changed.

Chapter 2

Installation and access to API

The API is provided as two files, a *header* file `gbatch.h` and a library file.

The header file should be copied to a suitable location for ready access. On Unix systems we suggest that the header file is copied to the directory `/usr/local/include` so that it may be included in C programs via the directive `#include <gbatch.h>`

The library file is set up so it can be invoked with the ld directive `-lgnubatch`. On some UNIX systems you may have to include a socket handling library as well when linking.

On Windows systems the library is supplied as a DLL. Again we suggest that it be placed in the default search path.

Chapter 3

Warning on Windows Version!

The default stack segment size allocated by some compilation systems, such as Microsoft Visual C++, is too small to accommodate the stack space required for some of these functions together with that for Windows and the Network software.

The manifestation of problems due to this can be very strange and seemingly unrelated.

Chapter 4

The API file descriptor

Each routine in the API uses a *file descriptor* to identify the instance in progress. This is an integer value, and is returned by a successful call to one of the open or login routines such as gbatch_open routine or gbatch_login. All other routines take this value as a first parameter. As mentioned before, more than one session may be in progress at once with different parameters.

Each session with the API should be commenced with a call to one of the open or login routines and terminated with a call to gbatch_close.

Note that each API session will cause a separate instance of xbnetserv to be spawned on the server to service it.

Chapter 5

Slot numbers

Each job or variable is identified to **GNUBatch** by means of two numbers:

1. The host or network identifier. This is a long corresponding to the Interment address in network byte order. The host identifier is given the type `netid_t`.
2. The shared memory offset, or *slot number*. This is the offset in shared memory on the relevant host of the job or variable and stays constant during the lifetime of the job or variable. The type for this is `slotno_t`.

These two quantities uniquely identify any job or variable.

It might be worth noting that there are two slot numbers relating to a remote job or variable.

1. The slot number of the record of the job or variable held in local shared memory. This is the slot number which will in all cases be manipulated directly by the API.
2. The slot number of the job or variable on the owning host. This is in fact available in the job structures as the field `bj_slotno` and in the variable structure as the field `var_id.slotno`. For local jobs or variables, these fields usually have the same value, but this should not be relied upon.

Chapter 6

Sequence numbers

These quantities are not available directly, but are held to determine how out-of-date the user's record of jobs or variables may be.

Every time you read a job or variable record, the sequence number of the job or variable list is checked, and if out-of-date, you will receive the error `GBATCH_SEQUENCE`. This is not so much of an error as a warning. If you re-read the job or variable required, then you will not receive this error, except where you and one or more other users have "raced" to update a variable and you have "lost the race".

If you want to bypass this, you can access the job or variable without worrying about the sequence using the *flag* `GBATCH_IGNORESEQ`, however you might receive an error about unknown job or variable if the job or variable has disappeared. In the case of variables you may still receive the `GBATCH_SEQUENCE` error if another user "wins a race" as described above.

Chapter 7

The Job Structure

The following structures are used to describe jobs within the API. All the structures and definitions are contained within the include file `gbatch.h`.

7.1 Overall Structure

A job structure consists of two parts, a *header part* and a *string table*. The header part contains all the run flags and parameters such as load level and priority, whilst the string table contains all the variable-length fields, namely the job title, directory, environment variables, arguments and redirections.

Whilst the C programmer may directly manipulate the string table if he or she wishes, this is strongly discouraged in favour of the use of the utility functions gbatch_gettitle, gbatch_puttitle etc. Future extensions to **GNUBatch** and the API will attempt wherever possible to preserve the interfaces to these functions.

```
typedef   struct    {
        apiBtjobh  h;
        char       bj_space[JOBSPACE];
}  apiBtjob;
```

The size of the `bj_space` vector is given by the constant `JOBSPACE` which is determined when **GNUBatch** is compiled. It may possibly vary from machine to machine, but the string manipulation functions pack the data at the start of the space and always pass the minimum length, so enabling copies of **GNUBatch** with different values of `JOBSPACE` to be able to talk to one another.

When creating new jobs, we suggest that you start by clearing the entire structure to zero and then insert the various fields. This way your code should work across various releases as we shall endeavour to keep the existing behaviour where the new fields are zero.

7.2 The job header

The header structure for the job is defined as follows:

```
typedef   struct    {
        jobno_t         bj_job;
        long            bj_time;
        long            bj_stime;
        long            bj_etime;
        int_pid_t       bj_pid;
        netid_t         bj_orighostid;
        netid_t         bj_hostid;
        netid_t         bj_runhostid;
        slotno_t        bj_slotno;
        unsigned char   bj_progress;
        unsigned char   bj_pri;
        unsigned short  bj_ll;
        unsigned short  bj_umask;
        unsigned short  bj_nredirs,
                        bj_nargs,
                        bj_nenv;
        unsigned char   bj_jflags;
        unsigned char   bj_jrunflags;
        short           bj_title;
        short           bj_direct;
        unsigned long   bj_runtime;
        unsigned short  bj_autoksig;
        unsigned short  bj_runon;
        unsigned short  bj_deltime;
        char            bj_cmdinterp[CI_MAXNAME+1];
        Btmode          bj_mode;
        apiJcond        bj_conds[MAXCVARS];
        apiJass         bj_asses[MAXSEVARS];
        Timecon         bj_times;
        long            bj_ulimit;
        short           bj_redirs;
        short           bj_env;
        short           bj_arg;
        unsigned short  bj_lastexit;
        Exits           bj_exits;
}  apiBtjobh;
```

The various constants `MAXCVARS`, `MAXSEVARS` etc are defined elsewhere in `gbatch.h`, and the sub-structures for times, modes, conditions, assignments and exit codes are described below.

The functions of the various fields are as follows:

`bj_job`	Job number
`bj_time`	Time job was submitted
`bj_stime`	Time job was (last) started
`bj_etime`	Time job last finished
`bj_pid`	Process id of running job
`bj_orighostid`	Originating host id, network byte order.
`bj_hostid`	Host id of job owner
`bj_runhostid`	Host id running job, if applicable
`bj_slotno`	Slot number on owning machine of non-local job
`bj_progress`	Progress code, see below
`bj_pri`	Priority
`bj_ll`	Load level
`bj_umask`	Umask value
`bj_nredirs`	Number of redirections
`bj_nargs`	Number of arguments
`bj_nenv`	Number of environment variables
`bj_jflags`	Job flags see below
`bj_jrunflags`	Job flags whilst running
`bj_title`	Offset of title field in job string area
`bj_direct`	Offset of directory field in job string area
`bj_runtime`	Maximum run time (seconds)
`bj_autoksig`	Signal number to kill with after run time
`bj_runon`	Grace time (seconds)
`bj_deltime`	Delete time automatically (hours)
`bj_cmdinterp`	Command interpreter name (NB string in R5 up)
`bj_mode`	Job permissions, see below
`bj_conds`	Job conditions, see below
`bj_asses`	Job assignments, see below
`bj_times`	Job time constraints, see below
`bj_ulimit`	Job maximum file size
`bj_redirs`	Offset of redirection table in job string area
`bj_env`	Offset of environment variables in job string area
`bj_arg`	Offset of arguments in job string area.
`bj_lastexit`	Saved exit code from last time job was run
`bj_exits`	Exit code constraints, see below

If the user only has "reveal" access when a job is read using gbatch_jobread, then all fields will be zeroed apart from `bj_job`, `bj_jflags`, `bj_progress`, `bj_hostid`, `bj_orighostid` and `bj_runhostid`. The completion of the `bj_mode` field depends upon whether the user has "display mode" access.

7.2.1 Progress codes

The progress code field of a job consists of one of the following values.

`BJP_NONE`	Job is ready to run
`BJP_DONE`	Job has completed
`BJP_ERROR`	Job terminated with error
`BJP_ABORTED`	Job aborted
`BJP_CANCELLED`	Job cancelled
`BJP_STARTUP1`	Startup - phase 1
`BJP_STARTUP2`	Startup - phase 2
`BJP_RUNNING`	Job is running
`BJP_FINISHED`	Job has finished - phase 1

The values `BJP_STARTUP1` and `BJP_STARTUP2`, and `BJP_FINISHED` are transient states.

Note that jobs should be created and updated with the values `BJP_NONE` (this is zero, so if the job structure is cleared initially it will be set to this) and `BJP_CANCELLED` only.

7.2.2 Job Flags

The field `bj_jflags` consists of some or all of the following values.

`BJ_WRT`	Send message to users terminal on completion
`BJ_MAIL`	Mail message to user on completion
`BJ_NOADVIFERR`	Do not advance time on error
`BJ_EXPORT`	Job is visible from outside world
`BJ_REMRUNNABLE`	Job is runnable from outside world
`BJ_CLIENTHOST`	Job was submitted from Windows client
`BJ_ROAMUSER`	Job was submitted from "dynamic IP" client

The flags `BJ_CLIENTHOST` and `BJ_ROAMUSER` are set as appropriate by the interface and will be ignored if a job is created or updated with these set.

7.2.3 Run Flags

The field `bj_jrunflags` in the job header contains some or all of the following bits:

`BJ_PROPOSED`	Remote job proposed. This is an intermediate step in a remote execution protocol.
`BJ_SKELHOLD`	Job held dependent on inaccessible remote variables
`BJ_AUTOKILLED`	Job has exceeded run time, initial signal applied.
`BJ_AUTOMURDER`	Job has exceeded "grace period", final kill applied.
`BJ_HOSTDIED`	Job killed because owning host died.
`BJ_FORCE`	Force job to run
`BJ_FORCENA`	Do not advance time on Force job to run

These are provided for reference only when a job is read and will be ignored if a job is created or updated with any of these set.

7.2.4 Mode Structures

These are applicable to both jobs and variables, and contain the permission structures in each case. Note that user profiles are held in a separate structure defined later.

```
typedef   struct    {
        int_ugid_t   o_uid, o_gid, c_uid, c_gid;
        char         o_user[UIDSIZE+1],
                     o_group[UIDSIZE+1],
                     c_user[UIDSIZE+1],
                     c_group[UIDSIZE+1];
        unsigned  short  u_flags,
                         g_flags,
                         o_flags;
}  Btmode;
```

The two sets of users and groups correspond to those of the current owner, and the creator. When ownership is changed, which is a two stage process in **GNUBatch**, the creator field is changed first when the owner is "given away" and then the owner field when the owner is "assumed".

The numeric user ids are unlikely to be very useful unless they are identical on the host machine to the calling machine (possibly if it is the same machine), but are included for completeness.

The flags fields consist of the following bitmaps.

BTM_READ	Item may be read
BTM_WRITE	Item may be written
BTM_SHOW	Item is visible at all
BTM_RDMODE	Mode may be displayed
BTM_WRMODE	Mode may be updated
BTM_UTAKE	User may be assumed
BTM_GTAKE	Group may be assumed
BTM_UGIVE	User may be given away
BTM_GGIVE	Group may be given away
BTM_DELETE	Item may be deleted
BTM_KILL	Job may be killed, not meaningful for variables.

The #define constants JALLMODES and VALLMODES combine all valid flags at once for jobs and variables respectively for where the user wants to "allow everything".

If a job or variable is read, and the BTM_RDMODE permission is not available to the user, then the whole of the mode field is set to zero apart from o_user and o_group. Jobs and variables may not be created without certain minimal modes enabling someone to delete them or change the modes.

7.2.5 Condition Structures

The job condition structures consist of the following fields:

```
typedef   struct    {
        unsigned char   bjc_compar;
        unsigned char   bjc_iscrit;
        apiVid          bjc_var;
        Btcon           bjc_value;
}  apiJcond;
```

The field bjc_compar has one of the following values:

C_UNUSED	Not used. This marks the end of a list of conditions if there are less than MAXCVARS. This is zero.
C_EQ	Compare equal to value
C_NE	Compare not equal to value
C_LT	Compare less than value
C_LE	Compare less than or equal to value
C_GT	Compare greater than value
C_GE	Compare greater than or equal to value

The field bjc_iscrit is set with some or all of the following bit flags:

`CCRIT_NORUN`	Set to indicate job should not run if remote variable in this condition unavailable.
`CCRIT_NONAVAIL`	Set internally to denote that condition is relying on unavailable variable.
`CCRIT_NOPERM`	Set internally to denote that condition is relying on remote variable which proves to be unreadable when machine has restarted.

The field `bjc_var` is an instance of the following structure:

```
typedef   struct  {
        slotno_t      slotno;
}  apiVid;
```

The slot number referred to is that on the host machine which the API is talking to, as returned by gbatch_varlist, and not the slot number on the owning machine.

The field `bjc_value` is an instance of the following structure.

```
typedef   struct    {
        short    const_type;
        union   {
                char   con_string[BTC_VALUE+1];
                long   con_long;
        }  con_un;
}  Btcon;
```

The field `const_type` may be either `CON_LONG` to denote a numeric (long) value, or `CON_STRING` to denote a string value.

7.2.6 Assignment structures

A job assignment structure consists of the following fields:

```
typedef   struct    {
        unsigned short bja_flags;
        unsigned char  bja_op;
        unsigned char  bja_iscrit;
        apiVid         bja_var;
        Btcon          bja_con;
}  apiJass;
```

The field `bja_flags` consists of one or more of the following bits

`BJA_START`	Apply at start of job
`BJA_OK`	Apply on normal exit
`BJA_ERROR`	Apply on error exit
`BJA_ABORT`	Apply on abort
`BJA_CANCEL`	Apply on cancel
`BJA_REVERSE`	Reverse assignment on exit

The field `bja_op` consists of one of the following values.

`BJA_NONE`	No operation. This is used to signify the end of a list of assignments if there are less than `MAXSEVARS`. This is zero.
`BJA_ASSIGN`	Assign value given
`BJA_INCR`	Increment by value given
`BJA_DECR`	Decrement by value given
`BJA_MULT`	Multiply by value given
`BJA_DIV`	Divide by value given
`BJA_MOD`	Modulus by value given
`BJA_SEXIT`	Assign job exit code
`BJA_SSIG`	Assign job signal number

The field `bja_iscrit` is set with some or all of the following bit flags:

`ACRIT_NORUN`	Set to indicate job should not run if remote variable in this assignment unavailable.
`ACRIT_NONAVAIL`	Set internally to denote that assignment is relying on unavailable variable.
`ACRIT_NOPERM`	Set internally to denote that assignment is relying on remote variable which proves to be unwritable when machine has restarted.

The fields `bja_var` and `bja_con` are similar to those in the condition fields above for variable and constant value.

7.2.7 Time Constraints

The time constraint field `bj_times` in a job header consists of the following structure.

```
typedef   struct    {
        long            tc_nexttime;
        unsigned  char  tc_istime;
        unsigned  char  tc_mday;
        unsigned  short tc_nvaldays;
        unsigned  char  tc_repeat;
        unsigned  char  tc_nposs;
```

```
        unsigned  long  tc_rate;
} Timecon;
```

The field `tc_nexttime` gives the next time at which the job is to be executed.

The field `tc_istime` is non-zero to indicate that the time constraint is valid, otherwise the job is a "do when you can" job.

The field `tc_mday` is the target day of the month for "months relative to the beginning of the month" repeats, or the number of days back from the end of the month (possibly zero) for "months relative to the end of the month" repeats.

The field `tc_nvaldays` is the "days to avoid" field with Sunday being bit (1 << 0), Monday being bit (1 << 1), through to Saturday being bit (1 << 6). Holidays are represented by bit (1 << 7), also given by the `#define` constant `TC_HOLIDAYBIT`.

The field `tc_repeat` is set to one of the following values.

`TC_DELETE`	Run and delete
`TC_RETAIN`	Run and retain
`TC_MINUTES`	Repeat in minutes
`TC_HOURS`	Repeat in hours
`TC_DAYS`	Repeat in days
`TC_WEEKS`	Repeat in weeks
`TC_MONTHSB`	Repeat in months relative to the beginning
`TC_MONTHSE`	Repeat in months relative to the end
`TC_YEARS`	Repeat in years

The field `tc_nposs` is set to one of the following values

`TC_SKIP`	Skip if not possible
`TC_WAIT1`	Delay current if not possible
`TC_WAITALL`	Delay all if not possible
`TC_CATCHUP`	Run one and catch up

The field `tc_rate` gives the repetition rate (number of units).

7.2.8 Exit code structure

The job header field `bj_exits` consists of an instance of the following structure.

```
typedef   struct    {
        unsigned  char  nlower;
        unsigned  char  nupper;
        unsigned  char  elower;
        unsigned  char  eupper;
} Exits;
```

The 4 values give the ranges of exit codes to be considered “normal” or “error” respectively. If the ranges overlap, then an exit code falling inside both ranges will be considered to fall in the smaller of the two ranges. An exit code not “covered” will be treated as “abort”.

Chapter 8

The Variable Structure

The following structure is used to manipulate variables.

```
typedef struct  {
      unsigned long  var_sequence;
      vident         var_id;
      long           var_c_time, var_m_time;
      unsigned  char var_type;
      unsigned char  var_flags;
      char           var_name[BTV_NAME+1];
      char           var_comment[BTV_COMMENT+1];
      Btmode         var_mode;
      Btcon          var_value;
}  apiBtvar;
```

The field `var_sequence` is updated every time the variable is changed, but should not be relied upon within the API.

The field `var_id` consists of an instance of the following structure, which denotes the location of the variable on the *owning* host.

```
typedef   struct    {
        netid_t   hostid;
        slotno_t  slotno;
}  vident;
```

The field `hostid` refers to the owning host, and the `slotno` field refers to the slot number on the owning host. **Remember that this should not be confused with the slot number used by the API to refer to variables, which refers to the slot number on the host with which the API is in communication**. (Actually this may be the same if the variable belongs to that machine).

The field `var_c_time` refers to the creation time of the variable, but this is not currently maintained by the API.

The field `var_m_time` gives the time at which the variable was last modified.

The field `var_type` gives the type of the variable if it is a system variable, otherwise it is zero to denote that the variable is an ordinary variable. Values are as follows:

`VT_LOADLEVEL`	Maximum Load Level variable
`VT_CURRLOAD`	Current load level variable
`VT_LOGJOBS`	Log jobs variable
`VT_LOGVARS`	Log vars variable
`VT_MACHNAME`	Machine name (constant) variable
`VT_STARTLIM`	Max number of jobs to start at once
`VT_STARTWAIT`	Wait time

The field `var_flags` gives certain flag bits for the variable as follows:

`VF_READONLY`	Read-only system variable
`VF_STRINGONLY`	System variable which may take strings only
`VF_LONGONLY`	System variable which may take numeric only
`VF_EXPORT`	Variable is exported
`VF_CLUSTER`	Variable is “clustered”
`VF_SKELETON`	Variable is “outline” for variable on remote host.

Only the `VF_EXPORT` and `VF_CLUSTER` flags may be set by the user, the latter only if the former is set.

The fields `var_name` and `var_comment` give the name and comment fields of the variable.

The field `var_mode` gives the permissions for the variable in a similar manner to the corresponding field in the job header structure, as given for jobs.

The field `var_value` gives the current value of the variable as described in the job condition and assignment structures.

If a user has no read access to a variable, but does have “reveal” access, then the fields `var_comment` and `var_value` are zeroed when the variable is read. The completion of the `var_mode` field depends upon whether the user has “display mode” access.

Chapter 9

User profile structures

The profile of a given user is described via a structure of the following format.

```
typedef   struct    {
        unsigned  char  btu_isvalid,
                        btu_minp,
                        btu_maxp,
                        btu_defp;
        int_ugid_t      btu_user;
        unsigned  short btu_maxll;
        unsigned  short btu_totll;
        unsigned  short btu_spec_ll;
        unsigned  short btu_priv;
        unsigned  short btu_jflags[3];
        unsigned  short btu_vflags[3];
}  apiBtuser;
```

The field `btu_isvalid` denotes that the user description is valid. This will always be non-zero.

The fields `btu_minp`, `btu_maxp` and `btu_defp` give the minimum, maximum and default priorities respectively.

The fields `btu_maxll`, `btu_totll` and `btu_spec_ll` give the maximum per job load level, the total load level and the ‘special create” load levels respectively.

The field `btu_priv` gives the user’s privileges as a combination of some of the following bits.

`BTM_SSTOP`	Stop the scheduler
`BTM_UMASK`	Change own default permissions
`BTM_SPCREATE`	Special create permission
`BTM_CREATE`	Create new entries
`BTM_RADMIN`	Read administration file
`BTM_WADMIN`	Write admin file
`BTM_ORP_UG`	Or user and group permissions
`BTM_ORP_UO`	Or user and other permissions
`BTM_ORP_GO`	Or group and other permissions

The fields `btu_jflags` and `btu_vflags` give the default permissions for jobs and variables respectively, and the owner, group and ‘others” permission as three successive fields for each. These are bit maps with the same meanings as that given for the job and variables permissions.

Chapter 10

Default user profile

The default user profile is applied to all new users on the system and consists of the following fields.

```
typedef   struct    {
        unsigned  char  btd_minp,
                        btd_maxp,
                        btd_defp,
                        btd_version;
        unsigned  short btd_maxll;
        unsigned  short btd_totll;
        unsigned  short btd_spec_ll;
        unsigned  short btd_priv;
        unsigned  short btd_jflags[3];
        unsigned  short btd_vflags[3];
}  apiBtdef;
```

The meanings of the various fields are the same as the corresponding elements of the `apiBtuser` structure, defined above apart from `btd_version`, which contains the current release number of **GNUBatch**, currently 1.

Chapter 11

Command Interpreters

The following structure is used to describe command interpreters.

```
typedef   struct    {
        unsigned  short   ci_ll;
        unsigned  char    ci_nice;
        unsigned  char    ci_flags;
        char              ci_name[CI_MAXNAME+1];
        char              ci_path[CI_MAXFPATH+1];
        char              ci_args[CI_MAXARGS+1];
}  Cmdint;
```

The field `ci_ll` gives the default load level for the command interpreter. If this is given as zero in an gbatch_ciadd or gbatch_ciupd function call, then the user's special create load level is substituted.

The field `ci_nice` gives the nice value at which jobs will run.

The field `ci_flags` contains a combination of:

`CIF_SETARG0`	Insert job title as argument 0 of job
`CIF_INTERPARGS`	Expand environment variables and `` `` `` constructs in arguments.

The fields `ci_name`, `ci_path` and `ci_args` give the name, the path name and the prefix to the arguments for the command interpreter. Neither the path nor the arguments are checked for validity. **GNUBatch** assumes virtually everywhere that few changes will ever be made to command interpreters and that they are more or less the same on each connected host. Accordingly changes to the command interpreter list should be sparing.

Chapter 12

API Functions

The following sub-sections describe the **GNUBatch** API C routines including each function's purpose, syntax, parameters and possible return values.

The function descriptions also contain additional information that illustrate how the function can be used to carry out tasks.

Apart from the string manipulation functions and Unix versions of the two functions gbatch_jobadd and gbatch_jobdata which return a standard I/O file descriptor, all functions return an integer value. This is negative to indicate an error, or zero if all is well, apart from gbatch_open, which may return a positive value.

In some cases there are differences between the Unix and Windows variants, these are noted where appropriate.

The negative values have the following meanings.

Error code	Meaning
GBATCH_INVALID_FD	Invalid file descriptor
GBATCH_NOMEM	API unable to allocate memory
GBATCH_INVALID_HOSTNAME	Invalid host name
GBATCH_INVALID_SERVICE	Invalid service name
GBATCH_NODEFAULT_SERVICE	Default service not found
GBATCH_NOSOCKET	Unable to create socket
GBATCH_NOBIND	Unable to bind socket
GBATCH_NOCONNECT	Unable to make connection
GBATCH_BADREAD	Failure reading from socket
GBATCH_BADWRITE	Failure writing to socket
GBATCH_CHILDPROC	Unable to create child process
GBATCH_NOT_USER	Not relevant user
GBATCH_BAD_CI	Invalid command interpreter
GBATCH_BAD_CVAR	Bad variable in condition
GBATCH_BAD_AVAR	Bad variable in assignment
GBATCH_NOMEM_QF	No memory or disk space for queue file
GBATCH_NOCRPERM	No create permission
GBATCH_BAD_PRIORITY	Invalid priority
GBATCH_BAD_LL	Invalid load level
GBATCH_BAD_USER	Invalid user
GBATCH_FILE_FULL	File system full creating job
GBATCH_QFULL	IPC Message system full
GBATCH_BAD_JOBDATA	Invalid job data
GBATCH_UNKNOWN_USER	Unknown user specified
GBATCH_UNKNOWN_GROUP	Unknown group specified
GBATCH_ERR	Undefined error
GBATCH_NORADMIN	No read admin file permission
GBATCH_NOCMODE	No change permissions permission
GBATCH_UNKNOWN_COMMAND	Unknown command in gbatch_jobop
GBATCH_SEQUENCE	Sequence error
GBATCH_UNKNOWN_JOB	Unknown job
GBATCH_UNKNOWN_VAR	Unknown variable
GBATCH_NOPERM	No permission for operation

Error code	Meaning
GBATCH_INVALID_YEAR	Invalid year in holiday file operations
GBATCH_ISRUNNING	Job is running
GBATCH_NOTIMETOA	Job has no time to advance
GBATCH_VAR_NULL	Null variable name
GBATCH_INVALIDSLOT	Invalid slot number
GBATCH_ISNOTRUNNING	Job is not running
GBATCH_NOMEMQ	No memory for queue name
GBATCH_NOPERM_VAR	No permission on variable(s) referenced in job
GBATCH_RVAR_LJOB	Remote variable in local job
GBATCH_LVAR_RJOB	Local variable in remote job
GBATCH_MINPRIV	Too few permissions given
GBATCH_SYSVAR	Invalid operation on system variable
GBATCH_SYSVTYPE	Invalid type assignment attempted to system variable
GBATCH_VEXISTS	Variable exits
GBATCH_DSYSVAR	Attempt to delete system variable
GBATCH_DINUSE	Attempt to delete variable in use
GBATCH_DELREMOTE	Attempt to delete remote variable.
GBATCH_NO_PASSWD	A password is required for the user
GBATCH_PASSWD_INVALID	The supplied password is invalid.
GBATCH_BAD_GROUP	Invalid group name (inaccessible to user).
GBATCH_NOTEXPORT	“Cluster” set on variable but not “Export”
GBATCH_RENAMECLUST	Attempt to rename clustered variable

12.1 Sign-on and off

12.1.1 gbatch_open

```
int gbatch_open(const char *hostname, const char *servname)
int gbatch_open(const char *hostname, const char *servname, const char *username)
/* Windows */
int gbatch_login(const char *hostname, const char *servname, const char *username,
char *passwd)
int gbatch_wlogin(const char *hostname, const char *servname, const char *username,
char *passwd)
int gbatch_locallogin(const char *servname, const char *username)
int gbatch_locallogin_byid(const char *servname, const int ugid_t uid)
```

The function gbatch_open is used to open a connection to the **GNUBatch** API. There are some variations in the semantics depending upon whether the caller is known to be a Unix host or a Windows or other client. This can be controlled by settings in the servers host file, typically `/usr/local/etc/gnubatch.hosts` and the user map file `/usr/local/etc/gbuser.map`.

The server will know that the caller is a Unix host if it appears in the hosts file as a potential server, maybe with a `manual` keyword to denote that it shouldn't be connected unless requested (with gbch-conn). In such cases user names will be taken as Unix user names.

In other cases the user names will be taken as Windows Client user names to be mapped appropriately.

Windows user names are mapped on the server to Unix user names using the user map file and constructs in the host file, with the latter taking priority.

Note that it is possible to use a different set of passwords on the server from the users' login passwords, setting them up with gbch-passwd. This is desirable in preference to people's login passwords appearing in various interface programs.

All of these functions return non-negative (possibly zero) on success, this should be quoted in all other calls.

In the event of an error, then a negative error code is returned as described on page 28.

gbatch_open may be used to open a connection with the current effective user id on Unix systems, or (using the extra `username` parameter a predefined connection for the given user on Windows systems.

No check takes place of passwords for Unix connections, but the call will only succeed on Windows systems if the client has a fixed user name assigned to it.

This happens if the client matches entries in `/usr/local/etc/gnubatch.hosts` of the forms:

```
mypc - client(unixuser) unixuser winuser clienthost(mypc)
```

The call will succeed in the first instance if the user is mapped to `unixuser` and running on `mypc`.

In the second case it will succeed if it is running on `mypc` and `winuser` is given in the call, whereupon it will be mapped to `unixuser`.

This is over-complicated, potentially insecure, and preserved for compatibility only, and gbatch_open should only really be used on Unix hosts to log in with the effective user id.

gbatch_login should normally be used to open a connection to the API with a username and password. If the client is not registered as a Unix client, then the user name is mapped to a user name on the server as specified in the user map file or the hosts file. The password should be that for the user mapped to (possibly as set by gbch-passwd rather than the login password).

gbatch_wlogin is similar to gbatch_login, but guarantees that the user name will be looked up as if the caller were not registered as Unix client so that there are no surprises if this is changed.

gbatch_locallogin and gbatch_locallogin_byid may be used to set up an API connection on the same machine as the server without a password. The `username`, if not null, may be used to specify a user other than that of the effective user id. To use a user other than the effective user id, *Write Admin* permission is required.

In all cases, `hostname` is the name of the host being connected to or null to use the loopback interface. `servname` may be NULL to use a standard service name, otherwise an alternative service may be specified. Note that more than one connection can be open at any time with various combinations of user names and hosts.

When finished, close the conection with a call to gbatch_close.

12.1.1.1 Return values

The function returns a positive value if successful, which is the file descriptor used in various other calls, otherwise one of the error codes listed on page 28 onwards, all of which are negative.

12.1.1.2 Example

```
int fd;
fd = gbatch_open("myhost", (char *) 0);
if (fd < 0)  { /* handle error */
    ...
}
        ...
gbatch_close(fd)
```

12.1.2 gbatch_close

```
int gbatch_close(const int fd)
```

The function gbatch_close is used to terminate a connection with the API.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

12.1.2.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

In most API programs the return value is ignored as it is only likely to report an error if an invalid API descriptor is passed.

12.1.3 gbatch_newgrp

```
int gbatch_newgrp(const int fd, const char * group)
```

The function gbatch_newgrp is used to select a new primary group as the user's primary group.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`group` is the required group name to be selected. If the user has *write admin file* privilege, this may be any valid group name, otherwise the group must be the user's default group or one of the user's supplementary groups.

12.1.3.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.1.4 gbatch_setqueue

```
int gbatch_setqueue(const int fd, const char *queuename)
```

The function gbatch_setqueue is used to allocate a queue name for transactions with the API. This may effect the selection of jobs and job titles.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`queuename` is the name of the proposed queue or NULL to delete a previous queue name.

12.1.4.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.1.5 gbatch_gethenv

```
char **gbatch_gethenv(const int fd)
```

The function gbatch_gethenv is used to obtain a copy of the static environment file for the server. This will provide the environment variables which a job running on that server would have unless overridden by separate environment variables in the job.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

The result is a vector of character pointers containing environment variable assignments of the form `NAME=VALUE`. This list is terminated by a null pointer. If there is no static environment file, an empty list is returned, i.e. it will be a pointer to a `char *` location containing NULL.

Unlike other routines, the user has the responsibility to deallocate the space allocated, each string and the overall vector, when not required.

12.1.5.1 Return values

The function returns a null-terminated vector of character vectors if successful, otherwise it returns NULL and one of the error codes listed on page 28 onwards is assigned to the external variable `gbatch_dataerror`.

12.1.6 gbatch_holread

```
int gbatch_holread(const int fd,
                   const unsigned flags,
                   int year,
                   unsigned char *bitmap)
```

The function gbatch_holread is used to read the holiday file for the specified year.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is currently unused but is reserved for future extensions.

`year` is the year for which the holiday file is required. This should be the actual number of the year or an offset from 1900. For example the year 1994 could be given as 1994 or 94. Note: The offset value should be less than 200.

bitmap is an array of characters representing the bitmap. Bits are set if the days is a holiday. To test the bitmap use the following formula:

```
if (bitmap[day >> 3] & (1 << (day & 7)))
    /*day is holiday*/
```

12.1.6.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.1.7 gbatch_holupd

```
int gbatch_holupd(const int fd,
                  const unsigned flags,
                  int year,
                  unsigned char *bitmap)
```

The function gbatch_holupd is used to update the holiday file for the specified year.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is currently unused but is reserved for future use.

`year` is the year for which the holiday file is required. This should be the actual number of the year or an offset from 1900. For example the year 1994 could be given as 1994 or 94. Note: The offset value should be less than 200.

`bitmap` is an array of characters representing the bitmap. Bits are set if the days is a holiday. To test the bitmap use the following formula:

```
if (bitmap[day >> 3] & (1 << (day & 7)))
    /*day is holiday*/
```

12.1.7.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2 Job access

12.2.1 gbatch_joblist

```
int gbatch_joblist(const int fd,
                   const unsigned flags,
                   int *numjobs,
                   slotno_t **slots)
```

The function gbatch_joblist is used to get a list of jobs from the API.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero, or a logical OR of one or more of the following values

`GBATCH_FLAG_LOCALONLY`	Ignore remote jobs/hosts, i.e. not local to the server, not the client.
`GBATCH_FLAG_QUEUEONLY`	Restrict to the selected queue (with gbatch_setqueue) only.
`GBATCH_FLAG_USERONLY`	Restrict to the user only.
`GBATCH_FLAG_GROUPONLY`	Restrict to the current group (possibly as selected by gbatch_newgrp) only.

`numjobs` is a pointer to an integer which upon return will contain the number of jobs in the list.

`slots` will upon return contain a list of slot numbers, each of which can be used to access an individual job. The memory used by this array is owned by the API and therefore the user should not attempt to deallocate it.

12.2.1.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.1.2 Example

```
int fd, ret, cnt, numjobs;

ret = gbatch_joblist(fd, 0, &numjobs, &list);
if (ret < 0) {  /* handle error */
    .  .  .
}

for (cnt = 0; cnt < numjobs; cnt++)  {
    slotno_t this_slot = list[cnt];
```

```
    /* process this_slot */
}

/* do not try to deallocate the list
```

12.2.2 gbatch_jobfind

```
int gbatch_jobfind(const int fd,
                   const unsigned flags,
                   const jobno_t jobnum,
                   const netid_t nid,
                   slotno_t *slot,
                   apiBtjob *jobd)

int gbatch_jobfindslot(const int fd,
                       const unsigned flags,
                       const jobno_t jobnum,
                       const netid_t nid,
                       slotno_t *slot)
```

The function gbatch_jobfind is used to retrieve the details of a job, starting from the job number, in one operation.

The function gbatch_jobfindslot is used to retrieve just the slot number of a job, starting from the job number.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or the logical OR of one or more of the following bits:

`GBATCH_FLAG_LOCALONLY`	Search for jobs local to the server only.
`GBATCH_FLAG_USERONLY`	Search for jobs owned by the user only.
`GBATCH_FLAG_GROUPONLY`	Search for jobs owned by the group only.
`GBATCH_FLAG_QUEUEONLY`	Search for jobs with the queue name specified by gbatch_setqueue only.

jobnum is the job number to be searched for.

`nid` is the IP address (in network byte order) of the host on which the searched-for job is to be located. It should be correct even if `GBATCH_FLAG_LOCALONLY` is specified.

`slot` is assigned the slot number corresponding to the job. It may be null is not required, but this would be nearly pointless with gbatch_jobfindslot (other than reporting that the job was unknown).

`jobp` is a pointer to a structure to contain the details of the job for gbatch_jobfind.

The definition of the job structure is given on page 12 onwards.

12.2.2.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.3 gbatch_jobread

```
int gbatch_jobread(const int fd,
                   const unsigned flags,
                   const slotno_t slot,
                   apiBtjob *jobd)
```

The function gbatch_jobread is used to retrieve the details of a job

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

`jobp` is a pointer to a structure to contain the details of the job.

The definition of the job structure is given on page 12 onwards.

12.2.3.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.4 gbatch_jobdata

```
FILE *gbatch_jobdata(const int fd,
                     const int flags,
                     const slotno_t slot)

int gbatch_jobdata(const int fd,
                   const int outfile,
                   int (*fn)(int,void*,unsigned),
                   const unsigned, flags,
                   const slotno_t slotno)
```

The function gbatch_jobdata is used to retrieve the job script of a job. There are two versions, one for the Unix and Linux API and one for the Windows API. The second form is used under Windows as there is no acceptable substitute for the pipe(2) system call.

In both forms of the call, `fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

The difference between the two versions of gbatch_jobadd is in the method of passing the job script.

12.2.4.1 Unix and Linux

The Unix and Linux API version returns a stdio file descriptor which may be used with the standard I/O functions getc(3), fread(3) etc to read the job script. The job script should always be read to the end and then using fclose(3) to ensure that all incoming data on the socket is collected.

If there is any kind of error, then gbatch_jobdata will return NULL, leaving the error code in the external variable `gbatch_dataerror`.

12.2.4.2 Windows

In the case of the Windows version, the specified function fn is invoked with parameters similar to write to read data to pass across as the job script, the argument `outfile` being passed as a file handle as the first argument to `fn`.

`fn` may very well be write. The reason for the routine not invoking write itself is partly flexibility but mostly because some versions of Windows DLLs do not allow write to be invoked directly from within them.

N.B. This routine is particularly susceptible to peculiar effects due to assignment of insufficient stack space.

The return value is zero for success, or an error code. The error code is also assigned to the external variable `gbatch_dataerror` for consistency with the Unix version.

12.2.4.3 Return values

The Unix version of gbatch_jobdata returns NULL if unsuccessful, placing the error code in the external variable `gbatch_dataerror`.

The Windows version of gbatch_jobdata returns zero if successful, otherwise an error code.

The error codes which may be returned are listed on page 28 onwards.

12.2.4.4 Example

```
int fd, ret, ch;
FILE *inf;
slotno_t slot;

/*  select a job assign it to slot */

inf = gbatch_jobdata(fd, XBABI_IGNORESEQ, slot);

if (!inf) { /* error in gbatch_dataerror*/
    .  .  .
}

while ((ch = getc(inf)) != EOF)
    putchar(ch);

fclose(inf);
```

12.2.5 gbatch_jobadd

```
FILE *gbatch_jobadd(const int fd,
                    apiBtjob *jobd)

int gbatch_jobres(const int fd,
                  jobno_t *jobno)

int gbatch_jobadd(const int fd,
                  const int infile,
                  int(*fn)(int,void*,unsigned),
                  apiBtjob *jobd)
```

The function gbatch_jobadd, is used to create a new **GNUBatch** job.

There are two forms of gbatch_jobadd. The first form, together with gbatch_jobres, is used to create jobs using the Unix or Linux version of the API.

The second form is used under Windows as there is no acceptable substitute for the pipe(2) system call.

In both forms of the call, `fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`jobd` is a pointer to a structure containing the attributes of the job to be created apart from the job script.

The definition of the job structure is given on page 12 onwards.

The difference between the two versions of gbatch_jobadd is in the method of passing the job script.

12.2.5.1 Unix and Linux

The Unix and Linux API version returns a stdio file descriptor which may be used with the standard I/O functions `fputs(3)`, `fprintf(3)` etc to write the job script. When complete, the job script should be closed using fclose(3). The result of the job submission is then collected using the gbatch_jobres routine, which assigns the job number to the contents of the `jobno` parameter and returns zero as its result. The job number is also placed into the `bj_job` field in the job structure.

For reasons of correctly synchronising socket messages, be sure to call gbatch_jobres immediately after the call to fclose(3), even if you do not require the answer.

If there is any kind of error, then depending upon at what point the error is detected, either gbatch_jobadd will return NULL, leaving the error code in the external variable `gbatch_dataerror`, or gbatch_jobres will return the error as its result rather than zero.

12.2.5.2 Windows

In the case of the Windows version, the specified function `fn` is invoked with parameters similar to read to read data to pass across as the job script, the argument infile being passed as a file handle as the first argument to fn.

`fn` may very well be read. The reason for the routine not invoking read itself is partly flexibility but mostly because some versions of Windows DLLs do not allow read to be invoked directly from within them.

N.B. This routine is particularly susceptible to peculiar effects due to assignment of insufficient stack space.

The return value is zero for success, in which case the job number will be assigned to the `bj_job` field of `jobd`, or an error code. The error code is also assigned to the external variable `gbatch_dataerror` for consistency with the Unix version.

12.2.5.3 Return values

The Unix version of gbatch_jobadd returns NULL if unsuccessful, placing the error code in the external variable `gbatch_dataerror`.

The Windows version of gbatch_jobadd and the gbatch_jobres under Unix return zero if successful, otherwise an error code.

The error codes which may be returned are listed on page 28 onwards.

12.2.5.4 Example

This example creates a job from standard input:

```
int fd, ret, ch;
FILE *outf;
jobno_t jn;
apiBtjob outj;

fd = gbatch_open("myhost", (char *) 0);
if (fd < 0) { /* error handling */
                . . .
}

/* always clear the structure first */
memset((void *)&outj, '\0', sizeof(outj));

/* only the following parameters are compulsory */

outj.h.bj_pri = 150;
outj.h.bj_ll = 1000;
outj.h.bj_mode.u_flags = JALLMODES;
outj.h.bj_exits.elower = 1;
outj.h.bj_eupper = 255;
outj.h.bj_ulimit = 0x10000;
strcpy(outj.h.bj_cmdinterp, "sh"); /* NB assumes sh defined */
gbatch_putdirec(&outj, "~/work");

/* set progress code to zero */
```

```
outj.h.bj_progress = BJP_CANCELLED;

/* set up a time constraint */
outj.h.bj_times.tc_istime = 1;
outj.h.bj_times.tc_nexttime = time(long *)0) + 300;
outj.h.bj_times.tc_repeat = TC_MINUTES;
outj.h.bj_times.tc_rate = 10;
outj.h.bj_times.tc_nposs = TC_SKIP;

gbatch_puttitle(&outj, "MyTitle");

outf = gbatch_jobadd(fd, &outj);
if  (!outf)  {  /* error in gbatch_dataerror*/
     .  .  .
}

while ((ch = getchar()) != EOF)
    putc(ch, outf);
fclose(outf);
ret = gbatch_jobres(fd, &jn);
if (ret < 0) {  /* error in ret */
    .  .  .
}
else
    printf("job number is %ld\n", jn);

gbatch_close(fd);
```

12.2.6 gbatch_jobdel

```
int gbatch_jobdel(const int fd, const unsigned flags, const slotno_t slot)
```

The function gbatch_jobdel is used to delete a job.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

12.2.6.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.6.2 Example

To delete all jobs for a user.

```
int fd, ret, cnt, numjobs;
slotno_t *list;

fd = gbatch_open("myhost", (char *) 0);
ret = gbatch_joblist(fd, GBATCH_FLAG_USERONLY, &numjobs, &list);
if (fd < 0)  { /* handle error */
    .  .  .
}

for (cnt = 0; cnt < numjobs; cnt++)  {
    if ((ret = gbatch_jobdel(fd, GBATCH_FLAG_IGNORESEQ, list[cnt])  {
        /* handle error */
        .  .  .
    }
}

gbatch_close(fd);
```

12.2.7 gbatch_jobupd

```
int gbatch_jobupd(const int fd,
                  const unsigned flags,
                  const slotno_t slot,
                  apiBtjob * jobd)
```

The function gbatch_jobupd is used to update the details of a job.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`jobp` is a pointer to a structure containing the details of the job.

The definition of the job structure is given on page 12 onwards.

12.2.7.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.8 gbatch_jobchown

```
int gbatch_jobchown(const int fd,
                    const unsigned flags,
```

```
                const slotno_t slot,
                const char *newowner)
```

The function gbatch_jobchown is used to change the ownership of a job

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

`newowner` is the user name of the prospective new owner.

12.2.8.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.9 gbatch_jobchgrp

```
int gbatch_jobchgrp(const int fd,
                const unsigned flags,
                const slotno_t slot,
                const char *newgroup)
```

The function gbatch_jobchgrp is used to attempt to change the group ownership of a job.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

`newgroup` is a valid group name.

12.2.9.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.10 gbatch_jobchmod

```
int gbatch_jobchmod(const int fd,
                const unsigned flags,
                const slotno_t slot,
                const Btmode *newmode)
```

The function gbatch_jobchmod is used to change the permissions of a job.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

`newmode` is a pointer to a structure containing the details of the new mode.

The definition of the job structure is given on page 12 onwards.

12.2.10.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.11 gbatch_jobop

```
int gbatch_jobop(const int fd,
                 const unsigned flags,
                 const slotno_t slot,
                 const unsigned op,
                 const unsigned param)
```

The function gbatch_jobop is used to perform an operation on a job.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the job list.

`slot` is the slot number corresponding to the job as returned by gbatch_joblist or gbatch_jobfindslot.

`op` is one of the following:

`GBATCH_JOP_SETRUN`	Set job running
`GBATCH_JOP_SETCANC`	Cancel a job
`GBATCH_JOP_FORCE`	Force a job to start
`GBATCH_JOP_FORCEADV`	Force to start and advance time
`GBATCH_JOP_ADVTIME`	Advance to next repeat
`GBATCH_JOP_KILL`	Kill job

`param` is only relevant to `GBATCH_JOP_KILL`, in which case it gives the signal number to kill the job.

12.2.11.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.2.12 gbatch_jobmon

```
int gbatch_jobmon(const int fd, void (*fn)(const int))
```

```
int gbatch_setmon(const int fd, HWND hWnd, UINT wMsg)

int gbatch_procmon(const int fd)

void gbatch_unsetmon(const int fd)
```

12.2.12.1 Unix and Linux

The function gbatch_jobmon is used to set a function to monitor changes to the job queue.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`fn` is a pointer to a function which must be declared as returning `void` and taking one `const int` argument. Alternatively, this may be NULL to cancel monitoring.

The function `fn` will be called upon each change to the job list. The argument passed will be `fd`. Note that any changes to the job queue are reported (including changes on other hosts whose details are passed through) as the API does not record which jobs the user is interested in.

12.2.12.2 Windows

The gbatch_setmon routine may be used to monitor changes to the job queue or variable list. Its parameters are as follows.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`hWnd` is a windows handle to which messages should be sent.

`wMsg` is the message id to be passed to the window (`WM_USER` or a constant based on this is suggested).

To decode the message, the gbatch_procmon is provided. This returns `XBWINAPI_JOBPROD` to indicate a change or changes to the job queue and `XBWINAPI_VARPROD` to indicate a change or changes to the variable list. If there are changes to both, two or more messages will be sent, each of which should be decoded via separate gbatch_procmon calls.

To cancel monitoring, invoke the routine

```
gbatch_unsetmon(fd)
```

If no monitoring is in progress, or the descriptor is invalid, this call is just ignored.

12.2.12.3 Return values

The function gbatch_jobmon returns 0 if successful otherwise the error code `GBATCH_INVALID_FD` if the file descriptor is invalid. Invalid fn parameters will not be detected and the application program will probably crash.

12.2.12.4 Example

```
void note_mod(const int fd)
{
    job_changes++;
}

.  .  .

gbatch_jobmon(fd, note_mod);
    .  .  .

if (job_changes) { /* handle changes */
    .  .  .
```

12.3 Job fields

12.3.1 gbatch_getarg

```
const char *gbatch_getarg(const apiBtjob *jobp, const unsigned indx)
```

The function gbatch_getarg is used to extract an argument string from a job string table.

`jobp` is a pointer to a structure containing the details of the job.

The definition of the job structure is given on page 12 onwards.

`indx` is the argument number required. This should be between 0 and 1 less than the total number of arguments (given by the field `jobp->h.bj_args`).

12.3.1.1 Return values

If successful the function will return the required argument as a `const` character string otherwise NULL if the argument number is invalid.

12.3.2 gbatch_getdirect

```
const char *gbatch_getdirect(const apiBtjob *jobp)
```

The function gbatch_getdirect is used to extract the working directory of a job from the string table of the job.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

12.3.2.1 Return values

The result is the working directory of the job as a `const` character string, or NULL if this is not set (but this is almost certainly an error).

12.3.3 gbatch_getenv

```
const char *gbatch_getenv(const apiBtjob *jobp, const char *name)
```

The function gbatch_getenv is used to extract an environment variable from a job string table.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`name` is the environment varible required.

12.3.3.1 Return values

The result is the environment variable value as a `const` character string or NULL if the variable does not exist in the job (perhaps because it is in the static environment file).

12.3.4 gbatch_getenvlist

```
const char **gbatch_getenvlist(const apiBtjob *jobp), const char *name)
```

The function gbatch_getenvlist is used to extract the list of environment variables from a job string table.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

12.3.4.1 Return values

The result is a null-terminated vector of environment variables in the form `NAME=VALUE`, or NULL if memory could not be allocated for it.

The space is allocated within the API. The user should not attempt to free it after use. Also note that the space is re-used if gbatch_getenv is invoked on a different job, the result should be copied if needed.

12.3.5 gbatch_getredir

```
const apiMredir *gbatch_getredir(const apiBtjob *jobp,
                                 const unsigned indx)
```

The function gbatch_getredir is used to extract a redirection structure from a job structure.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`indx` is the redirection number required. This should be between 0 and 1 less than the number of redirections as given by the field `jobp->h.bj_nredirs`.

12.3.5.1 Redirection structure

The format of the redirection structure is as follows:

```
typedef struct {
        unsigned char fd;
        unsigned char action;
        union  {
                unsigned short arg;
                const char *buffer;
        }  un;
} apiMredir;
```

In this structure `fd` represents the file descriptor, and `action` gives the action required as follows:

`RD_ACT_RD`	Open file name given in `un.buffer` for reading.
`RD_ACT_WRT`	Open file name given in `un.buffer` for writing.
`RD_ACT_APPEND`	Append to file name given in `un.buffer`, opened for writing.
`RD_ACT_RDWR`	Open file name given in `un.buffer` for read/write.
`RD_ACT_RDWRAPP`	Open file name given in `un.buffer` for read/write and append.
`RD_ACT_PIPEO`	Open pipe to shell command given in `un.buffer` for output.
`RD_ACT_PIPEI`	Open pipe from shell command given in `un.buffer` for input.
`RD_ACT_CLOSE`	Close file descriptor.
`RD_ACT_DUP`	Duplicate file descriptor given in `un.arg`.

12.3.5.2 Return values

The result is a pointer to a static structure containing the required redirection of the job NULL if the redirection number is invalid.

Note that the structure used will be overwritten by a further call to gbatch_getredir with different arguments, hence it should be copied if required.

12.3.5.3 gbatch_gettitle

```
const char *gbatch_gettitle(const int fd, const apiBtjob *jobp)
```

The function gbatch_gettitle may be used to extract the title from a job as a character string. Optionally the queue name (as set by gbatch_setqueue) may be elided from the title.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open, or -1 to disregard the queue name.

jobp is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

12.3.5.4 Return values

The result is the title of the job as a `const` character string.

If a valid file descriptor is provided, and this has a queue name set using gbatch_setqueue, and the queue name is the same as that in the job title, then the queue name is deleted from the title returned to the user.

12.3.6 gbatch_delarg

```
int gbatch_delarg(apiBtjob *jobp, const unsigned indx)
```

The function gbatch_delarg is used to delete an argument from a job structure string table.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`indx` is the number of the argument being deleted. Note that all the following arguments are "moved up" the list and their index numbers will be reduced by one.

12.3.6.1 Return values

The result is non-zero if successful, or zero if the string table overflowed, an likely event in the case of gbatch_delarg.

12.3.7 gbatch_delenv

```
int gbatch_delenv(const apiBtjob *jobp, const char *name)
```

The function gbatch_getenv is used to delete an environment variable from a job string table.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`name` is the environment variable to be deleted.

12.3.7.1 Return values

The result is non-zero if successful, or zero if the string table overflowed, an unlikely event in the case of a deletion.

No error is reported if the specified variable does not exist.

12.3.7.2 Notes

Environment variables common to all jobs may be held in a "static environment file" to which the job structure environment variables represent differences only. This routine will not affect entries in the static environment file.

12.3.8 gbatch_delredir

```
int gbatch_delredir(apiBtjob *jobp, const unsigned indx)
```

The function gbatch_delredir is used to delete a redirection from a job structure string table.

The definition of the job structure is given on page 12 onwards.

jobp is a pointer to a structure containing the details of the job.

indx is the number of the redirection. Note that any subsequent redirections are "moved up" one place as a result of this function and their index numbers reduced by one.

12.3.8.1 Return values

The result is non-zero if successful, or zero if the string table overflowed, an likely event in the case of gbatch_delredir.

12.3.9 gbatch_putarg

```
int gbatch_putarg(apiBtjob *jobp, const unsigned indx, const char *newarg)
```

The function gbatch_putarg is used to replace or add a new argument to the argument list of a job.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`indx` is the number of the argument to be replaced or added. This may be greater than any number of existing arguments if required, in which case any intervening arguments are created as empty strings.

`newarg` is the character string containing the new argument.

12.3.9.1 Return values

The result is non-zero if successful or zero if the string table overflowed. In the latter case the contents of the string table should not be relied upon. The job structure should be saved first if in doubt.

12.3.10 gbatch_putargglist

```
int gbatch_putarglist(apiBtjob *jobp, const char **alist)
```

The function gbatch_putarglist is used to replace the entire argument list within a string table of a job.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`alist` is a vector of strings containing the new arguments.

The new argument list completely replaces the old

12.3.10.1 Return values

The result is non-zero if successful or zero if the string table overflowed. In the latter case the contents of the string table should not be relied upon. The job structure should be saved first if in doubt.

12.3.11 gbatch_putdirect

```
int gbatch_putdirect(apiBtjob *jobp, const char *direct)
```

The function gbatch_putdirect is used to insert a new working directory name into a job structure.

`jobp` is a pointer to a structure containing the job details. The definition of the job structure is given on page 12 onwards.

`direct` is the name of the directory to be inserted.

12.3.11.1 Return values

The result will be non-zero if successful or zero if the string table overflowed. In the latter case the string table contents of the job should not be relied upon. The job structure should be saved first if in doubt.

12.3.12 gbatch_putenv

```
const char *gbatch_putenv(const apiBtjob *jobp, const char *name)
```

The function gbatch_putenv is used to insert an environment variable into a job string table.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`name` is the environment varible required, in the form NAME=VALUE.

12.3.12.1 Return values

This function will return non-zero if successful otherwise zero if the string table overflowed. In the latter case the contents of the job structure should not be relied upon. If in doubt copy the job structure first.

12.3.13 gbatch_putenvlist

```
int gbatch_putenvlist(const apiBtjob *jobp, const char **elist)
```

The function gbatch_putenv is used to replace the entire environment variable list of a job string table.

`jobp` is a pointer to a structure containing the details of the job. The definition of the job structure is given on page 12 onwards.

`elist` is a null-terminated list of environment variables. Each should be of the form `NAME=VALUE`. Any entries not in this form are ignored.

12.3.13.1 Return values

The result will be no-zero if successful or zero if the string table overflowed. In the latter case the string table contents of the job should not be relied upon. The job structure should be saved first if in doubt.

12.3.13.2 Notes

Remember that these entries merely override settings in any “static environment file” on the server running the job.

12.3.14 gbatch_putredir

```
int gbatch_putredir(apiBtjob *jobp,
                    const unsigned indx,
                    const apiMredir *newredir)
```

The function gbatch_putredir is used to insert a new or replacement redirection into a job structure.

`jobp` is a pointer to a structure containing the job details. The definition of the job structure is given on page 12 onwards.

`indx` is the number of the redirection to be inserted or replaced (starting at zero). This should be equal to the number of existing redirections to create a new one.

`newredir` is the redirection structure representing the redirection to be inserted or replaced.

Details of the redirection structure and fields therein are documented under gbatch_getredir on page 48.

12.3.14.1 Return values

The result will be no-zero if successful or zero if the string table overflowed. In the latter case the string table contents of the job should not be relied upon. The job structure should be saved first if in doubt.

12.3.15 gbatch_putredirlist

```
int gbatch_putredirlist(apiBtjob *jobp,
                        const apiMredir *rdlist,
                        const unsigned num)
```

The function gbatch_putredirlist is used to replace the entire redirection list for a job in one operation.

`jobp` is a pointer to a structure which contains the job details. The definition of the job structure is given on page 12 onwards.

`rdlist` is a vector of redirections.

Details of the redirection structure and fields therein are documented under gbatch_getredir on page 48.

`num` is the number of elements in `rdlist`.

The new redirection list completely replaces the old.

12.3.15.1 Return values

The function will return non-zero if successful otherwise zero if the string table overflowed. In the latter case the contents of the job should no be relied upon, the job structure should be saved first if in doubt.

12.3.16 gbatch_puttitle

```
int gbatch_puttitle(const int fd, apiBtjob *jobp, const char *title)
```

The function gbatch_puttitle is used to insert a new or replacement title into the string table of a job structure, possibly automatically inserting the current queue name as set by gbatch_setqueue.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open, or -1 to disregard any queue name set by gbatch_setqueue.

`jobp` is a pointer which contains the details of the job.

The definition of the job structure is given on page 12 onwards.

`title` is the required new title or NULL if the title is to be deleted. If fd is a valid API descriptor, then any queue name set by gbatch_setqueue will be added to it.

12.3.16.1 Return values

The result will be no-zero if successful or zero if the string table overflowed. In the latter case the string table contents of the job should not be relied upon. The job structure should be saved first if in doubt.

12.4 Variable access

12.4.1 gbatch_varlist

```
int gbatch_varlist(const int fd,
                   const unsigned flags,
                   int *numvars,
                   slotno_t **slots)
```

The function gbatch_varlist is used to obtain a vector of slots which can be used to access the details of variables readable by the user.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero, or a logical OR of one or more of the following values

`GBATCH_FLAG_LOCALONLY`	Ignore remote variables/hosts, i.e. not local to the server, not the client.
`GBATCH_FLAG_USERONLY`	Restrict to the user only.
`GBATCH_FLAG_GROUPONLY`	Restrict to the current group (possibly as selected by gbatch_newgrp) only.

`numvars` is a pointer to an integer which will contain the number of variables in the list.

`slots` is a pointer to an array of slots. The memory used by this list should not be freed by the user as it is owned by the API.

12.4.1.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.2 gbatch_varfind

```
int gbatch_varfind(const int fd,
                   const unsigned flags,
                   const char *vname,
                   const netid_t nid,
                   slotno_t *slot,
                   apiBtvar *vard)

int gbatch_varfindslot(const int fd,
                       const unsigned flags,
```

```
                const char *vname,
                const netid_t nid,
                slotno_t *slot)
```

The function gbatch_varfind is used to retrieve the details of a variable, starting from its name, in one operation.

The function gbatch_varfindslot is used to retrieve just the slot number of a variable, starting from its name.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or the logical OR of one or more of the following bits:

`GBATCH_FLAG_LOCALONLY`	Search for variables local to the server only.
`GBATCH_FLAG_USERONLY`	Search for variables owned by the user only.
`GBATCH_FLAG_GROUPONLY`	Search for variables owned by the group only.

`vname` is the variable name to be searched for.

`nid` is the IP address (in network byte order) of the host on which the searched-for variable is to be located. It should be correct even if `GBATCH_FLAG_LOCALONLY` is specified.

`slot` is assigned the slot number corresponding to the variable. It may be null is not required, but this would be nearly pointless with gbatch_varfindslot (other than reporting that the variable was unknown).

`vard` is a pointer to a structure which will contain the details of the variable for `gbatch_varfind`. The definition of the variable structure is given on page 22 onwards.

12.4.2.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.3 gbatch_varread

```
int gbatch_varread(const int fd,
                const unsigned flags,
                const slotno_t slot,
                apiBtvar *vard)
```

The function gbatch_varread is used to read the details for a variable

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfind.

`vard` is a pointer to a structure which will contain the details of the variable. The definition of the variable structure is given on page 22 onwards.

12.4.3.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.4 gbatch_varadd

```
int gbatch_varadd(const int fd, apiBtvar *vard)
```

The function gbatch_varadd is used to create a new variable.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`vard` is a pointer to a structure which contains the details of the new variable. The definition of the variable structure is given on page 22 onwards.

12.4.4.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.4.2 Example

```
int fd, ret
int apiBtvar outv;

fd = gbatch_open("myhost", (char *) 0);
if (fd < 0)  { /* error handling */
    ...
}
memset((void *)&outv, '\0', sizeof(outv));
strcpy(outv.var_name, "var1");
strcpy(outv.var_comment, "A comment");
outv.var_value.const_type = CON_LONG;
outv.var_value.con_un.con_long = 1;
outv.var_mode.u_flags = VALLMODES;
ret = gbatch_varadd(fd, &outv);
if (ret < 0) { /* error handling */
    ...
}
gbatch_close(fd);
```

12.4.5 gbatch_vardel

```
int gbatch_vardel(const int fd, const unsigned flags, const slotno_t slot)
```

The function gbatch_vardel is used to delete a variable from the variable list.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is 0 or `GBATCH_FLAG_IGNORESEQ` to attempt to ignore recent changes to the variable list.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

12.4.5.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.5.2 Example

This example deletes all the variables owned by the user.

```
int fd, ret, cnt;
int numvars;
slotno_t *list;

fd = gbatch_open("myhost", (char *)0);
ret = gbatch_varlist(fd, GBATCH_FLAG_USERONLY, &numvars, &list);
if (fd < 0)  { /* process error */
     .  .  .
}

for (cnt = 0; cnt < numvars, cnt++) {
    if  ((ret = gbatch_vardel(fd, 0, list[cnt])) < 0)   {
         /* process error */
         .  .  .
```

12.4.6 gbatch_varupd

```
int gbatch_varupd(const int fd,
                  const unsigned flags,
                  const slotno slot,
                  apiBtvar *vard)
```

The function gbatch_varupd is used to update the details of a variable

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is 0 or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list if possible.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

`vard` is a pointer to a descriptor which contains the new details for the variable. The definition of the variable structure is given on page 22 onwards.

12.4.6.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.7 gbatch_varchcomm

```
int gbatch_varchcomm(const int fd,
                     const unsigned flags,
                     const slotno_t slot,
                     const char *newcomment)
```

The function gbatch_varchcomm is used to change the comment which is associated with a variable

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is 0 or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list if possible.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

`newcomment` is the proposed new comment for the variable.

12.4.7.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.8 gbatch_varchown

```
int gbatch_varchown(const int fd,
                    const unsigned flags,
                    const slotno_t slot,
                    const char *newowner)
```

The function gbatch_varchown is used to change the ownership of a variable to new user.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is 0 or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list if possible.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

`newname` is the name of the user who is to gain ownership of the variable.

12.4.8.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.9 gbatch_varchgrp

```
int gbatch_varchgrp(const int fd,
                    const unsigned flags,
                    const slotno_t slot,
                    const char *newgroup)
```

The function gbatch_varchgrp is used to attempt to change the group ownership of a variable.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is 0 or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list if possible.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

12.4.9.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.10 gbatch_varchmod

```
int gbatch_varchmod(const int fd,
                    const unsigned flags,
                    const slotno_t slot,
                    const Btmode *newmode)
```

The function gbatch_varchmod is used to change the permissions associated with a variable.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is 0 or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list if possible.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

`newmode` is a pointer to the structure which contains all the new mode details. The definition of the variable structure is given on page 22 onwards.

12.4.10.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.11 gbatch_varrename

```
int gbatch_varrename(const int fd,
                     const unsigned flags,
                     const slotno_t slot,
                     const char *newname)
```

The function gbatch_varrename is used to change the name of a variable

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is zero or `GBATCH_FLAG_IGNORESEQ` to ignore recent changes to the variable list.

`slot` is the slot number corresponding to the variable as returned by gbatch_varlist or gbatch_varfindslot.

`newname` is the proposed new name for the variable.

12.4.11.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.12 gbatch_varmon

```
int gbatch_varmon(const int fd, void (*fn)(const int))
```

Note that this routine is not available in the Windows version, please see the section on gbatch_setmon on page 45 which covers both jobs and variables.

The function gbatch_varmon is used to set a function to monitor changes to the variables list.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`fn` is a pointer to a function which will handle the changes to the list or NULL, which cancels any previous call. This function will be called with fd as an argument when any change is noted. The API does not note which variables the user is interested in, so any changes to variables may provoke a call to this function.

12.4.12.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.4.12.2 Example

```
void note_mod(const int fd)
{
    var_changes++;
}

...

gbatch_varmon(fd, note_mod);
if (var_changes)  {
    var_changes = 0;
     ...
/* Re-read list etc */
```

```
    ...
}

gbatch_varmon(fd, NULL);
```

12.5 Command Interpreters

12.5.1 gbatch_ciread

```
int ciread(const int fd,
           const unsigned flags,
           int *numcis,
           Cmdint **cilist)
```

The function gbatch_ciread is used to read the list of command interpreters from the given server. This may be invoked by any user, no special permission is required.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is currently unsused, but is reserved for future use. Set it to zero.

`numcis` is a pointer to an integer which upon return will contain the number of command interpreter structures returned in cilist. (This might exceed the number of actual command interpeters if some have been deleted).

`cilist` is a pointer to which a vector of command interpreter structures will be assigned by this routine. The user should not attempt to free the memory used by this structure as it is owned by the API. The list returned may possibly have “holes” in it where previously-created command interpreters have been deleted. These holes can be identified by having a null `ci_name` field.

The definition of the command interpreter structure is given on page 27 onwards.

The index number of each element in the vector is the number which should be used as the third argument in gbatch_cidel and gbatch_ciupd calls.

12.5.1.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.5.2 gbatch_ciadd

```
int gbatch_ciadd(const int fd,
                 const unsigned flags,
                 const Cmdint *newci,
                 unsigned *indx)
```

The function gbatch_ciadd is used to create a new command interpreter on a **GNUBatch** server. The invoking user must have special create permission or the call will be rejected.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is currently unused, but reserved for future use. Set it to zero.

`newci` is a pointer to a structure containing the new command interpreter details.

`indx` is a pointer to an unsigned location into which the index number of the new command interpreter is placed.

The definition of the command interpreter structure is given on page 27 onwards.

12.5.2.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.5.3 gbatch_cidel

```
int gbatch_cidel(const int fd, const unsigned flags, const unsigned indx)
```

The function gbatch_cidel is used to delete a command interpreter from a **GNUBatch** server. The invoking user must have *special create* permission or the call will be rejected.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is currently unused, but is reserved for future extensions. Set it to zero.

`indx` is the number of the command interpreter to be deleted.

12.5.3.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.5.3.2 Notes

The standard shell entry, entry zero, cannot be deleted and attempts to do so will always return an error code (`GBATCH_BAD_CI`).

There are few checks and interlocks on command interpreter lists, which are assumed to be likely to be changed sparingly. The user should satisfy him or herself that there are no jobs likely to use the command interpreter about to be deleted before invoking this operation.

12.5.4 gbatch_ciupd

```
int gbatch_ciupd(const int fd,
                 const unsigned flags,
                 const int indx,
                 const Cmdint *newci)
```

The function gbatch_ciupd is used to update the details of a command interpreter on a **GNUBatch** server. The invoking user must have *special create* permission or the call will be rejected.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`flags` is currently unused, but is reserved for future extensions. Set it to zero.

`indx` is the number of the command interpreter to be updated (see gbatch_ciread).

`newci` is a pointer to a structure containing the new command interpreter details.

The definition of the command interpreter structure is given on page 27 onwards.

12.5.4.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.6 User permissions

12.6.1 gbatch_getbtd

```
int gbatch_getbtd(const int fd, apiBtdef *defs)
```

The function gbatch_getbtd is used to read the default user profile parameters for the given host.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`defs` is a pointer to a structure which will on successful return, contain the default user details. The definition of the default user profile structure is given on page 26.

12.6.1.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.6.2 gbatch_getbtu

```
int gbatch_getbtu(const int fd,
                  char *username,
                  char *groupname,
```

```
                apiBtuser *ustr)
```

The function gbatch_getbtu is used to read the user profile of a specific user. Only a user with *read admin file privilege* will be able to read the profiles of other users.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`username` is the name of a valid user on the server.

`groupname` will be assigned with the default group name on the server.

`ustr` is a pointer to a structure which will on successful return, contain the profile of the specific user. The definition of the user profile structure is given on page 24.

12.6.2.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.6.3 gbatch_putbtd

```
int gbatch_putbtd(const int fd, const apiBtdef *defs)
```

The function gbatch_putbtd is used to update the default user profile parameters for the given host. It may only be invoked by a user with write admin file privilege.

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`defs` is a pointer to a structure containing the new default user profile. The definition of the default user profile structure is given on page 26.

12.6.3.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

12.6.4 gbatch_putbtu

```
int gbatch_putbtu(const int fd,
                const char *username,
                const apiBtuser *ustr)
```

The function gbatch_putbtu is used to update a user's profile parameters for the given host. It may only be invoked by a user with *write admin file* privilege, unless the user just wants to change his or her default modes and has change default modes privilege. (The privileges are those applying prior to the operation).

`fd` is a file descriptor which was previously returned by a successful call to gbatch_open or equivalent.

`username` is the name of the user, whose details are being updated.

`ustr` is a pointer to a structure which contains the new user profile. The definition of the user profile structure is given on page 24.

12.6.4.1 Return values

The function returns 0 if successful otherwise one of the error codes listed on page 28 onwards.

Chapter 13

Example API program

The following program is an example of the use of the Unix API to provide a simple read-only screen displaying some jobs and variables simultaneously.

```
#include <sys/types.h>
#include <curses.h>
#include <time.h>
#include <signal.h>
#include <gbatch.h>

#define  MAXJOBSATONCE    10
#define  MAXVARSATONCE    7
#define  V_START          (MAXJOBSATONCE+2)

int  jslotnums = -1,      /* Number we are monitoring */
     jslotlast = -1,      /* Last number of jobs on list */
     vslotnums = -1,      /* Number of variables we are monitoring */
     vslotlast = -1,      /* Last number of vars on list */
     jobchanges = 0,      /* Changes noted in jobs */
     varchanges = 0,      /* Changes noted in variables */
     vnamecnt,            /* Number of variables we asked about */
     apifd;               /* "File descriptor" for api */

slotno_t jslotno[MAXJOBSATONCE],  /* Slot numbers of jobs being monitored */
         vslotno[MAXVARSATONCE];  /* Slot numbers of vars being monitored */

char    **vnames,        /* Names of variables we want */
        *hostname,       /* Machine we want to talk to */
        *queuename;      /* Queue name */

static  char    *statenames[] = {
     "",
     "Done",
     "Error",
     "Aborted",
     "Cancelled",
     "Strt1",
     "Strt2",
```

```
     "Running",
     "Finished"
};

/*  Invoked in the event of a signal  */

void    quitit()
{
    gbatch_close(apifd);
    endwin();
    exit(0);
}

/*  Fill up the screen according to jobs and variables.  */

void    fillscreen()
{
    intcnt, row;
    time_t  now = time((time_t *) 0);

    /*  Clear the existing text on the screen  */

    erase();

    /* For each job....   */

    for  (cnt = 0; cnt < jslotnums;  cnt++)  {
        const  char*tit;
        char    tbuf[16];
        apiBtjobjob;

        /* Read the job, if it has disappeared, forget it */

        if  (gbatch_jobread(apifd, GBATCH_FLAG_IGNORESEQ, jslotno[cnt], &job)
< 0)
            continue;

        /* Extract title */

        tit = gbatch_gettitle(apifd, &job);

        /* If time applies, print time, or date if not in 24 hours */

        if  (job.h.bj_times.tc_istime)  {
            struct  tm  *tp = localtime(&job.h.bj_times.tc_nexttime);
            if  (job.h.bj_times.tc_nexttime < now ||
                 job.h.bj_times.tc_nexttime > now + (24L*60L*60L))
                sprintf(tbuf, "%.2d/%.2d", tp->tm_mday, tp->tm_mon+1);
            else
                sprintf(tbuf, "%.2d:%.2d", tp->tm_hour, tp->tm_min);
        }
        else
            tbuf[0] = '\0';
        mvprintw(cnt, 0, "%.7d %-16s %-5.5s %s", job.h.bj_job, tit, tbuf,
```

```
                statenames[job.h.bj_progress]);
    }

    row = V_START;

    for  (cnt = 0; cnt < vslotnums;  cnt++)  {
        apiBtvar var;
        if  (gbatch_varread(apifd, GBATCH_FLAG_IGNORESEQ, vslotno[cnt], &var)
< 0)
            continue;

        /*  Print variable name, value and comment string */

        if  (var.var_value.const_type == CON_LONG)
            mvprintw(row,
                    0, "%-15s %ld %s", var.var_name,
                    var.var_value.con_un.con_long, var.var_comment);
        else
            mvprintw(row,
                    0, "%-15s %s %s", var.var_name,
                    var.var_value.con_un.con_string, var.var_comment);
        row++;
    }

    move(LINES-1,COLS-1);
    refresh();
}

void    readjlist()
{
    intnjs, cnt;
    slotno_t*jsls;

    jobchanges = 0;

    /*  Read the list of jobs in the queue.   */

    if  (gbatch_joblist(apifd, GBATCH_FLAG_IGNORESEQ, &njs, &jsls) < 0)
        return;

    /*  If the number of jobs is the same as last time,
        we can assume that no new ones have been created.  */

    if  (njs == jslotlast)
        return;

    jslotlast = njs;

    /* If we have more than we can fit on the screen,
       skip the rest */

    if  (njs > MAXJOBSATONCE)
        njs = MAXJOBSATONCE;

    jslotnums = njs;
```

```
    for  (cnt = 0;  cnt < njs;  cnt++)
        jslotno[cnt] = jsls[cnt];
}

void    readvlist()
{
    int  nvs, cnt, cnt2;
    slotno_t *vsls;

    varchanges = 0;

    /*  Read the list of variables available to us.  */

    if  (gbatch_varlist(apifd, GBATCH_FLAG_IGNORESEQ, &nvs, &vsls) < 0)
        return;

    /*  If the number of variables is the same, we can assume that
        we haven't created or deleted any.  */

    if  (nvs == vslotlast)
        return;

    /*  Reset the pointer of slot numbers we are interested in */

    vslotlast = nvs;
    vslotnums = 0;

    /*  Look through the list of variables we got back for the
        ones we are interested in.  */

    for  (cnt = 0; cnt < nvs;  cnt++)  {
        apiBtvar var;

        /* Read the variable */

        if  (gbatch_varread(apifd, GBATCH_FLAG_IGNORESEQ, vsls[cnt], &var) <
0)
            continue;

        /* Look through the list of names.
           If we find it, remember the slot number.  */

        for  (cnt2 = 0; cnt2 < vnamecnt;  cnt2++)
            if  (strcmp(vnames[cnt2], var.var_name) == 0)  {
                vslotno[vslotnums++] = vsls[cnt];
                break;
            }
    }
}

void    catchjob(const int fd)
{
    jobchanges++;
}
```

```
void    catchvar(const int fd)
{
    varchanges++;
}

void    process()
{
    apifd = gbatch_open(hostname, (const char *) 0);
    if  (apifd < 0)  {
        fprintf(stderr, "Cannot open API\n");
        exit(250);
    }

    gbatch_setqueue(apifd, queuename);

    initscr();
    noecho();
    nonl();

    readjlist();
    readvlist();
    fillscreen();

    /* Let the user abort the program with quit or interrupt */

    sigset(SIGINT, quitit);
    sigset(SIGQUIT, quitit);

    /*  Get signals to detect changes to jobs and variables */

    gbatch_jobmon(apifd, catchjob);
    gbatch_varmon(apifd, catchvar);

    for  (;;)  {

        /*  Any changes to jobs or variables cause
            a reread and refill.  */

        while  (jobchanges || varchanges)  {
            if  (jobchanges)
                 readjlist();
            if (varchanges)
                 readvlist();
            fillscreen();
        }

        /* Wait for a signal */

        pause();
    }
}

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

```
    if  (argc < 3)  {
        fprintf(stderr,
                "Usage:  %s hostname queuename var1 var2 ....\n", argv[0]);
        exit(1);
    }

    hostname = argv[1];
    queuename = argv[2];

    vnamecnt = argc - 3;

    if  (vnamecnt > MAXVARSATONCE)  {
        fprintf(stderr, "Sorry to many variables at once\n");
        exit(2);
    }

    vnames = &argv[3];

    process();       /* Does Not Return */
    return  0;       /* Silence compilers */
}
```