It comes in the night and sucks the essence from your computers.

Kern Sibbald

April 26, 2005
This is the Bacula Developer's Guide

Contents

List of Figures

List of Tables

Bacula Developer Notes

General

This document is intended mostly for developers and describes the the general framework of making Bacula source changes.

Contributions

Contributions from programmers are broken into two groups. The first are contributions that are aids and not essential to Bacula. In general, these will be scripts or will go into and examples or contributions directory.

The second class of contributions are those which will be integrated with Bacula and become an essential part. Within this class of contributions, there are two hurdles to surmount. One is getting your patch accepted, and two is dealing with copyright issues. The rest of this document describes some of the requirements for such code.

Patches

Subject to the copyright assignment described below, your patches should be sent in **diff -u** format relative to the current contents of the Source Forge CVS, which is the easiest for me to understand. If you plan on doing significant development work over a period of time, after having your first patch reviewed and approved, you will be eligible for having CVS access so that you can commit your changes directly to the CVS repository. To do so, you will need a userid on Source Forge.

Copyrights

To avoid future problems concerning changing licensing or copyrights, all code contributions more than a hand full of lines must be in the Public Domain or have the copyright assigned to Kern Sibbald as in the current code. Note, prior to November 2004, the code was copyrighted by Kern Sibbald and John Walker.

Your name should be clearly indicated as the author of the code, and you must be extremely careful not to violate any copyrights or use other

people's code without acknowledging it. The purpose of this requirement is to avoid future copyright, patent, or intellectual property problems. To understand on possible source of future problems, please examine the difficulties Mozilla is (was?) having finding previous contributors at http://www.mozilla.org/MPL/missing.html. The other important issue is to avoid copyright, patent, or intellectual property violations as are currently (May 2003) being claimed by SCO against IBM.

Although the copyright will be held by Kern, each developer is expected to indicate that he wrote and/or modified a particular module (or file) and any other sources. The copyright assignment may seem a bit unusual, but in reality, it is not. Most large projects require this. In fact, the paperwork associated with making contributions to the Free Software Foundation, was for me unsurmountable.

If you have any doubts about this, please don't hesitate to ask. Our (John and my) track records with Autodesk are easily available; early programmers/founders/contributors and later employees had substantial shares of the company, and no one founder had a controlling part of the company. Even though Microsoft created many millionaires among early employees, the politics of Autodesk (during our time at the helm) is in stark contrast to Microsoft where the majority of the company is still tightly held among a few.

Items not needing a copyright assignment are: most small changes, enhancements, or bug fixes of 5-10 lines of code, and documentation.

Copyright Assignment

Since this is not a commercial enterprise, and I prefer to believe in everyone's good faith, developers can assign the copyright by explicitly acknowledging that they do so in their first submission. This is sufficient if the developer is independent, or an employee of a not-for-profit organization or a university. Any developer that wants to contribute and is employed by a company must get a copyright assignment from his employer. This is to avoid misunderstandings between the employee, the company, and the Bacula project.

Corporate Assignment Statement

The following statement must be filled out by the employer, signed, and mailed to my address (please ask me for my address and I will email it – I'd prefer not to include it here).

```
Copyright release and transfer statement.
   <On company letter head>

   To: Kern Sibbald
   Concerning: Copyright release and transfer

   <Company, Inc> is hereby agrees that <names-of-developers> and
   other employees of <Company, Inc> are authorized to develop
   code for and contribute code to the Bacula project for an
   undetermined period of time.  The copyright as well as all
   other rights to and interests in such contributed code are
   hereby transferred to Kern Sibbald.

   Signed in <City, Country> on <Date>:

   <Name of Person>, <Position in Company, Inc>
```

This release/transfer statement must be sent to: Kern Sibbald Address-to-be-given

Developing Bacula

Typically the simplest way to develop Bacula is to open one xterm window pointing to the source directory you wish to update; a second xterm window at the top source directory level, and a third xterm window at the bacula directory <top>/src/bacula. After making source changes in one of the directories, in the top source directory xterm, build the source, and start the daemons by entering:

make and

./startit then in the enter:

./console or

./gnome-console to start the Console program. Enter any commands for testing. For example: run kernsverify full.

Note, the instructions here to use **./startit** are different from using a production system where the administrator starts Bacula by entering **./bacula start**. This difference allows a development version of **Bacula** to be run on a computer at the same time that a production system is running. The **./startit** strip starts **Bacula** using a different set of configuration files, and thus permits avoiding conflicts with any production system.

To make additional source changes, exit from the Console program, and in

the top source directory, stop the daemons by entering:

./stopit then repeat the process.

Debugging

Probably the first thing to do is to turn on debug output.

A good place to start is with a debug level of 20 as in **./startit -d20**. The startit command starts all the daemons with the same debug level. Alternatively, you can start the appropriate daemon with the debug level you want. If you really need more info, a debug level of 60 is not bad, and for just about everything a level of 200.

Using a Debugger

If you have a serious problem such as a segmentation fault, it can usually be found quickly using a good multiple thread debugger such as **gdb**. For example, suppose you get a segmentation violation in **bacula-dir**. You might use the following to find the problem:

<start the Storage and File daemons> cd dird gdb ./bacula-dir run -f -s -c ./dird.conf <it dies with a segmentation fault> where The **-f** option is specified on the **run** command to inhibit **dird** from going into the background. You may also want to add the **-s** option to the run command to disable signals which can potentially interfere with the debugging.

As an alternative to using the debugger, each **Bacula** daemon has a built in back trace feature when a serious error is encountered. It calls the debugger on itself, produces a back trace, and emails the report to the developer. For more details on this, please see the chapter in the main Bacula manual entitled "What To Do When Bacula Crashes (Kaboom)".

Memory Leaks

Because Bacula runs routinely and unattended on client and server machines, it may run for a long time. As a consequence, from the very beginning, Bacula uses SmartAlloc to ensure that there are no memory leaks. To make detection of memory leaks effective, all Bacula code that dynamically allocates memory MUST have a way to release it. In general when the memory is no longer needed, it should be immediately released, but in

some cases, the memory will be held during the entire time that Bacula is executing. In that case, there MUST be a routine that can be called at termination time that releases the memory. In this way, we will be able to detect memory leaks. Be sure to immediately correct any and all memory leaks that are printed at the termination of the daemons.

Special Files

Kern uses files named 1, 2, ... 9 with any extension as scratch files. Thus any files with these names are subject to being rudely deleted at any time.

When Implementing Incomplete Code

Please identify all incomplete code with a comment that contains ***FIXME***, where there are three asterisks (*) before and after the word FIXME (in capitals) and no intervening spaces. This is important as it allows new programmers to easily recognize where things are partially implemented.

Bacula Source File Structure

The distribution generally comes as a tar file of the form **bacula.x.y.z.tar.gz** where x, y, and z are the version, release, and update numbers respectively.

Once you detar this file, you will have a directory structure as follows:

```
|
|- depkgs
   |- mtx              (autochanger control program + tape drive info)
   |- sqlite           (SQLite database program)
|- depkgs-win32
   |- pthreads         (Native win32 pthreads library -- dll)
   |- zlib             (Native win32 zlib library)
   |- wx               (wxWidgets source code)
|- bacula              (main source directory containing configuration
   |                    and installation files)
   |- autoconf         (automatic configuration files, not normally used
   |                    by users)
   |- doc              (documentation directory)
      |- home-page     (Bacula's home page source)
      |- html-manual   (html document directory)
      |- techlogs      (Technical development notes);
```

```
   |- intl             (programs used to translate)
   |- platforms        (OS specific installation files)
      |- redhat        (Red Hat installation)
      |- solaris       (Sun installation)
      |- freebsd       (FreeBSD installation)
      |- irix          (Irix installation -- not tested)
      |- unknown       (Default if system not identified)
   |- po               (translations of source strings)
   |- src              (source directory; contains global header files)
      |- cats          (SQL catalog database interface directory)
      |- console       (bacula user agent directory)
      |- dird          (Director daemon)
      |- filed         (Unix File daemon)
         |- win32      (Win32 files to make bacula-fd be a service)
      |- findlib       (Unix file find library for File daemon)
      |- gnome-console (GNOME version of console program)
      |- lib           (General Bacula library)
      |- stored        (Storage daemon)
      |- tconsole      (Tcl/tk console program -- not yet working)
      |- testprogs     (test programs -- normally only in Kern's tree)
      |- tools         (Various tool programs)
      |- win32         (Native Win32 File daemon)
         |- baculafd   (Visual Studio project file)
         |- compat     (compatibility interface library)
         |- filed      (links to src/filed)
         |- findlib    (links to src/findlib)
         |- lib        (links to src/lib)
         |- console    (beginning of native console program)
         |- wx-console (wxWidget console Win32 specific parts)
     |- wx-console     (wxWidgets console main source program)
|- regress             (Regression scripts)
   |- bin              (temporary directory to hold Bacula installed binaries)
   |- build            (temporary directory to hold Bacula source)
   |- scripts          (scripts and .conf files)
   |- tests            (test scripts)
   |- tmp              (temporary directory for temp files)
```

Header Files

Please carefully follow the scheme defined below as it permits in general only two header file includes per C file, and thus vastly simplifies programming. With a large complex project like Bacula, it isn't always easy to ensure that the right headers are invoked in the right order (there are a few kludges to make this happen – i.e. in a few include files because of the chicken and egg problem, certain references to typedefs had to be replaced with **void**).

Every file should include **bacula.h**. It pulls in just about everything, with very few exceptions. If you have system dependent ifdefing, please do it in **baconfig.h**. The version number and date are kept in **version.h**.

Each of the subdirectories (console, cats, dird, filed, findlib, lib, stored, ...) contains a single directory dependent include file generally the name of the directory, which should be included just after the include of **bacula.h**. This file (for example, for the dird directory, it is **dird.h**) contains either definitions of things generally needed in this directory, or it includes the appropriate header files. It always includes **protos.h**. See below.

Each subdirectory contains a header file named **protos.h**, which contains the prototypes for subroutines exported by files in that directory. **protos.h** is always included by the main directory dependent include file.

Programming Standards

For the most part, all code should be written in C unless there is a burning reason to use C++, and then only the simplest C++ constructs will be used. Note, Bacula is slowly evolving to use more and more C++.

Code should have some documentation – not a lot, but enough so that I can understand it. Look at the current code, and you will see that I document more than most, but am definitely not a fanatic.

I prefer simple linear code where possible. Gotos are strongly discouraged except for handling an error to either bail out or to retry some code, and such use of gotos can vastly simplify the program.

Remember this is a C program that is migrating to a **tiny** subset of C++, so be conservative in your use of C++ features.

Do Not Use

- STL – it is totally incomprehensible.

Avoid if Possible

- Returning a malloc'ed buffer from a subroutine – someone will forget to release it.
- Using reference variables – it allows subroutines to create side effects.
- Heap allocation (malloc) unless needed – it is expensive.
- Templates – they can create portability problems.

- Fancy or tricky C or C++ code, unless you give a good explanation of why you used it.
- Too much inheritance – it can complicate the code, and make reading it difficult (unless you are in love with colons)

Do Use Whenever Possible

- Locking and unlocking within a single subroutine.
- Malloc and free within a single subroutine.
- Comments and global explanations on what your code or algorithm does.

Indenting Standards

I cannot stand code indented 8 columns at a time. This makes the code unreadable. Even 4 at a time uses a lot of space, so I have adopted indenting 3 spaces at every level. Note, indention is the visual appearance of the source on the page, while tabbing is replacing a series of up to 8 spaces from a tab character.

The closest set of parameters for the Linux **indent** program that will produce reasonably indented code are:

```
-nbad -bap -bbo -nbc -br -brs -c36 -cd36 -ncdb -ce -ci3 -cli0
-cp36 -d0 -di1 -ndj -nfc1 -nfca -hnl -i3 -ip0 -l85 -lp -npcs
-nprs -npsl -saf -sai -saw -nsob -nss -nbc -ncs -nbfda
```

You can put the above in your .indent.pro file, and then just invoke indent on your file. However, be warned. This does not produce perfect indenting, and it will mess up C++ class statements pretty badly.

Braces are required in all if statements (missing in some very old code). To avoid generating too many lines, the first brace appears on the first line (e.g. of an if), and the closing brace is on a line by itself. E.g.

```
if (abc) {
   some_code;
}
```

Just follow the convention in the code. Originally I indented case clauses under a switch(), but now I prefer non-indented cases.

```
switch (code) {
case 'A':
   do something
   break;
case 'B':
   again();
   break;
default:
   break;
}
```

Avoid using // style comments except for temporary code or turning off debug code. Standard C comments are preferred (this also keeps the code closer to C).

Attempt to keep all lines less than 85 characters long so that the whole line of code is readable at one time. This is not a rigid requirement.

Always put a brief description at the top of any new file created describing what it does and including your name and the date it was first written. Please don't forget any Copyrights and acknowledgments if it isn't 100% your code. Also, include the Bacula copyright notice that is in **src/c**.

In general you should have two includes at the top of the an include for the particular directory the code is in, for includes are needed, but this should be rare.

In general (except for self-contained packages), prototypes should all be put in **protos.h** in each directory.

Always put space around assignment and comparison operators.

```
a = 1;
if (b >= 2) {
  cleanup();
}
```

but your can compress things in a **for** statement:

```
for (i=0; i < del.num_ids; i++) {
 ...
```

Don't overuse the inline if (?:). A full **if** is preferred, except in a print statement, e.g.:

```
if (ua->verbose \&& del.num_del != 0) {
   bsendmsg(ua, _("Pruned %d %s on Volume %s from catalog.\n"), del.num_del,
      del.num_del == 1 ? "Job" : "Jobs", mr->VolumeName);
}
```

Leave a certain amount of debug code (Dmsg) in code you submit, so that future problems can be identified. This is particularly true for complicated code likely to break. However, try to keep the debug code to a minimum to avoid bloating the program and above all to keep the code readable.

Please keep the same style in all new code you develop. If you include code previously written, you have the option of leaving it with the old indenting or re-indenting it. If the old code is indented with 8 spaces, then please re-indent it to Bacula standards.

If you are using **vim**, simply set your tabstop to 8 and your shiftwidth to 3.

Tabbing

Tabbing (inserting the tab character in place of spaces) is as normal on all Unix systems – a tab is converted space up to the next column multiple of 8. My editor converts strings of spaces to tabs automatically – this results in significant compression of the files. Thus, you can remove tabs by replacing them with spaces if you wish. Please don't confuse tabbing (use of tab characters) with indenting (visual alignment of the code).

Don'ts

Please don't use:

```
strcpy()
strcat()
strncpy()
strncat();
sprintf()
snprintf()
```

They are system dependent and un-safe. These should be replaced by the Bacula safe equivalents:

```
char *bstrncpy(char *dest, char *source, int dest_size);
char *bstrncat(char *dest, char *source, int dest_size);
```

```
int bsnprintf(char *buf, int32_t buf_len, const char *fmt, ...);
int bvsnprintf(char *str, int32_t size, const char  *format, va_list ap);
```

See src/lib/bsys.c for more details on these routines.

Don't use the **%lld** or the **%q** printf format editing types to edit 64 bit integers – they are not portable. Instead, use **%s** with **edit_uint64()**. For example:

```
char buf[100];
uint64_t num = something;
char ed1[50];
bsnprintf(buf, sizeof(buf), "Num=%s\n", edit_uint64(num, ed1));
```

The edit buffer **ed1** must be at least 27 bytes long to avoid overflow. See src/lib/edit.c for more details. If you look at the code, don't start screaming that I use **lld**. I actually use subtle trick taught to me by John Walker. The **lld** that appears in the editing routine is actually **#define** to a what is needed on your OS (usually "lld" or "q") and is defined in autoconf/configure.in for each OS. C string concatenation causes the appropriate string to be concatenated to the "%".

Also please don't use the STL or Templates or any complicated C++ code.

Message Classes

Currently, there are four classes of messages: Debug, Error, Job, and Memory.

Debug Messages

Debug messages are designed to be turned on at a specified debug level and are always sent to STDOUT. There are designed to only be used in the development debug process. They are coded as:

DmsgN(level, message, arg1, ...) where the N is a number indicating how many arguments are to be substituted into the message (i.e. it is a count of the number arguments you have in your message – generally the number of percent signs (%)). **level** is the debug level at which you wish the message to be printed. message is the debug message to be printed, and arg1, ... are the arguments to be substituted. Since not all compilers support #defines with varargs, you must explicitly specify how many arguments you have.

When the debug message is printed, it will automatically be prefixed by the name of the daemon which is running, the filename where the Dmsg is, and the line number within the file.

Some actual examples are:

Dmsg2(20, "MD5len=%d MD5=%s\n", strlen(buf), buf);

Dmsg1(9, "Created client %s record\n", client-¿hdr.name);

Error Messages

Error messages are messages that are related to the daemon as a whole rather than a particular job. For example, an out of memory condition my generate an error message. They are coded as:

EmsgN(error-code, level, message, arg1, ...) As with debug messages, you must explicitly code the of arguments to be substituted in the message. error-code indicates the severity or class of error, and it may be one of the following:

M_ABORT	Causes the daemon to immediately abort. This should be used only in extreme cases. It attempts to produce a traceback.
M_ERROR_TERM	Causes the daemon to immediately terminate. This should be used only in extreme cases. It does not produce a traceback.
M_FATAL	Causes the daemon to terminate the current job, but the daemon keeps running
M_ERROR	Reports the error. The daemon and the job continue running
M_WARNING	Reports an warning message. The daemon and the job continue running
M_INFO	Reports an informational message.

There are other error message classes, but they are in a state of being redesigned or deprecated, so please do not use them. Some actual examples are:

Emsg1(M_ABORT, 0, "Cannot create message thread: %s\n", strerror(status));

Emsg3(M_WARNING, 0, "Connect to File daemon %s at %s:%d failed.

Retrying ...\n", client->hdr.name, client->address, client->port);

Emsg3(M_FATAL, 0, "bdird<filed: bad response from Filed to %s command: %d %s\n", cmd, n, strerror(errno));

Job Messages

Job messages are messages that pertain to a particular job such as a file that could not be saved, or the number of files and bytes that were saved.

Memory Messages

Memory messages are messages that are edited into a memory buffer. Generally they are used in low level routines such as the low level device file dev.c in the Storage daemon or in the low level Catalog routines. These routines do not generally have access to the Job Control Record and so they return error messages reformatted in a memory buffer. Mmsg() is the way to do this.

Daemon Protocol

General

This document describes the protocols used between the various daemons. As Bacula has developed, it has become quite out of date. The general idea still holds true, but the details of the fields for each command, and indeed the commands themselves have changed considerably.

It is intended to be a technical discussion of the general daemon protocols and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bacula**.

Low Level Network Protocol

At the lowest level, the network protocol is handled by **BSOCK** packets which contain a lot of information about the status of the network connection: who is at the other end, etc. Each basic **Bacula** network read or write actually consists of two low level network read/writes. The first write always sends four bytes of data in machine independent byte order. If data is to follow, the first four bytes are a positive non-zero integer indicating the length of the data that follow in the subsequent write. If the four byte integer is zero or negative, it indicates a special request, a sort of network signaling capability. In this case, no data packet will follow. The low level BSOCK routines expect that only a single thread is accessing the socket at a time. It is advised that multiple threads do not read/write the same socket. If you must do this, you must provide some sort of locking mechanism. I would not be appropriate for efficiency reasons to make every call to the BSOCK routines lock and unlock the packet.

General Daemon Protocol

In general, all the daemons follow the following global rules. There may be exceptions depending on the specific case. Normally, one daemon will be sending commands to another daemon (specifically, the Director to the Storage daemon and the Director to the File daemon).

- Commands are always ASCII commands that are upper/lower case dependent as well as space sensitive.

- All binary data is converted into ASCII (either with printf statements or using base64 encoding).
- All responses to commands sent are always prefixed with a return numeric code where codes in the 1000's are reserved for the Director, the 2000's are reserved for the File daemon, and the 3000's are reserved for the Storage daemon.
- Any response that is not prefixed with a numeric code is a command (or subcommand if you like) coming from the other end. For example, while the Director is corresponding with the Storage daemon, the Storage daemon can request Catalog services from the Director. This convention permits each side to send commands to the other daemon while simultaneously responding to commands.
- Any response that is of zero length, depending on the context, either terminates the data stream being sent or terminates command mode prior to closing the connection.
- Any response that is of negative length is a special sign that normally requires a response. For example, during data transfer from the File daemon to the Storage daemon, normally the File daemon sends continuously without intervening reads. However, periodically, the File daemon will send a packet of length -1 indicating that the current data stream is complete and that the Storage daemon should respond to the packet with an OK, ABORT JOB, PAUSE, etc. This permits the File daemon to efficiently send data while at the same time occasionally "polling" the Storage daemon for his status or any special requests.

  Currently, these negative lengths are specific to the daemon, but shortly, the range 0 to -999 will be standard daemon wide signals, while -1000 to -1999 will be for Director user, -2000 to -2999 for the File daemon, and -3000 to -3999 for the Storage daemon.

The Protocol Used Between the Director and the Storage Daemon

Before sending commands to the File daemon, the Director opens a Message channel with the Storage daemon, identifies itself and presents its password. If the password check is OK, the Storage daemon accepts the Director. The Director then passes the Storage daemon, the JobId to be run as well as the File daemon authorization (append, read all, or read for a specific session). The Storage daemon will then pass back to the Director a enabling key for this JobId that must be presented by the File daemon when opening the

job. Until this process is complete, the Storage daemon is not available for use by File daemons.

```
SD: listens
DR: makes connection
DR: Hello <Director-name> calling <password>
SD: 3000 OK Hello
DR: JobId=nnn Allow=(append, read) Session=(*, SessionId)
                    (Session not implemented yet)
SD: 3000 OK Job Authorization=<password>
DR: use device=<device-name> media_type=<media-type>
        pool_name=<pool-name> pool_type=<pool_type>
SD: 3000 OK use device
```

For the Director to be authorized, the <Director-name> and the <password> must match the values in one of the Storage daemon's Director resources (there may be several Directors that can access a single Storage daemon).

The Protocol Used Between the Director and the File Daemon

A typical conversation might look like the following:

```
FD: listens
DR: makes connection
DR: Hello <Director-name> calling <password>
FD: 2000 OK Hello
DR: JobId=nnn Authorization=<password>
FD: 2000 OK Job
DR: storage address = <Storage daemon address> port = <port-number>
          name = <DeviceName> mediatype = <MediaType>
FD: 2000 OK storage
DR: include
DR: <directory1>
DR: <directory2>
    ...
DR: Null packet
FD: 2000 OK include
DR: exclude
DR: <directory1>
DR: <directory2>
    ...
DR: Null packet
FD: 2000 OK exclude
DR: full
FD: 2000 OK full
DR: save
FD: 2000 OK save
```

```
FD: Attribute record for each file as sent to the
    Storage daemon (described above).
FD: Null packet
FD: <append close responses from Storage daemon>
    e.g.
    3000 OK Volumes = <number of volumes>
    3001 Volume = <volume-id> <start file> <start block>
         <end file> <end block> <volume session-id>
    3002 Volume data = <date/time of last write> <Number bytes written>
         <number errors>
    ... additional Volume / Volume data pairs for volumes 2 .. n
FD: Null packet
FD: close socket
```

The Save Protocol Between the File Daemon and the Storage Daemon

Once the Director has send a **save** command to the File daemon, the File daemon will contact the Storage daemon to begin the save.

In what follows: FD: refers to information set via the network from the File daemon to the Storage daemon, and SD: refers to information set from the Storage daemon to the File daemon.

Command and Control Information

Command and control information is exchanged in human readable ASCII commands.

```
FD: listens
SD: makes connection
FD: append open session = <JobId> [<password>]
SD: 3000 OK ticket = <number>
FD: append data <ticket-number>
SD: 3000 OK data address = <IPaddress> port = <port>
```

Data Information

The Data information consists of the file attributes and data to the Storage daemon. For the most part, the data information is sent one way: from the File daemon to the Storage daemon. This allows the File daemon to transfer information as fast as possible without a lot of handshaking and network overhead.

However, from time to time, the File daemon needs to do a sort of checkpoint of the situation to ensure that everything is going well with the Storage daemon. To do so, the File daemon sends a packet with a negative length indicating that he wishes the Storage daemon to respond by sending a packet of information to the File daemon. The File daemon then waits to receive a packet from the Storage daemon before continuing.

All data sent are in binary format except for the header packet, which is in ASCII. There are two packet types used data transfer mode: a header packet, the contents of which are known to the Storage daemon, and a data packet, the contents of which are never examined by the Storage daemon.

The first data packet to the Storage daemon will be an ASCII header packet consisting of the following data.

<File-Index> <Stream-Id> <Info> where <**File-Index**> is a sequential number beginning from one that increments with each file (or directory) sent.

where <**Stream-Id**> will be 1 for the Attributes record and 2 for uncompressed File data. 3 is reserved for the MD5 signature for the file.

where <**Info**> transmit information about the Stream to the Storage Daemon. It is a character string field where each character has a meaning. The only character currently defined is 0 (zero), which is simply a place holder (a no op). In the future, there may be codes indicating compressed data, encrypted data, etc.

Immediately following the header packet, the Storage daemon will expect any number of data packets. The series of data packets is terminated by a zero length packet, which indicates to the Storage daemon that the next packet will be another header packet. As previously mentioned, a negative length packet is a request for the Storage daemon to temporarily enter command mode and send a reply to the File daemon. Thus an actual conversation might contain the following exchanges:

```
FD: <1 1 0> (header packet)
FD: <data packet containing file-attributes>
FD: Null packet
FD: <1 2 0>
FD: <multiple data packets containing the file data>
FD: Packet length = -1
SD: 3000 OK
FD: <2 1 0>
FD: <data packet containing file-attributes>
FD: Null packet
FD: <2 2 0>
```

```
FD: <multiple data packets containing the file data>
FD: Null packet
FD: Null packet
FD: append end session <ticket-number>
SD: 3000 OK end
FD: append close session <ticket-number>
SD: 3000 OK Volumes = <number of volumes>
SD: 3001 Volume = <volumeid> <start file> <start block>
     <end file> <end block> <volume session-id>
SD: 3002 Volume data = <date/time of last write> <Number bytes written>
     <number errors>
SD: ... additional Volume / Volume data pairs for
     volumes 2 .. n
FD: close socket
```

The information returned to the File daemon by the Storage daemon in response to the **append close session** is transmit in turn to the Director.

Director Services Daemon

This chapter is intended to be a technical discussion of the Director services and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bacula**.

The **Bacula Director** services consist of the program that supervises all the backup and restore operations.

To be written ...

File Services Daemon

Please note, this section is somewhat out of date as the code has evolved significantly. The basic idea has not changed though.

This chapter is intended to be a technical discussion of the File daemon services and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bacula**.

The **Bacula File Services** consist of the programs that run on the system to be backed up and provide the interface between the Host File system and Bacula – in particular, the Director and the Storage services.

When time comes for a backup, the Director gets in touch with the File daemon on the client machine and hands it a set of "marching orders" which, if written in English, might be something like the following:

OK, **File daemon**, it's time for your daily incremental backup. I want you to get in touch with the Storage daemon on host archive.mysite.com and perform the following save operations with the designated options. You'll note that I've attached include and exclude lists and patterns you should apply when backing up the file system. As this is an incremental backup, you should save only files modified since the time you started your last backup which, as you may recall, was 2000-11-19-06:43:38. Please let me know when you're done and how it went. Thank you.

So, having been handed everything it needs to decide what to dump and where to store it, the File daemon doesn't need to have any further contact with the Director until the backup is complete providing there are no errors. If there are errors, the error messages will be delivered immediately to the Director. While the backup is proceeding, the File daemon will send the file coordinates and data for each file being backed up to the Storage daemon, which will in turn pass the file coordinates to the Director to put in the catalog.

During a **Verify** of the catalog, the situation is different, since the File daemon will have an exchange with the Director for each file, and will not contact the Storage daemon.

A **Restore** operation will be very similar to the **Backup** except that during the **Restore** the Storage daemon will not send storage coordinates to the Director since the Director presumably already has them. On the other hand, any error messages from either the Storage daemon or File daemon will normally be sent directly to the Directory (this, of course, depends on

how the Message resource is defined).

Commands Received from the Director for a Backup

To be written ...

Commands Received from the Director for a Restore

To be written ...

Storage Daemon Design

This chapter is intended to be a technical discussion of the Storage daemon services and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bacula**.

SD Design Introduction

The Bacula Storage daemon provides storage resources to a Bacula installation. An individual Storage daemon is associated with a physical permanent storage device (for example, a tape drive, CD writer, tape changer or jukebox, etc.), and may employ auxiliary storage resources (such as space on a hard disk file system) to increase performance and/or optimize use of the permanent storage medium.

Any number of storage daemons may be run on a given machine; each associated with an individual storage device connected to it, and BACULA operations may employ storage daemons on any number of hosts connected by a network, local or remote. The ability to employ remote storage daemons (with appropriate security measures) permits automatic off-site backup, possibly to publicly available backup repositories.

SD Development Outline

In order to provide a high performance backup and restore solution that scales to very large capacity devices and networks, the storage daemon must be able to extract as much performance from the storage device and network with which it interacts. In order to accomplish this, storage daemons will eventually have to sacrifice simplicity and painless portability in favor of techniques which improve performance. My goal in designing the storage daemon protocol and developing the initial prototype storage daemon is to provide for these additions in the future, while implementing an initial storage daemon which is very simple and portable to almost any POSIX-like environment. This original storage daemon (and its evolved descendants) can serve as a portable solution for non-demanding backup requirements (such as single servers of modest size, individual machines, or small local networks), while serving as the starting point for development of higher performance configurable derivatives which use techniques such as POSIX threads, shared memory, asynchronous I/O, buffering to high-speed intermediate media, and support for tape changers and jukeboxes.

SD Connections and Sessions

A client connects to a storage server by initiating a conventional TCP connection. The storage server accepts the connection unless its maximum number of connections has been reached or the specified host is not granted access to the storage server. Once a connection has been opened, the client may make any number of Query requests, and/or initiate (if permitted), one or more Append sessions (which transmit data to be stored by the storage daemon) and/or Read sessions (which retrieve data from the storage daemon).

Most requests and replies sent across the connection are simple ASCII strings, with status replies prefixed by a four digit status code for easier parsing. Binary data appear in blocks stored and retrieved from the storage. Any request may result in a single-line status reply of "`3201 Notification pending`", which indicates the client must send a "Query notification" request to retrieve one or more notifications posted to it. Once the notifications have been returned, the client may then resubmit the request which resulted in the 3201 status.

The following descriptions omit common error codes, yet to be defined, which can occur from most or many requests due to events like media errors, restarting of the storage daemon, etc. These details will be filled in, along with a comprehensive list of status codes along with which requests can produce them in an update to this document.

SD Append Requests

append open session = <JobId> [<Password>] A data append session is opened with the Job ID given by *JobId* with client password (if required) given by *Password*. If the session is successfully opened, a status of `3000 OK` is returned with a "`ticket =` *number*" reply used to identify subsequent messages in the session. If too many sessions are open, or a conflicting session (for example, a read in progress when simultaneous read and append sessions are not permitted), a status of "`3502 Volume busy`" is returned. If no volume is mounted, or the volume mounted cannot be appended to, a status of "`3503 Volume not mounted`" is returned.

append data = <ticket-number> If the append data is accepted, a status of `3000 OK data address =` <`IPaddress`> `port =` <`port`> is returned, where the `IPaddress` and `port` specify the IP address and port number of the data channel. Error status codes are `3504`

`Invalid ticket number` and `3505 Session aborted`, the latter of which indicates the entire append session has failed due to a daemon or media error.

Once the File daemon has established the connection to the data channel opened by the Storage daemon, it will transfer a header packet followed by any number of data packets. The header packet is of the form:

`<file-index> <stream-id> <info>`

The details are specified in the Daemon Protocol section of this document.

***append abort session = <ticket-number>** The open append session with ticket *ticket-number* is aborted; any blocks not yet written to permanent media are discarded. Subsequent attempts to append data to the session will receive an error status of `3505 Session aborted`.

append end session = <ticket-number> The open append session with ticket *ticket-number* is marked complete; no further blocks may be appended. The storage daemon will give priority to saving any buffered blocks from this session to permanent media as soon as possible.

append close session = <ticket-number> The append session with ticket *ticket* is closed. This message does not receive an `3000 OK` reply until all of the content of the session are stored on permanent media, at which time said reply is given, followed by a list of volumes, from first to last, which contain blocks from the session, along with the first and last file and block on each containing session data and the volume session key identifying data from that session in lines with the following format:

`Volume = <Volume-id> <start-file> <start-block> <end-file> <end-block> <volume-session-id>`where *Volume-id* is the volume label, *start-file* and *start-block* are the file and block containing the first data from that session on the volume, *end-file* and *end-block* are the file and block with the last data from the session on the volume and *volume-session-id* is the volume session ID for blocks from the session stored on that volume.

SD Read Requests

Read open session = <JobId> <Volume-id> <start-file> <start-block> <end-file> <end-b
where *Volume-id* is the volume label, *start-file* and *start-block* are

the file and block containing the first data from that session on the volume, *end-file* and *end-block* are the file and block with the last data from the session on the volume and *volume-session-id* is the volume session ID for blocks from the session stored on that volume.

If the session is successfully opened, a status of

`3100 OK Ticket =` *number*‘‘

is returned with a reply used to identify subsequent messages in the session. If too many sessions are open, or a conflicting session (for example, an append in progress when simultaneous read and append sessions are not permitted), a status of "`3502 Volume busy`" is returned. If no volume is mounted, or the volume mounted cannot be appended to, a status of "`3503 Volume not mounted`" is returned. If no block with the given volume session ID and the correct client ID number appears in the given first file and block for the volume, a status of "`3505 Session not found`" is returned.

Read data = <Ticket> > <Block> The specified Block of data from open read session with the specified Ticket number is returned, with a status of `3000 OK` followed by a "`Length =` *size*" line giving the length in bytes of the block data which immediately follows. Blocks must be retrieved in ascending order, but blocks may be skipped. If a block number greater than the largest stored on the volume is requested, a status of "`3201 End of volume`" is returned. If a block number greater than the largest in the file is requested, a status of "`3401 End of file`" is returned.

Read close session = <Ticket> The read session with Ticket number is closed. A read session may be closed at any time; you needn't read all its blocks before closing it.

by John Walker January 30th, MM

Storage Media Output Format

General

This document describes the media format written by the Storage daemon. The Storage daemon reads and writes in units of blocks. Blocks contain records. Each block has a block header followed by records, and each record has a record header followed by record data.

This chapter is intended to be a technical discussion of the Media Format and as such is not targeted at end users but rather at developers and system administrators that want or need to know more of the working details of **Bacula**.

Definitions

Block A block represents the primitive unit of information that the Storage daemon reads and writes to a physical device. Normally, for a tape device, it will be the same as a tape block. The Storage daemon always reads and writes blocks. A block consists of block header information followed by records. Clients of the Storage daemon (the File daemon) normally never see blocks. However, some of the Storage tools (bls, bscan, bextract, ...) may be use block header information. In older Bacula tape versions, a block could contain records (see record definition below) from multiple jobs. However, all blocks currently written by Bacula are block level BB02, and a given block contains records for only a single job. Different jobs simply have their own private blocks that are intermingled with the other blocks from other jobs on the Volume (previously the records were intermingled within the blocks). Having only records from a single job in any give block permitted moving the VolumeSessionId and VolumeSessionTime (see below) from each record heading to the Block header. This has two advantages: 1. a block can be quickly rejected based on the contents of the header without reading all the records. 2. because there is on the average more than one record per block, less data is written to the Volume for each job.

Record A record consists of a Record Header, which is managed by the Storage daemon and Record Data, which is the data received from the Client. A record is the primitive unit of information sent to and from the Storage daemon by the Client (File daemon) programs. The details are described below.

JobId A number assigned by the Director daemon for a particular job. This number will be unique for that particular Director (Catalog). The daemons use this number to keep track of individual jobs. Within the Storage daemon, the JobId may not be unique if several Directors are accessing the Storage daemon simultaneously.

Session A Session is a concept used in the Storage daemon corresponds one to one to a Job with the exception that each session is uniquely identified within the Storage daemon by a unique SessionId/SessionTime pair (see below).

VolSessionId A unique number assigned by the Storage daemon to a particular session (Job) it is having with a File daemon. This number by itself is not unique to the given Volume, but with the VolSessionTime, it is unique.

VolSessionTime A unique number assigned by the Storage daemon to a particular Storage daemon execution. It is actually the Unix time_t value of when the Storage daemon began execution cast to a 32 bit unsigned integer. The combination of the **VolSessionId** and the **VolSessionTime** for a given Storage daemon is guaranteed to be unique for each Job (or session).

FileIndex A sequential number beginning at one assigned by the File daemon to the files within a job that are sent to the Storage daemon for backup. The Storage daemon ensures that this number is greater than zero and sequential. Note, the Storage daemon uses negative FileIndexes to flag Session Start and End Labels as well as End of Volume Labels. Thus, the combination of VolSessionId, VolSessionTime, and FileIndex uniquely identifies the records for a single file written to a Volume.

Stream While writing the information for any particular file to the Volume, there can be any number of distinct pieces of information about that file, e.g. the attributes, the file data, ... The Stream indicates what piece of data it is, and it is an arbitrary number assigned by the File daemon to the parts (Unix attributes, Win32 attributes, data, compressed data, ...) of a file that are sent to the Storage daemon. The Storage daemon has no knowledge of the details of a Stream; it simply represents a numbered stream of bytes. The data for a given stream may be passed to the Storage daemon in single record, or in multiple records.

Block Header A block header consists of a block identification ("BB02"), a block length in bytes (typically 64,512) a checksum, and sequential block number. Each block starts with a Block Header and is followed

by Records. Current block headers also contain the VolSessionId and VolSessionTime for the records written to that block.

Record Header A record header contains the Volume Session Id, the Volume Session Time, the FileIndex, the Stream, and the size of the data record which follows. The Record Header is always immediately followed by a Data Record if the size given in the Header is greater than zero. Note, for Block headers of level BB02 (version 1.27 and later), the Record header as written to tape does not contain the Volume Session Id and the Volume Session Time as these two fields are stored in the BB02 Block header. The in-memory record header does have those fields for convenience.

Data Record A data record consists of a binary stream of bytes and is always preceded by a Record Header. The details of the meaning of the binary stream of bytes are unknown to the Storage daemon, but the Client programs (File daemon) defines and thus knows the details of each record type.

Volume Label A label placed by the Storage daemon at the beginning of each storage volume. It contains general information about the volume. It is written in Record format. The Storage daemon manages Volume Labels, and if the client wants, he may also read them.

Begin Session Label The Begin Session Label is a special record placed by the Storage daemon on the storage medium as the first record of an append session job with a File daemon. This record is useful for finding the beginning of a particular session (Job), since no records with the same VolSessionId and VolSessionTime will precede this record. This record is not normally visible outside of the Storage daemon. The Begin Session Label is similar to the Volume Label except that it contains additional information pertaining to the Session.

End Session Label The End Session Label is a special record placed by the Storage daemon on the storage medium as the last record of an append session job with a File daemon. The End Session Record is distinguished by a FileIndex with a value of minus two (-2). This record is useful for detecting the end of a particular session since no records with the same VolSessionId and VolSessionTime will follow this record. This record is not normally visible outside of the Storage daemon. The End Session Label is similar to the Volume Label except that it contains additional information pertaining to the Session.

Storage Daemon File Output Format

The file storage and tape storage formats are identical except that tape records are by default blocked into blocks of 64,512 bytes, except for the last block, which is the actual number of bytes written rounded up to a multiple of 1024 whereas the last record of file storage is not rounded up. The default block size of 64,512 bytes may be overridden by the user (some older tape drives only support block sizes of 32K). Each Session written to tape is terminated with an End of File mark (this will be removed later). Sessions written to file are simply appended to the end of the file.

Overall Format

A Bacula output file consists of Blocks of data. Each block contains a block header followed by records. Each record consists of a record header followed by the record data. The first record on a tape will always be the Volume Label Record.

No Record Header will be split across Bacula blocks. However, Record Data may be split across any number of Bacula blocks. Obviously this will not be the case for the Volume Label which will always be smaller than the Bacula Block size.

To simplify reading tapes, the Start of Session (SOS) and End of Session (EOS) records are never split across blocks. If this is about to happen, Bacula will write a short block before writing the session record (actually, the SOS record should always be the first record in a block, excepting perhaps the Volume label).

Due to hardware limitations, the last block written to the tape may not be fully written. If your drive permits backspace record, Bacula will backup over the last record written on the tape, re-read it and verify that it was correctly written.

When a new tape is mounted Bacula will write the full contents of the partially written block to the new tape ensuring that there is no loss of data. When reading a tape, Bacula will discard any block that is not totally written, thus ensuring that there is no duplication of data. In addition, since Bacula blocks are sequentially numbered within a Job, it is easy to ensure that no block is missing or duplicated.

Serialization

All Block Headers, Record Headers, and Label Records are written using Bacula's serialization routines. These routines guarantee that the data is written to the output volume in a machine independent format.

Block Header

The format of the Block Header (version 1.27 and later) is:

```
uint32_t CheckSum;              /* Block check sum */
uint32_t BlockSize;             /* Block byte size including the header */
uint32_t BlockNumber;           /* Block number */
char ID[4] = "BB02";            /* Identification and block level */
uint32_t VolSessionId;          /* Session Id for Job */
uint32_t VolSessionTime;        /* Session Time for Job */
```

The Block header is a fixed length and fixed format and is followed by Record Headers and Record Data. The CheckSum field is a 32 bit checksum of the block data and the block header but not including the CheckSum field. The Block Header is always immediately followed by a Record Header. If the tape is damaged, a Bacula utility will be able to recover as much information as possible from the tape by recovering blocks which are valid. The Block header is written using the Bacula serialization routines and thus is guaranteed to be in machine independent format. See below for version 2 of the block header.

Record Header

Each binary data record is preceded by a Record Header. The Record Header is fixed length and fixed format, whereas the binary data record is of variable length. The Record Header is written using the Bacula serialization routines and thus is guaranteed to be in machine independent format.

The format of the Record Header (version 1.27 or later) is:

```
int32_t FileIndex;   /* File index supplied by File daemon */
int32_t Stream;      /* Stream number supplied by File daemon */
uint32_t DataSize;   /* size of following data record in bytes */
```

This record is followed by the binary Stream data of DataSize bytes, followed

by another Record Header record and the binary stream data. For the definitive definition of this record, see record.h in the src/stored directory.

Additional notes on the above:

The VolSessionId is a unique sequential number that is assigned by the Storage Daemon to a particular Job. This number is sequential since the start of execution of the daemon.

The VolSessionTime is the time/date that the current execution of the Storage Daemon started. It assures that the combination of VolSessionId and VolSessionTime is unique for every jobs written to the tape, even if there was a machine crash between two writes.

The FileIndex is a sequential file number within a job. The Storage daemon requires this index to be greater than zero and sequential. Note, however, that the File daemon may send multiple Streams for the same FileIndex. In addition, the Storage daemon uses negative FileIndices to hold the Begin Session Label, the End Session Label, and the End of Volume Label.

The Stream is defined by the File daemon and is used to identify separate parts of the data saved for each file (Unix attributes, Win32 attributes, file data, compressed file data, sparse file data, ...). The Storage Daemon has no idea of what a Stream is or what it contains except that the Stream is required to be a positive integer. Negative Stream numbers are used internally by the Storage daemon to indicate that the record is a continuation of the previous record (the previous record would not entirely fit in the block).

For Start Session and End Session Labels (where the FileIndex is negative), the Storage daemon uses the Stream field to contain the JobId. The current stream definitions are:

```
STREAM_UNIX_ATTRIBUTES   1    /* Generic Unix attributes */
STREAM_FILE_DATA         2    /* Standard uncompressed data */
STREAM_MD5_SIGNATURE     3    /* MD5 signature for the file */
STREAM_GZIP_DATA         4    /* GZip compressed file data */
STREAM_WIN32_ATTRIBUTES  5    /* Windows attributes (superset of Unix) */
STREAM_SPARSE_DATA       6    /* Sparse data stream */
STREAM_SPARSE_GZIP_DATA  7    /* Sparse data stream compressed by GZIP */
STREAM_PROGRAM_NAMES     8    /* program names for program data */
STREAM_PROGRAM_DATA      9    /* Data needing program */
STREAM_SHA1_SIGNATURE   10    /* SHA1 signature for the file */
STREAM_WIN32_DATA       11    /* Win32 BackupRead data */
STREAM_WIN32_GZIP_DATA  12    /* Gzipped Win32 BackupRead data */
```

The DataSize is the size in bytes of the binary data record that follows the Session Record header. The Storage Daemon has no idea of the actual contents of the binary data record. For standard Unix files, the data record typically contains the file attributes or the file data. For a sparse file the first 64 bits of the file data contains the storage address for the data block.

The Record Header is never split across two blocks. If there is not enough room in a block for the full Record Header, the block is padded to the end with zeros and the Record Header begins in the next block. The data record, on the other hand, may be split across multiple blocks and even multiple physical volumes. When a data record is split, the second (and possibly subsequent) piece of the data is preceded by a new Record Header. Thus each piece of data is always immediately preceded by a Record Header. When reading a record, if Bacula finds only part of the data in the first record, it will automatically read the next record and concatenate the data record to form a full data record.

Version BB02 Block Header

Each session or Job has its own private block. As a consequence, the SessionId and SessionTime are written once in each Block Header and not in the Record Header. So, the second and current version of the Block Header BB02 is:

```
uint32_t CheckSum;        /* Block check sum */
uint32_t BlockSize;       /* Block byte size including the header */
uint32_t BlockNumber;     /* Block number */
char ID[4] = "BB02";      /* Identification and block level */
uint32_t VolSessionId;    /* Applies to all records  */
uint32_t VolSessionTime;  /*   contained in this block */
```

As with the previous version, the BB02 Block header is a fixed length and fixed format and is followed by Record Headers and Record Data. The CheckSum field is a 32 bit CRC checksum of the block data and the block header but not including the CheckSum field. The Block Header is always immediately followed by a Record Header. If the tape is damaged, a Bacula utility will be able to recover as much information as possible from the tape by recovering blocks which are valid. The Block header is written using the Bacula serialization routines and thus is guaranteed to be in machine independent format.

Version 2 Record Header

Version 2 Record Header is written to the medium when using Version BB02 Block Headers. The memory representation of the record is identical to the old BB01 Record Header, but on the storage medium, the first two fields, namely VolSessionId and VolSessionTime are not written. The Block Header is filled with these values when the First user record is written (i.e. non label record) so that when the block is written, it will have the current and unique VolSessionId and VolSessionTime. On reading each record from the Block, the VolSessionId and VolSessionTime is filled in the Record Header from the Block Header.

Volume Label Format

Tape volume labels are created by the Storage daemon in response to a **label** command given to the Console program, or alternatively by the **btape** program. created. Each volume is labeled with the following information using the Bacula serialization routines, which guarantee machine byte order independence.

For Bacula versions 1.27 and later, the Volume Label Format is:

```
char Id[32];              /* Bacula 1.0 Immortal\n */
uint32_t VerNum;          /* Label version number */
/* VerNum 11 and greater Bacula 1.27 and later */
btime_t   label_btime;    /* Time/date tape labeled */
btime_t   write_btime;    /* Time/date tape first written */
/* The following are 0 in VerNum 11 and greater */
float64_t write_date;     /* Date this label written */
float64_t write_time;     /* Time this label written */
char VolName[128];        /* Volume name */
char PrevVolName[128];    /* Previous Volume Name */
char PoolName[128];       /* Pool name */
char PoolType[128];       /* Pool type */
char MediaType[128];      /* Type of this media */
char HostName[128];       /* Host name of writing computer */
char LabelProg[32];       /* Label program name */
char ProgVersion[32];     /* Program version */
char ProgDate[32];        /* Program build date/time */
```

Note, the LabelType (Volume Label, Volume PreLabel, Session Start Label, ...) is stored in the record FileIndex field of the Record Header and does not appear in the data part of the record.

Session Label

The Session Label is written at the beginning and end of each session as well as the last record on the physical medium. It has the following binary format:

```
char Id[32];              /* Bacula Immortal ... */
uint32_t VerNum;          /* Label version number */
uint32_t JobId;           /* Job id */
uint32_t VolumeIndex;     /* sequence no of vol */
/* Prior to VerNum 11 */
float64_t write_date;     /* Date this label written */
/* VerNum 11 and greater */
btime_t   write_btime;    /* time/date record written */
/* The following is zero VerNum 11 and greater */
float64_t write_time;    /* Time this label written */
char PoolName[128];        /* Pool name */
char PoolType[128];        /* Pool type */
char JobName[128];         /* base Job name */
char ClientName[128];
/* Added in VerNum 10 */
char Job[128];             /* Unique Job name */
char FileSetName[128];     /* FileSet name */
uint32_t JobType;
uint32_t JobLevel;
```

In addition, the EOS label contains:

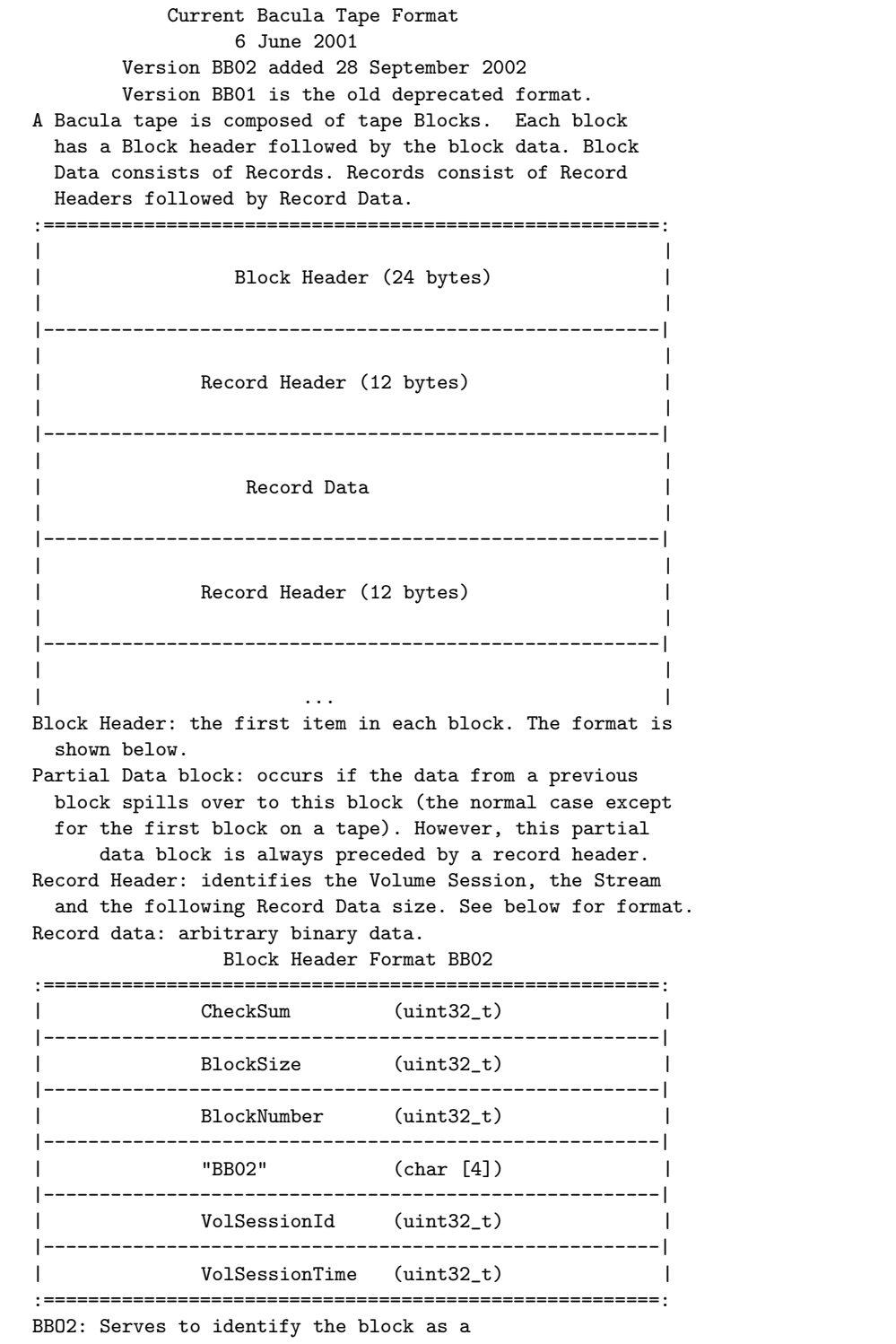
```
/* The remainder are part of EOS label only */
uint32_t JobFiles;
uint64_t JobBytes;
uint32_t start_block;
uint32_t end_block;
uint32_t start_file;
uint32_t end_file;
uint32_t JobErrors;
```

In addition, for VerNum greater than 10, the EOS label contains (in addition to the above):

```
uint32_t JobStatus          /* Job termination code */
```

: Note, the LabelType (Volume Label, Volume PreLabel, Session Start Label, ...) is stored in the record FileIndex field and does not appear in the data part of the record. Also, the Stream field of the Record Header contains the JobId. This permits quick filtering without actually reading all the session data in many cases.

Overall Storage Format



```
  Bacula block and also servers as a block format identifier
  should we ever need to change the format.
BlockSize: is the size in bytes of the block. When reading
  back a block, if the BlockSize does not agree with the
  actual size read, Bacula discards the block.
CheckSum: a checksum for the Block.
BlockNumber: is the sequential block number on the tape.
VolSessionId: a unique sequential number that is assigned
              by the Storage Daemon to a particular Job.
              This number is sequential since the start
              of execution of the daemon.
VolSessionTime: the time/date that the current execution
              of the Storage Daemon started.  It assures
              that the combination of VolSessionId and
              VolSessionTime is unique for all jobs
              written to the tape, even if there was a
              machine crash between two writes.
               Record Header Format BB02
:=======================================================:
|              FileIndex        (int32_t)               |
|-------------------------------------------------------|
|              Stream           (int32_t)               |
|-------------------------------------------------------|
|              DataSize         (uint32_t)              |
:=======================================================:
FileIndex: a sequential file number within a job.  The
              Storage daemon enforces this index to be
              greater than zero and sequential.  Note,
              however, that the File daemon may send
              multiple Streams for the same FileIndex.
              The Storage Daemon uses negative FileIndices
              to identify Session Start and End labels
              as well as the End of Volume labels.
Stream: defined by the File daemon and is intended to be
              used to identify separate parts of the data
              saved for each file (attributes, file data,
              ...).  The Storage Daemon has no idea of
              what a Stream is or what it contains.
DataSize: the size in bytes of the binary data record
              that follows the Session Record header.
              The Storage Daemon has no idea of the
              actual contents of the binary data record.
              For standard Unix files, the data record
              typically contains the file attributes or
              the file data.  For a sparse file
              the first 64 bits of the data contains
              the storage address for the data block.
                    Volume Label
:=======================================================:
|              Id               (32 bytes)              |
|-------------------------------------------------------|
|              VerNum           (uint32_t)              |
|-------------------------------------------------------|
|              label_date       (float64_t)             |
```

```
|              label_btime      (btime_t VerNum 11      |
|-------------------------------------------------------|
|              label_time       (float64_t)             |
|              write_btime      (btime_t VerNum 11      |
|-------------------------------------------------------|
|              write_date       (float64_t)             |
|                  0            (float64_t) VerNum 11   |
|-------------------------------------------------------|
|              write_time       (float64_t)             |
|                  0            (float64_t) VerNum 11   |
|-------------------------------------------------------|
|              VolName          (128 bytes)             |
|-------------------------------------------------------|
|              PrevVolName      (128 bytes)             |
|-------------------------------------------------------|
|              PoolName         (128 bytes)             |
|-------------------------------------------------------|
|              PoolType         (128 bytes)             |
|-------------------------------------------------------|
|              MediaType        (128 bytes)             |
|-------------------------------------------------------|
|              HostName         (128 bytes)             |
|-------------------------------------------------------|
|              LabelProg        (32 bytes)              |
|-------------------------------------------------------|
|              ProgVersion      (32 bytes)              |
|-------------------------------------------------------|
|              ProgDate         (32 bytes)              |
|-------------------------------------------------------|
:=======================================================:

Id: 32 byte Bacula identifier "Bacula 1.0 immortal\n"
(old version also recognized:)
Id: 32 byte Bacula identifier "Bacula 0.9 mortal\n"
LabelType (Saved in the FileIndex of the Header record).
    PRE_LABEL -1    Volume label on unwritten tape
    VOL_LABEL -2    Volume label after tape written
    EOM_LABEL -3    Label at EOM (not currently implemented)
    SOS_LABEL -4    Start of Session label (format given below)
    EOS_LABEL -5    End of Session label (format given below)
VerNum: 11
label_date: Julian day tape labeled
label_time: Julian time tape labeled
write_date: Julian date tape first used (data written)
write_time: Julian time tape first used (data written)
VolName: "Physical" Volume name
PrevVolName: The VolName of the previous tape (if this tape is
             a continuation of the previous one).
PoolName: Pool Name
PoolType: Pool Type
MediaType: Media Type
HostName: Name of host that is first writing the tape
LabelProg: Name of the program that labeled the tape
ProgVersion: Version of the label program
```

```
ProgDate: Date Label program built
                    Session Label
:=======================================================:
|              Id               (32 bytes)              |
|-------------------------------------------------------|
|              VerNum           (uint32_t)              |
|-------------------------------------------------------|
|              JobId            (uint32_t)              |
|-------------------------------------------------------|
|              write_btime      (btime_t)   VerNum 11   |
|-------------------------------------------------------|
|                 0             (float64_t) VerNum 11   |
|-------------------------------------------------------|
|              PoolName         (128 bytes)             |
|-------------------------------------------------------|
|              PoolType         (128 bytes)             |
|-------------------------------------------------------|
|              JobName          (128 bytes)             |
|-------------------------------------------------------|
|              ClientName       (128 bytes)             |
|-------------------------------------------------------|
|              Job              (128 bytes)             |
|-------------------------------------------------------|
|              FileSetName      (128 bytes)             |
|-------------------------------------------------------|
|              JobType          (uint32_t)              |
|-------------------------------------------------------|
|              JobLevel         (uint32_t)              |
|-------------------------------------------------------|
|              FileSetMD5       (50 bytes)   VerNum 11  |
|-------------------------------------------------------|
        Additional fields in End Of Session Label
|-------------------------------------------------------|
|              JobFiles         (uint32_t)              |
|-------------------------------------------------------|
|              JobBytes         (uint32_t)              |
|-------------------------------------------------------|
|              start_block      (uint32_t)              |
|-------------------------------------------------------|
|              end_block        (uint32_t)              |
|-------------------------------------------------------|
|              start_file       (uint32_t)              |
|-------------------------------------------------------|
|              end_file         (uint32_t)              |
|-------------------------------------------------------|
|              JobErrors        (uint32_t)              |
|-------------------------------------------------------|
|              JobStatus        (uint32_t) VerNum 11    |
:=======================================================:
* => fields deprecated
Id: 32 byte Bacula Identifier "Bacula 1.0 immortal\n"
LabelType (in FileIndex field of Header):
    EOM_LABEL -3     Label at EOM
    SOS_LABEL -4     Start of Session label
```

```
    EOS_LABEL -5     End of Session label
VerNum: 11
JobId: JobId
write_btime: Bacula time/date this tape record written
write_date: Julian date tape this record written - deprecated
write_time: Julian time tape this record written - deprecated.
PoolName: Pool Name
PoolType: Pool Type
MediaType: Media Type
ClientName: Name of File daemon or Client writing this session
            Not used for EOM_LABEL.
```

Unix File Attributes

The Unix File Attributes packet consists of the following:

<File-Index> <Type> <Filename>@<File-Attributes>@<Link> @<Extended-Attributes@> where

@ represents a byte containing a binary zero.

FileIndex is the sequential file index starting from one assigned by the File daemon.

Type is one of the following:

```
#define FT_LNKSAVED   1    /* hard link to file already saved */
#define FT_REGE       2    /* Regular file but empty */
#define FT_REG        3    /* Regular file */
#define FT_LNK        4    /* Soft Link */
#define FT_DIR        5    /* Directory */
#define FT_SPEC       6    /* Special file -- chr, blk, fifo, sock */
#define FT_NOACCESS   7    /* Not able to access */
#define FT_NOFOLLOW   8    /* Could not follow link */
#define FT_NOSTAT     9    /* Could not stat file */
#define FT_NOCHG     10    /* Incremental option, file not changed */
#define FT_DIRNOCHG  11    /* Incremental option, directory not changed */
#define FT_ISARCH    12    /* Trying to save archive file */
#define FT_NORECURSE 13    /* No recursion into directory */
#define FT_NOFSCHG   14    /* Different file system, prohibited */
#define FT_NOOPEN    15    /* Could not open directory */
#define FT_RAW       16    /* Raw block device */
#define FT_FIFO      17    /* Raw fifo device */
```

Filename is the fully qualified filename.

File-Attributes consists of the 13 fields of the stat() buffer in ASCII base64 format separated by spaces. These fields and their meanings

are shown below. This stat() packet is in Unix format, and MUST be provided (constructed) for ALL systems.

Link when the FT code is FT_LNK or FT_LNKSAVED, the item in question is a Unix link, and this field contains the fully qualified link name. When the FT code is not FT_LNK or FT_LNKSAVED, this field is null.

Extended-Attributes The exact format of this field is operating system dependent. It contains additional or extended attributes of a system dependent nature. Currently, this field is used only on WIN32 systems where it contains a ASCII base64 representation of the WIN32_FILE_ATTRIBUTE_DATA structure as defined by Windows. The fields in the base64 representation of this structure are like the File-Attributes separated by spaces.

The File-attributes consist of the following:

Field No.	Stat Name	Unix	Win98/NT	MacOS
1	st_dev	Device number of filesystem	Drive number	vRefNum
2	st_ino	Inode number	Always 0	fileID/dirID
3	st_mode	File mode	File mode	777 dirs/apps; 666 docs; 444 locked docs
4	st_nlink	Number of links to the file	Number of link (only on NTFS)	Always 1
5	st_uid	Owner ID	Always 0	Always 0
6	st_gid	Group ID	Always 0	Always 0
7	st_rdev	Device ID for special files	Drive No.	Always 0
8	st_size	File size in bytes	File size in bytes	Data fork file size in bytes
9	st_blksize	Preferred block size	Always 0	Preferred block size
10	st_blocks	Number of blocks allocated	Always 0	Number of blocks allocated
11	st_atime	Last access time since epoch	Last access time since epoch	Last access time -66 years

12	st_mtime	Last modify time since epoch	Last modify time since epoch	Last access time -66 years
13	st_ctime	Inode change time since epoch	File create time since epoch	File create time -66 years

Old Depreciated Tape Format

The format of the Block Header (version 1.26 and earlier) is:

```
uint32_t CheckSum;      /* Block check sum */
uint32_t BlockSize;     /* Block byte size including the header */
uint32_t BlockNumber;   /* Block number */
char ID[4] = "BB01";    /* Identification and block level */
```

The format of the Record Header (version 1.26 or earlier) is:

```
uint32_t VolSessionId;    /* Unique ID for this session */
uint32_t VolSessionTime;  /* Start time/date of session */
int32_t FileIndex;        /* File index supplied by File daemon */
int32_t Stream;           /* Stream number supplied by File daemon */
uint32_t DataSize;        /* size of following data record in bytes */
```

```
             Current Bacula Tape Format
                   6 June 2001
         Version BB01 is the old deprecated format.
 A Bacula tape is composed of tape Blocks.  Each block
   has a Block header followed by the block data. Block
   Data consists of Records. Records consist of Record
   Headers followed by Record Data.
 :=========================================================:
 |                                                         |
 |                 Block Header                            |
 |                 (16 bytes version BB01)                 |
 |---------------------------------------------------------|
 |                                                         |
 |              Record Header                              |
 |               (20 bytes version BB01)                   |
 |---------------------------------------------------------|
 |                                                         |
 |                  Record Data                            |
 |                                                         |
 |---------------------------------------------------------|
 |                                                         |
```

```
|              Record Header                              |
|               (20 bytes version BB01)                   |
|---------------------------------------------------------|
|                                                         |
|                        ...                              |
Block Header: the first item in each block. The format is
  shown below.
Partial Data block: occurs if the data from a previous
  block spills over to this block (the normal case except
  for the first block on a tape). However, this partial
      data block is always preceded by a record header.
Record Header: identifies the Volume Session, the Stream
  and the following Record Data size. See below for format.
Record data: arbitrary binary data.
                 Block Header Format BB01 (deprecated)
:=========================================================:
|              CheckSum         (uint32_t)                |
|---------------------------------------------------------|
|              BlockSize        (uint32_t)                |
|---------------------------------------------------------|
|              BlockNumber      (uint32_t)                |
|---------------------------------------------------------|
|              "BB01"           (char [4])                |
:=========================================================:
BB01: Serves to identify the block as a
  Bacula block and also servers as a block format identifier
  should we ever need to change the format.
BlockSize: is the size in bytes of the block. When reading
  back a block, if the BlockSize does not agree with the
  actual size read, Bacula discards the block.
CheckSum: a checksum for the Block.
BlockNumber: is the sequential block number on the tape.
VolSessionId: a unique sequential number that is assigned
              by the Storage Daemon to a particular Job.
              This number is sequential since the start
              of execution of the daemon.
VolSessionTime: the time/date that the current execution
              of the Storage Daemon started.  It assures
              that the combination of VolSessionId and
              VolSessionTime is unique for all jobs
              written to the tape, even if there was a
              machine crash between two writes.
               Record Header Format BB01 (deprecated)
:=========================================================:
|              VolSessionId     (uint32_t)                |
|---------------------------------------------------------|
|              VolSessionTime   (uint32_t)                |
|---------------------------------------------------------|
|              FileIndex        (int32_t)                 |
|---------------------------------------------------------|
|              Stream           (int32_t)                 |
|---------------------------------------------------------|
|              DataSize         (uint32_t)                |
:=========================================================:
```

```
VolSessionId: a unique sequential number that is assigned
              by the Storage Daemon to a particular Job.
              This number is sequential since the start
              of execution of the daemon.
VolSessionTime: the time/date that the current execution
              of the Storage Daemon started.  It assures
              that the combination of VolSessionId and
              VolSessionTime is unique for all jobs
              written to the tape, even if there was a
              machine crash between two writes.
FileIndex: a sequential file number within a job.  The
              Storage daemon enforces this index to be
              greater than zero and sequential.  Note,
              however, that the File daemon may send
              multiple Streams for the same FileIndex.
              The Storage Daemon uses negative FileIndices
              to identify Session Start and End labels
              as well as the End of Volume labels.
Stream: defined by the File daemon and is intended to be
              used to identify separate parts of the data
              saved for each file (attributes, file data,
              ...).  The Storage Daemon has no idea of
              what a Stream is or what it contains.
DataSize: the size in bytes of the binary data record
              that follows the Session Record header.
              The Storage Daemon has no idea of the
              actual contents of the binary data record.
              For standard Unix files, the data record
              typically contains the file attributes or
              the file data.  For a sparse file
              the first 64 bits of the data contains
              the storage address for the data block.
```

Volume Label

Id	(32 bytes)	
VerNum	(uint32_t)	
label_date	(float64_t)	
label_time	(float64_t)	
write_date	(float64_t)	
write_time	(float64_t)	
VolName	(128 bytes)	
PrevVolName	(128 bytes)	
PoolName	(128 bytes)	
PoolType	(128 bytes)	

```
|              MediaType        (128 bytes)             |
|-------------------------------------------------------|
|              HostName         (128 bytes)             |
|-------------------------------------------------------|
|              LabelProg        (32 bytes)              |
|-------------------------------------------------------|
|              ProgVersion      (32 bytes)              |
|-------------------------------------------------------|
|              ProgDate         (32 bytes)              |
|-------------------------------------------------------|
:=======================================================:

Id: 32 byte Bacula identifier "Bacula 1.0 immortal\n"
(old version also recognized:)
Id: 32 byte Bacula identifier "Bacula 0.9 mortal\n"
LabelType (Saved in the FileIndex of the Header record).
    PRE_LABEL -1    Volume label on unwritten tape
    VOL_LABEL -2    Volume label after tape written
    EOM_LABEL -3    Label at EOM (not currently implemented)
    SOS_LABEL -4    Start of Session label (format given below)
    EOS_LABEL -5    End of Session label (format given below)
label_date: Julian day tape labeled
label_time: Julian time tape labeled
write_date: Julian date tape first used (data written)
write_time: Julian time tape first used (data written)
VolName: "Physical" Volume name
PrevVolName: The VolName of the previous tape (if this tape is
             a continuation of the previous one).
PoolName: Pool Name
PoolType: Pool Type
MediaType: Media Type
HostName: Name of host that is first writing the tape
LabelProg: Name of the program that labeled the tape
ProgVersion: Version of the label program
ProgDate: Date Label program built
                    Session Label
:=======================================================:
|              Id               (32 bytes)              |
|-------------------------------------------------------|
|              VerNum           (uint32_t)              |
|-------------------------------------------------------|
|              JobId            (uint32_t)              |
|-------------------------------------------------------|
|             *write_date       (float64_t) VerNum 10   |
|-------------------------------------------------------|
|             *write_time       (float64_t) VerNum 10   |
|-------------------------------------------------------|
|              PoolName         (128 bytes)             |
|-------------------------------------------------------|
|              PoolType         (128 bytes)             |
|-------------------------------------------------------|
|              JobName          (128 bytes)             |
|-------------------------------------------------------|
|              ClientName       (128 bytes)             |
```

```
|-------------------------------------------------------|
|              Job              (128 bytes)             |
|-------------------------------------------------------|
|              FileSetName      (128 bytes)             |
|-------------------------------------------------------|
|              JobType          (uint32_t)              |
|-------------------------------------------------------|
|              JobLevel         (uint32_t)              |
|-------------------------------------------------------|
|              FileSetMD5       (50 bytes)   VerNum 11  |
|-------------------------------------------------------|
        Additional fields in End Of Session Label
|-------------------------------------------------------|
|              JobFiles         (uint32_t)              |
|-------------------------------------------------------|
|              JobBytes         (uint32_t)              |
|-------------------------------------------------------|
|              start_block      (uint32_t)              |
|-------------------------------------------------------|
|              end_block        (uint32_t)              |
|-------------------------------------------------------|
|              start_file       (uint32_t)              |
|-------------------------------------------------------|
|              end_file         (uint32_t)              |
|-------------------------------------------------------|
|              JobErrors        (uint32_t)              |
|-------------------------------------------------------|
|              JobStatus        (uint32_t) VerNum 11    |
:=======================================================:
* => fields deprecated
Id: 32 byte Bacula Identifier "Bacula 1.0 immortal\n"
LabelType (in FileIndex field of Header):
    EOM_LABEL -3     Label at EOM
    SOS_LABEL -4     Start of Session label
    EOS_LABEL -5     End of Session label
VerNum: 11
JobId: JobId
write_btime: Bacula time/date this tape record written
write_date: Julian date tape this record written - deprecated
write_time: Julian time tape this record written - deprecated.
PoolName: Pool Name
PoolType: Pool Type
MediaType: Media Type
ClientName: Name of File daemon or Client writing this session
            Not used for EOM_LABEL.
```

Bacula Porting Notes

This document is intended mostly for developers who wish to port Bacula to a system that is not **officially** supported.

It is hoped that Bacula clients will eventually run on every imaginable system that needs backing up (perhaps even a Palm). It is also hoped that the Bacula Directory and Storage daemons will run on every system capable of supporting them.

Porting Requirements

In General, the following holds true:

- **Bacula** has been compiled and run on Linux RedHat, FreeBSD, and Solaris systems.
- In addition, clients exist on Win32 (Cygwin), and Irix
- It requires GNU C++ to compile. You can try with other compilers, but you are on your own. The Irix client is built with the Irix complier, but, in general, you will need GNU.
- Your compiler must provide support for 64 bit signed and unsigned integers.
- You will need a recent copy of the **autoconf** tools loaded on your system (version 2.13 or later). The **autoconf** tools are used to build the configuration program, but are not part of the Bacula source distribution.
- There are certain third party packages that Bacula needs. Except for MySQL, they can all be found in the **depkgs** and **depkgs1** releases.
- If you want to build the Win32 binaries, you will need the full Cygwin 1.5.5 release. Although all components build (console has some warnings), only the File daemon has been tested. Please note that if you attempt to build Bacula on any other version of Cygwin, particularly previous versions, you will be on your own.
- **Bacula** requires a good implementation of pthreads to work.
- The source code has been written with portability in mind and is mostly POSIX compatible. Thus porting to any POSIX compatible operating system should be relatively easy.

Steps to Take for Porting

- The first step is to ensure that you have version 2.13 or later of the **autoconf** tools loaded. You can skip this step, but making changes to the configuration program will be difficult or impossible.
- The run a **./configure** command in the main source directory and examine the output. It should look something like the following:

```
Configuration on Mon Oct 28 11:42:27 CET 2002:
  Host:                           i686-pc-linux-gnu -- redhat 7.3
  Bacula version:                 1.27 (26 October 2002)
  Source code location:           .
  Install binaries:               /sbin
  Install config files:           /etc/bacula
  C Compiler:                     gcc
  C++ Compiler:                   c++
  Compiler flags:                 -g -O2
  Linker flags:
  Libraries:                      -lpthread
  Statically Linked Tools:        no
  Database found:                 no
  Database type:                  Internal
  Database lib:
  Job Output Email:               root@localhost
  Traceback Email:                root@localhost
  SMTP Host Address:              localhost
  Director Port                   9101
  File daemon Port                9102
  Storage daemon Port             9103
  Working directory               /etc/bacula/working
  SQL binaries Directory
  Large file support:             yes
  readline support:               yes
  cweb support:                   yes /home/kern/bacula/depkgs/cweb
  TCP Wrappers support:           no
  ZLIB support:                   yes
  enable-smartalloc:              yes
  enable-gnome:                   no
  gmp support:                    yes
```

  The details depend on your system. The first thing to check is that it properly identified your host on the **Host:** line. The first part (added in version 1.27) is the GNU four part identification of your system. The part after the – is your system and the system version. Generally, if your system is not yet supported, you must correct these.

- If the **./configure** does not function properly, you must determine the cause and fix it. Generally, it will be because some required system routine is not available on your machine.

- To correct problems with detection of your system type or with routines and libraries, you must edit the file **<bacula-src>/autoconf/configure.in**. This is the "source" from which **configure** is built. In general, most of the changes for your system will be made in **autoconf/aclocal.m4** in the routine **BA_CHECK_OPSYS** or in the routine **BA_CHECK_OPSYS_DISTNAME**. I have already added the necessary code for most systems, but if yours shows up as **unknown** you will need to make changes. Then as mentioned above, you will need to set a number of system dependent items in **configure.in** in the **case** statement at approximately line 1050 (depending on the Bacula release).

- The items to in the case statement that corresponds to your system are the following:

  - DISTVER – set to the version of your operating system. Typically some form of **uname** obtains it.
  - TAPEDRIVE – the default tape drive. Not too important as the user can set it as an option.
  - PSCMD – set to the **ps** command that will provide the PID in the first field and the program name in the second field. If this is not set properly, the **bacula stop** script will most likely not be able to stop Bacula in all cases.
  - hostname – command to return the base host name (non-qualified) of your system. This is generally the machine name. Not too important as the user can correct this in his configuration file.
  - CFLAGS – set any special compiler flags needed. Many systems need a special flag to make pthreads work. See cygwin for an example.
  - LDFLAGS – set any special loader flags. See cygwin for an example.
  - PTHREAD_LIB – set for any special pthreads flags needed during linking. See freebsd as an example.
  - lld – set so that a "long long int" will be properly edited in a printf() call.
  - llu – set so that a "long long unsigned" will be properly edited in a printf() call.
  - PFILES – set to add any files that you may define is your platform subdirectory. These files are used for installation of automatic system startup of Bacula daemons.

- To rebuild a new version of **configure** from a changed **autoconf/configure.in** you enter **make configure** in the top level Bacula source directory. You must have done a ./configure prior to trying to rebuild the configure script or it will get into an infinite loop.
- If the **make configure** gets into an infinite loop, ctl-c it, then do **./configure** (no options are necessary) and retry the **make configure**, which should now work.
- To rebuild **configure** you will need to have **autoconf** version 2.57-3 or higher loaded. Older versions of autoconf will complain about unknown or bad options, and won't work.
- After you have a working **configure** script, you may need to make a few system dependent changes to the way Bacula works. Generally, these are done in **src/baconfig.h**. You can find a few examples of system dependent changes toward the end of this file. For example, on Irix systems, there is no definition for **socklen_t**, so it is made in this file. If your system has structure alignment requirements, check the definition of BALIGN in this file. Currently, all Bacula allocated memory is aligned on a **double** boundary.
- If you are having problems with Bacula's type definitions, you might look at **src/bc_types.h** where all the types such as **uint32_t**, **uint64_t**, etc. that Bacula uses are defined.

Bacula Regression Testing

General

This document is intended mostly for developers who wish to ensure that their changes to Bacula don't introduce bugs in the base code.

You can find the existing regression script in the Bacula CVS on the SourceForge CVS in the project tree named **regress**.

There are two different aspects of regression testing that this document will discuss: 1. Running the Regression Script, 2. Writing a Regression test.

Running the Regression Script

There are a number of different tests that may be run, such as: the standard set that uses disk Volumes and runs under any userid; a small set of tests that write to tape; another set of tests where you must be root to run them. To date, each subset of tests runs no more than about 15 minutes.

Setting the Configuration Parameters

Once you have the regression directory loaded, you will first need to create a custom xxx.conf file for your system. You can either edit **prototype.conf** directly or copy it to a new file and edit it. To see a real example of a configuration file, look at **kern.conf**. The variables you need to modify are:

```
# Where to get the source to be tested
BACULA_SOURCE="${HOME}/bacula/k"

# Where to send email   !!!! Change me !!!!!!!
EMAIL=your-email@domain.com

# Full path where to find sqlite
DEPKGS="${HOME}/bacula/depkgs/sqlite"

TAPE_DRIVE="/dev/nst0"
# if you don't have an autochanger set
#   AUTOCHANGER to /dev/null
AUTOCHANGER="/dev/sg0"
# This must be the path to the autochanger
#  including its name
AUTOCHANGER_PATH="/bin/mtx"
```

- **BACULA_SOURCE** should be the full path to the Bacula source code that you wish to test.
- **EMAIL** should be your email addres. Please remember to change this or I will get a flood of unwanted messages. You may or may not want to see these emails. In my case, I don't need them so I direct it to the bit bucket.
- **SQLITE_DIR** should be the full path to the sqlite package, must be build before running a Bacula regression, if you are using SQLite. This variable is ignored if you are using MySQL or PostgreSQL. To use PostgreSQL, edit the Makefile and change (or add) WHICHDB?="--with-postgresql". For MySQL use "WHICHDB?="--with-mysql".
- **TAPE_DRIVE** is the full path to your tape drive. The base set of regression tests do not use a tape, so this is only important if you want to run the full tests.
- **AUTOCHANGER** is the name of your autochanger device. Set this to /dev/null if you do not have one.
- **AUTOCHANGER_PATH** is the full path including the program name for your autochanger program (normally **mtx**. Leave the default value if you do not have one.

Building the Test Bacula

Once the above variables are set, you can build Bacula by entering:

```
./config xxx.conf
make setup
```

Where xxx.conf is the name of the conf file containing your system parameters. This will build a Makefile from Makefile.in, then copy the source code within the regression tree (in directory regress/build), configure it, and build it. There should be no errors. If there are, please correct them before continuing.

Running the Disk Only Regression

Once Bacula is built, you can run the basic disk only non-root regression test by entering:

```
make test
```

This will run the base set of tests using disk Volumes, currently (19 Dec 2003), there are current 18 separate tests that run. If you are testing on a non-Linux machine two of the tests will not be run. In any case, as we add new tests, the number will vary. It will take about 5 or 10 minutes if you have a fast (2 GHz) machine, and you don't need to be root to run these tests (I run under my regular userid). The result should be something similar to:

```
Test results

  ===== Backup Bacula Test OK =====
  ===== Verify Volume Test OK =====
  ===== sparse-test OK =====
  ===== compressed-test OK =====
  ===== sparse-compressed-test OK =====
  ===== Weird files test OK =====
  ===== two-jobs-test OK =====
  ===== two-vol-test OK =====
  ===== six-vol-test OK =====
  ===== bscan-test OK =====
  ===== Weird files2 test OK =====
  ===== concurrent-jobs-test OK =====
  ===== four-concurrent-jobs-test OK =====
  ===== bsr-opt-test OK =====
  ===== bextract-test OK =====
  ===== recycle-test OK =====
  ===== span-vol-test OK =====
  ===== restore-by-file-test OK =====
  ===== restore2-by-file-test OK =====
  ===== four-jobs-test OK =====
  ===== incremental-test OK =====
```

and the working tape tests are:

```
Test results

  ===== Bacula tape test OK =====
  ===== Small File Size test OK =====
  ===== restore-by-file-tape test OK =====
```

```
===== incremental-tape test OK =====
===== four-concurrent-jobs-tape OK =====
===== four-jobs-tape OK =====
```

Each separate test is self contained in that it initializes to run Bacula from scratch (i.e. newly created database). It will also kill any Bacula session that is currently running. In addition, it uses ports 8101, 8102, and 8103 so that it does not intefere with a production system.

Other Tests

There are a number of other tests that can be run as well. All the tests are a simply shell script keep in the regress directory. For example the "make test" simply executes **./all-non-root-tests**. The other tests are:

all_non-root-tests All non-tape tests not requiring root. This is the standard set of tests, that in general, backup some data, then restore it, and finally compares the restored data with the original data.

all-root-tests All non-tape tests requiring root permission. These are a relatively small number of tests that require running as root. The amount of data backed up can be quite large. For example, one test backs up /usr, another backs up /etc. One or more of these tests reports an error – I'll fix it one day.

all-non-root-tape-tests All tape test not requiring root. There are currently three tests, all run without being root, and backup to a tape. The first two tests use one volume, and the third test requires an autochanger, and uses two volumes. If you don't have an autochanger, then this script will probably produce an error.

all-tape-and-file-tests All tape and file tests not requiring root. This includes just about everything, and I don't run it very often.

If a Test Fails

If you one or more tests fail, the line output will be similar to:

```
!!!!! concurrent-jobs-test failed!!! !!!!!
```

If you want to determine why the test failed, you will need to modify the script so that it prints. Do so by finding the file in **regress/tests** that

corresponds to the name printed. For example, the script for the above error message is in: **regress/tests/concurrent-jobs-test**.

In order to see the output produced by Bacula, you need only change the lines that start with **@output** to **@tee**, then rerun the test by hand. it is very important to start the test from the **regress** directory.

To modify the test mentioned above so that you can see the output, change every occurrence of **@output** to **@tee** in the file. In rare cases you might need to remove the **2>&1 >/dev/null** from the end of the **bacula**, **bconsole**, or **diff** lines, but this is rare.

Writing a Regression Test

Any developer, who implements a major new feature, should write a regression test that exercises and validates the new feature. Each regression test is a complete test by itself. It terminates any running Bacula, initializes the database, starts Bacula, then runs the test by using the console program.

Running the Tests by Hand

You can run any individual test by hand by cd'ing to the **regress** directory and entering:

```
tests/<test-name>
```

Directory Structure

The directory structure of the regression tests is:

```
regress                - Makefile, scripts to start tests
  |------ scripts      - Scripts and conf files
  |-------tests        - All test scripts are here
  |
  |------------------ -- All directories below this point are used
  |                      for testing, but are created from the
  |                      above directories and are removed with
  |                      "make distclean"
  |
  |------ bin          - This is the install directory for
  |                       Bacula to be used testing
  |------ build        - Where the Bacula source build tree is
```

```
|------ tmp          - Most temp files go here
|------ working      - Bacula working directory
|------ weird-files  - Weird files used in two of the tests.
```

Adding a New Test

If you want to write a new regression test, it is best to start with one of the existing test scripts, and modify it to do the new test.

When adding a new test, be extremely careful about adding anything to any of the daemons' configuration files. The reason is that it may change the prompts that are sent to the console. For example, adding a Pool means that the current scripts, which assume that Bacula automatically selects a Pool, will now be presented with a new prompt, so the test will fail. If you need to enhance the configuration files, consider making your own versions.

Bacula MD5 Algorithm

Command Line Message Digest Utility

This page describes **md5**, a command line utility usable on either Unix or MS-DOS/Windows, which generates and verifies message digests (digital signatures) using the MD5 algorithm. This program can be useful when developing shell scripts or Perl programs for software installation, file comparison, and detection of file corruption and tampering.

Name

md5 - generate / check MD5 message digest

Synopsis

md5 [**-c***signature*] [**-u**] [**-d***input_text* — *infile*] [*outfile*]

Description

A *message digest* is a compact digital signature for an arbitrarily long stream of binary data. An ideal message digest algorithm would never generate the same signature for two different sets of input, but achieving such theoretical perfection would require a message digest as long as the input file. Practical message digest algorithms compromise in favour of a digital signature of modest size created with an algorithm designed to make preparation of input text with a given signature computationally infeasible. Message digest algorithms have much in common with techniques used in encryption, but to a different end; verification that data have not been altered since the signature was published.

Many older programs requiring digital signatures employed 16 or 32 bit *cyclical redundancy codes* (CRC) originally developed to verify correct transmission in data communication protocols, but these short codes, while adequate to detect the kind of transmission errors for which they were intended, are insufficiently secure for applications such as electronic commerce and verification of security related software distributions.

The most commonly used present-day message digest algorithm is the 128 bit MD5 algorithm, developed by Ron Rivest of the MIT

Laboratory for Computer Science and RSA Data Security, Inc. The algorithm, with a reference implementation, was published as Internet RFC 1321 in April 1992, and was placed into the public domain at that time. Message digest algorithms such as MD5 are not deemed "encryption technology" and are not subject to the export controls some governments impose on other data security products. (Obviously, the responsibility for obeying the laws in the jurisdiction in which you reside is entirely your own, but many common Web and Mail utilities use MD5, and I am unaware of any restrictions on their distribution and use.)

The MD5 algorithm has been implemented in numerous computer languages including C, Perl, and Java; if you're writing a program in such a language, track down a suitable subroutine and incorporate it into your program. The program described on this page is a *command line* implementation of MD5, intended for use in shell scripts and Perl programs (it is much faster than computing an MD5 signature directly in Perl). This **md5** program was originally developed as part of a suite of tools intended to monitor large collections of files (for example, the contents of a Web site) to detect corruption of files and inadvertent (or perhaps malicious) changes. That task is now best accomplished with more comprehensive packages such as Tripwire, but the command line **md5** component continues to prove useful for verifying correct delivery and installation of software packages, comparing the contents of two different systems, and checking for changes in specific files.

Options

-c*signature* Computes the signature of the specified *infile* or the string supplied by the **-d** option and compares it against the specified *signature*. If the two signatures match, the exit status will be zero, otherwise the exit status will be 1. No signature is written to *outfile* or standard output; only the exit status is set. The signature to be checked must be specified as 32 hexadecimal digits.

-d*input_text* A signature is computed for the given *input_text* (which must be quoted if it contains white space characters) instead of input from *infile* or standard input. If input is specified with the **-d** option, no *infile* should be specified.

-u Print how-to-call information.

Files

If no *infile* or **-d** option is specified or *infile* is a single "-", **md5** reads from standard input; if no *outfile* is given, or *outfile* is a single "-", output is sent to standard output. Input and output are processed strictly serially; consequently **md5** may be used in pipelines.

Bugs

The mechanism used to set standard input to binary mode may be specific to Microsoft C; if you rebuild the DOS/Windows version of the program from source using another compiler, be sure to verify binary files work properly when read via redirection or a pipe.

This program has not been tested on a machine on which `int` and/or `long` are longer than 32 bits.

Download md5.zip (Zipped archive)

The program is provided as md5.zip, a Zipped archive containing an ready-to-run Win32 command-line executable program, `md5.exe` (compiled using Microsoft Visual C++ 5.0), and in source code form along with a `Makefile` to build the program under Unix.

See Also

sum(1)

Exit Status

md5 returns status 0 if processing was completed without errors, 1 if the **-c** option was specified and the given signature does not match that of the input, and 2 if processing could not be performed at all due, for example, to a nonexistent input file.

Copying

Acknowledgements

The MD5 algorithm was developed by Ron Rivest. The public domain C language implementation used in this program was written by Colin Plumb in 1993. *by John Walker January 6th, MIM*

Bacula Memory Management

General

This document describes the memory management routines that are used in Bacula and is meant to be a technical discussion for developers rather than part of the user manual.

Since Bacula may be called upon to handle filenames of varying and more or less arbitrary length, special attention needs to be used in the code to ensure that memory buffers are sufficiently large. There are four possibilities for memory usage within **Bacula**. Each will be described in turn. They are:

- Statically allocated memory.
- Dynamically allocated memory using malloc() and free().
- Non-pooled memory.
- Pooled memory.

Statically Allocated Memory

Statically allocated memory is of the form:

```
char buffer[MAXSTRING];
```

The use of this kind of memory is discouraged except when you are 100% sure that the strings to be used will be of a fixed length. One example of where this is appropriate is for **Bacula** resource names, which are currently limited to 127 characters (MAX_NAME_LENGTH). Although this maximum size may change, particularly to accommodate Unicode, it will remain a relatively small value.

Dynamically Allocated Memory

Dynamically allocated memory is obtained using the standard malloc() routines. As in:

```
char *buf;
buf = malloc(256);
```

This kind of memory can be released with:

```
free(buf);
```

It is recommended to use this kind of memory only when you are sure that you know the memory size needed and the memory will be used for short periods of time – that is it would not be appropriate to use statically allocated memory. An example might be to obtain a large memory buffer for reading and writing files. When **SmartAlloc** is enabled, the memory obtained by malloc() will automatically be checked for buffer overwrite (overflow) during the free() call, and all malloc'ed memory that is not released prior to termination of the program will be reported as Orphaned memory.

Pooled and Non-pooled Memory

In order to facility the handling of arbitrary length filenames and to efficiently handle a high volume of dynamic memory usage, we have implemented routines between the C code and the malloc routines. The first is called "Pooled" memory, and is memory, which once allocated and then released, is not returned to the system memory pool, but rather retained in a Bacula memory pool. The next request to acquire pooled memory will return any free memory block. In addition, each memory block has its current size associated with the block allowing for easy checking if the buffer is of sufficient size. This kind of memory would normally be used in high volume situations (lots of malloc()s and free()s) where the buffer length may have to frequently change to adapt to varying filename lengths.

The non-pooled memory is handled by routines similar to those used for pooled memory, allowing for easy size checking. However, non-pooled memory is returned to the system rather than being saved in the Bacula pool. This kind of memory would normally be used in low volume situations (few malloc()s and free()s), but where the size of the buffer might have to be adjusted frequently.

Types of Memory Pool: Currently there are three memory pool types:

- PM_NOPOOL – non-pooled memory.
- PM_FNAME – a filename pool.
- PM_MESSAGE – a message buffer pool.
- PM_EMSG – error message buffer pool.

Getting Memory: To get memory, one uses:

```
void *get_pool_memory(pool);
```

where **pool** is one of the above mentioned pool names. The size of the memory returned will be determined by the system to be most appropriate for the application.

If you wish non-pooled memory, you may alternatively call:

```
void *get_memory(size_t size);
```

The buffer length will be set to the size specified, and it will be assigned to the PM_NOPOOL pool (no pooling).

Releasing Memory: To free memory acquired by either of the above two calls, use:

```
void free_pool_memory(void *buffer);
```

where buffer is the memory buffer returned when the memory was acquired. If the memory was originally allocated as type PM_NOPOOL, it will be released to the system, otherwise, it will be placed on the appropriate Bacula memory pool free chain to be used in a subsequent call for memory from that pool.

Determining the Memory Size: To determine the memory buffer size, use:

```
size_t sizeof_pool_memory(void *buffer);
```

Resizing Pool Memory: To resize pool memory, use:

```
void *realloc_pool_memory(void *buffer);
```

The buffer will be reallocated, and the contents of the original buffer will be preserved, but the address of the buffer may change.

Automatic Size Adjustment: To have the system check and if necessary adjust the size of your pooled memory buffer, use:

```
void *check_pool_memory_size(void *buffer, size_t new-size);
```

where **new-size** is the buffer length needed. Note, if the buffer is already equal to or larger than **new-size** no buffer size change will occur. However, if a buffer size change is needed, the original contents of the buffer will be preserved, but the buffer address may change. Many of the low level Bacula subroutines expect to be passed a pool memory buffer and use this call to ensure the buffer they use is sufficiently large.

Releasing All Pooled Memory: In order to avoid orphaned buffer error messages when terminating the program, use:

```
void close_memory_pool();
```

to free all unused memory retained in the Bacula memory pool. Note, any memory not returned to the pool via free_pool_memory() will not be released by this call.

Pooled Memory Statistics: For debugging purposes and performance tuning, the following call will print the current memory pool statistics:

```
void print_memory_pool_stats();
```

an example output is:

```
Pool  Maxsize  Maxused  Inuse
   0      256        0      0
   1      256        1      0
   2      256        1      0
```

TCP/IP Network Protocol

General

This document describes the TCP/IP protocol used by Bacula to communicate between the various daemons and services. The definitive definition of the protocol can be found in src/lib/bsock.h, src/lib/bnet.c and src/lib/bnet_server.c.

Bacula's network protocol is basically a "packet oriented" protocol built on a standard TCP/IP streams. At the lowest level all packet transfers are done with read() and write() requests on system sockets. Pipes are not used as they are considered unreliable for large serial data transfers between various hosts.

Using the routines described below (bnet_open, bnet_write, bnet_recv, and bnet_close) guarantees that the number of bytes you write into the socket will be received as a single record on the other end regardless of how many low level write() and read() calls are needed. All data transferred are considered to be binary data.

bnet and Threads

These bnet routines work fine in a threaded environment. However, they assume that there is only one reader or writer on the socket at any time. It is highly recommended that only a single thread access any BSOCK packet. The exception to this rule is when the socket is first opened and it is waiting for a job to start. The wait in the Storage daemon is done in one thread and then passed to another thread for subsequent handling.

If you envision having two threads using the same BSOCK, think twice, then you must implement some locking mechanism. However, it probably would not be appropriate to put locks inside the bnet subroutines for efficiency reasons.

bnet_open

To establish a connection to a server, use the subroutine:

BSOCK *bnet_open(void *jcr, char *host, char *service, int port, int *fatal) bnet_open(), if successful, returns the Bacula sock descriptor pointer to be used in subsequent bnet_send() and bnet_read() requests. If not successful,

bnet_open() returns a NULL. If fatal is set on return, it means that a fatal error occurred and that you should not repeatedly call bnet_open(). Any error message will generally be sent to the JCR.

bnet_send

To send a packet, one uses the subroutine:

int bnet_send(BSOCK *sock) This routine is equivalent to a write() except that it handles the low level details. The data to be sent is expected to be in sock->msg and be sock->msglen bytes. To send a packet, bnet_send() first writes four bytes in network byte order than indicate the size of the following data packet. It returns:

```
Returns 0 on failure
Returns 1 on success
```

In the case of a failure, an error message will be sent to the JCR contained within the bsock packet.

bnet_fsend

This form uses:

int bnet_fsend(BSOCK *sock, char *format, ...) and it allows you to send a formatted messages somewhat like fprintf(). The return status is the same as bnet_send.

Additional Error information

Fro additional error information, you can call **is_bnet_error(BSOCK *bsock)** which will return 0 if there is no error or non-zero if there is an error on the last transmission. The **is_bnet_stop(BSOCK *bsock)** function will return 0 if there no errors and you can continue sending. It will return non-zero if there are errors or the line is closed (no more transmissions should be sent).

bnet_recv

To read a packet, one uses the subroutine:

int bnet_recv(BSOCK *sock) This routine is similar to a read() except that it handles the low level details. bnet_read() first reads packet length that follows as four bytes in network byte order. The data is read into sock->msg and is sock->msglen bytes. If the sock->msg is not large enough, bnet_recv() realloc() the buffer. It will return an error (-2) if maxbytes is less than the record size sent. It returns:

```
* Returns number of bytes read
* Returns 0 on end of file
* Returns -1 on hard end of file (i.e. network connection close)
* Returns -2 on error
```

It should be noted that bnet_recv() is a blocking read.

bnet_sig

To send a "signal" from one daemon to another, one uses the subroutine:

int bnet_sig(BSOCK *sock, SIGNAL) where SIGNAL is one of the following:

1. BNET_EOF - deprecated use BNET_EOD
2. BNET_EOD - End of data stream, new data may follow
3. BNET_EOD_POLL - End of data and poll all in one
4. BNET_STATUS - Request full status
5. BNET_TERMINATE - Conversation terminated, doing close()
6. BNET_POLL - Poll request, I'm hanging on a read
7. BNET_HEARTBEAT - Heartbeat Response requested
8. BNET_HB_RESPONSE - Only response permitted to HB
9. BNET_PROMPT - Prompt for UA

bnet_strerror

Returns a formated string corresponding to the last error that occurred.

bnet_close

The connection with the server remains open until closed by the subroutine:

void bnet_close(BSOCK *sock)

Becoming a Server

The bnet_open() and bnet_close() routines described above are used on the client side to establish a connection and terminate a connection with the server. To become a server (i.e. wait for a connection from a client), use the routine **bnet_thread_server**. The calling sequence is a bit complicated, please refer to the code in bnet_server.c and the code at the beginning of each daemon as examples of how to call it.

Higher Level Conventions

Within Bacula, we have established the convention that any time a single record is passed, it is sent with bnet_send() and read with bnet_recv(). Thus the normal exchange between the server (S) and the client (C) are:

```
S: wait for connection            C: attempt connection
S: accept connection              C: bnet_send() send request
S: bnet_recv() wait for request
S: act on request
S: bnet_send() send ack            C: bnet_recv() wait for ack
```

Thus a single command is sent, acted upon by the server, and then acknowledged.

In certain cases, such as the transfer of the data for a file, all the information or data cannot be sent in a single packet. In this case, the convention is that the client will send a command to the server, who knows that more than one packet will be returned. In this case, the server will enter a loop:

```
while ((n=bnet_recv(bsock)) > 0) {
   act on request
}
if (n < 0)
   error
```

The client will perform the following:

```
bnet_send(bsock);
bnet_send(bsock);
...
bnet_sig(bsock, BNET_EOD);
```

Thus the client will send multiple packets and signal to the server when all the packets have been sent by sending a zero length record.

Smart Memory Allocation With Orphaned Buffer Detection

Few things are as embarrassing as a program that leaks, yet few errors are so easy to commit or as difficult to track down in a large, complicated program as failure to release allocated memory. SMARTALLOC replaces the standard C library memory allocation functions with versions which keep track of buffer allocations and releases and report all orphaned buffers at the end of program execution. By including this package in your program during development and testing, you can identify code that loses buffers right when it's added and most easily fixed, rather than as part of a crisis debugging push when the problem is identified much later in the testing cycle (or even worse, when the code is in the hands of a customer). When program testing is complete, simply recompiling with different flags removes SMARTALLOC from your program, permitting it to run without speed or storage penalties.

In addition to detecting orphaned buffers, SMARTALLOC also helps to find other common problems in management of dynamic storage including storing before the start or beyond the end of an allocated buffer, referencing data through a pointer to a previously released buffer, attempting to release a buffer twice or releasing storage not obtained from the allocator, and assuming the initial contents of storage allocated by functions that do not guarantee a known value. SMARTALLOC's checking does not usually add a large amount of overhead to a program (except for programs which use `realloc()` extensively; see below). SMARTALLOC focuses on proper storage management rather than internal consistency of the heap as checked by the malloc_debug facility available on some systems. SMARTALLOC does not conflict with malloc_debug and both may be used together, if you wish. SMARTALLOC makes no assumptions regarding the internal structure of the heap and thus should be compatible with any C language implementation of the standard memory allocation functions.

Installing SMARTALLOC

SMARTALLOC is provided as a Zipped archive, smartall.zip; see the download instructions below.

To install SMARTALLOC in your program, simply add the statement:

to every C program file which calls any of the memory allocation functions (`malloc`, `calloc`, `free`, etc.). SMARTALLOC must be used for all memory allocation with a program, so include file for your entire program, if you have such a thing. Next, define the symbol SMARTALLOC in the compilation before the inclusion of smartall.h. I usually do this by having my Makefile add the "`-DSMARTALLOC`" option to the C compiler for non-production builds. You can define the symbol manually, if you prefer, by adding the statement:

```
#define SMARTALLOC
```

At the point where your program is all done and ready to relinquish control to the operating system, add the call:

```
sm_dump(datadump);
```

where *datadump* specifies whether the contents of orphaned buffers are to be dumped in addition printing to their size and place of allocation. The data are dumped only if *datadump* is nonzero, so most programs will normally use "`sm_dump(0);`". If a mysterious orphaned buffer appears that can't be identified from the information this prints about it, replace the statement with "`sm_dump(1);`". Usually the dump of the buffer's data will furnish the additional clues you need to excavate and extirpate the elusive error that left the buffer allocated.

Finally, add the files "smartall.h" and "smartall.c" from this release to your source directory, make dependencies, and linker input. You needn't make inclusion of smartall.c in your link optional; if compiled with SMARTALLOC not defined it generates no code, so you may always include it knowing it will waste no storage in production builds. Now when you run your program, if it leaves any buffers around when it's done, each will be reported by `sm_dump()` on stderr as follows:

```
Orphaned buffer:     120 bytes allocated at line 50 of gutshot.c
```

Squelching a SMARTALLOC

Usually, when you first install SMARTALLOC in an existing program you'll find it nattering about lots of orphaned buffers. Some of these turn out to be legitimate errors, but some are storage allocated during program initialisation that, while dynamically allocated, is logically static storage not intended to be released. Of course, you can get rid of the complaints about these buffers by adding code to release them, but by doing so you're adding unnecessary complexity and code size to your program just to silence the nattering of a SMARTALLOC, so an escape hatch is provided to eliminate the need to release these buffers.

Normally all storage allocated with the functions `malloc()`, `calloc()`, and `realloc()` is monitored by SMARTALLOC. If you make the function call:

```
sm_static(1);
```

you declare that subsequent storage allocated by `malloc()`, `calloc()`, and `realloc()` should not be considered orphaned if found to be allocated when `sm_dump()` is called. I use a call on "`sm_static(1);`" before I allocate things like program configuration tables so I don't have to add code to release them at end of program time. After allocating unmonitored data this way, be sure to add a call to:

```
sm_static(0);
```

to resume normal monitoring of buffer allocations. Buffers allocated while `sm_static(1)` is in effect are not checked for having been orphaned but all the other safeguards provided by SMARTALLOC remain in effect. You may release such buffers, if you like; but you don't have to.

Living with Libraries

Some library functions for which source code is unavailable may gratuitously allocate and return buffers that contain their results, or require you to pass them buffers which they subsequently release. If you have source code for the library, by far the best approach is to simply install SMARTALLOC in it, particularly since this kind of ill-structured dynamic storage management is the source of so many storage leaks. Without source code, however, there's no option but to provide a way to bypass SMARTALLOC for the buffers the library allocates and/or releases with the standard system functions.

For each function *xxx* redefined by SMARTALLOC, a corresponding routine named "`actually`*xxx*" is furnished which provides direct access to the underlying system function, as follows:

Standard function	**Direct access function**
`malloc(`*size*`)`	`actuallymalloc(`*size*`)`
`calloc(`*nelem*`, `*elsize*`)`	`actuallycalloc(`*nelem*`, `*elsize*`)`
`realloc(`*ptr*`, `*size*`)`	`actuallyrealloc(`*ptr*`, `*size*`)`
`free(`*ptr*`)`	`actuallyfree(`*ptr*`)`

For example, suppose there exists a system library function named "`getimage()`" which reads a raster image file and returns the address of a buffer containing it. Since the library routine allocates the image directly with `malloc()`, you can't use SMARTALLOC's `free()`, as that call expects information placed in the buffer by SMARTALLOC's special version of `malloc()`, and hence would report an error. To release the buffer you should call `actuallyfree()`, as in this code fragment:

```
struct image *ibuf = getimage("ratpack.img");
display_on_screen(ibuf);
actuallyfree(ibuf);
```

Conversely, suppose we are to call a library function, "`putimage()`", which writes an image buffer into a file and then releases the buffer with `free()`. Since the system `free()` is being called, we can't pass a buffer allocated by SMARTALLOC's allocation routines, as it contains special information that the system `free()` doesn't expect to be there. The following code uses `actuallymalloc()` to obtain the buffer passed to such a routine.

```
struct image *obuf =
   (struct image *) actuallymalloc(sizeof(struct image));
dump_screen_to_image(obuf);
putimage("scrdump.img", obuf);  /* putimage() releases obuf */
```

It's unlikely you'll need any of the "actually" calls except under very odd circumstances (in four products and three years, I've only needed them once), but they're there for the rare occasions that demand them. Don't use them to subvert the error checking of SMARTALLOC; if you want to disable orphaned buffer detection, use the `sm_static(1)` mechanism described above. That way you don't forfeit all the other advantages of SMARTALLOC as you do when using `actuallymalloc()` and `actuallyfree()`.

SMARTALLOC Details

When you include "smartall.h" and define SMARTALLOC, the following standard system library functions are redefined with the #define mechanism to call corresponding functions within smartall.c instead. (For details of the redefinitions, please refer to smartall.h.)

```
void *malloc(size_t size)
void *calloc(size_t nelem, size_t elsize)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
void cfree(void *ptr)
```

`cfree()` is a historical artifact identical to `free()`.

In addition to allocating storage in the same way as the standard library functions, the SMARTALLOC versions expand the buffers they allocate to include information that identifies where each buffer was allocated and to chain all allocated buffers together. When a buffer is released, it is removed from the allocated buffer chain. A call on `sm_dump()` is able, by scanning the chain of allocated buffers, to find all orphaned buffers. Buffers allocated while `sm_static(1)` is in effect are specially flagged so that, despite appearing on the allocated buffer chain, `sm_dump()` will not deem them orphans.

When a buffer is allocated by `malloc()` or expanded with `realloc()`, all bytes of newly allocated storage are set to the hexadecimal value 0x55 (alternating one and zero bits). Note that for `realloc()` this applies only to the bytes added at the end of buffer; the original contents of the buffer are not modified. Initializing allocated storage to a distinctive nonzero pattern is intended to catch code that erroneously assumes newly allocated buffers are cleared to zero; in fact their contents are random. The `calloc()` function, defined as returning a buffer cleared to zero, continues to zero its buffers under SMARTALLOC.

Buffers obtained with the SMARTALLOC functions contain a special sentinel byte at the end of the user data area. This byte is set to a special key value based upon the buffer's memory address. When the buffer is released, the key is tested and if it has been overwritten an assertion in the `free` function will fail. This catches incorrect program code that stores beyond the storage allocated for the buffer. At `free()` time the queue links are also validated and an assertion failure will occur if the program has destroyed them by storing before the start of the allocated storage.

In addition, when a buffer is released with `free()`, its contents are immediately destroyed by overwriting them with the hexadecimal pattern 0xAA

(alternating bits, the one's complement of the initial value pattern). This will usually trip up code that keeps a pointer to a buffer that's been freed and later attempts to reference data within the released buffer. Incredibly, this is *legal* in the standard Unix memory allocation package, which permits programs to free() buffers, then raise them from the grave with `realloc()`. Such program "logic" should be fixed, not accommodated, and SMARTALLOC brooks no such Lazarus buffer" nonsense.

Some C libraries allow a zero size argument in calls to `malloc()`. Since this is far more likely to indicate a program error than a defensible programming stratagem, SMARTALLOC disallows it with an assertion.

When the standard library `realloc()` function is called to expand a buffer, it attempts to expand the buffer in place if possible, moving it only if necessary. Because SMARTALLOC must place its own private storage in the buffer and also to aid in error detection, its version of `realloc()` always moves and copies the buffer except in the trivial case where the size of the buffer is not being changed. By forcing the buffer to move on every call and destroying the contents of the old buffer when it is released, SMARTALLOC traps programs which keep pointers into a buffer across a call on `realloc()` which may move it. This strategy may prove very costly to programs which make extensive use of `realloc()`. If this proves to be a problem, such programs may wish to use `actuallymalloc()`, `actuallyrealloc()`, and `actuallyfree()` for such frequently-adjusted buffers, trading error detection for performance. Although not specified in the System V Interface Definition, many C library implementations of `realloc()` permit an old buffer argument of NULL, causing `realloc()` to allocate a new buffer. The SMARTALLOC version permits this.

When SMARTALLOC is Disabled

When SMARTALLOC is disabled by compiling a program with the symbol SMARTALLOC not defined, calls on the functions otherwise redefined by SMARTALLOC go directly to the system functions. In addition, compile-time definitions translate calls on the "`actually...()`" functions into the corresponding library calls; "`actuallymalloc(100)`", for example, compiles into "`malloc(100)`". The two special SMARTALLOC functions, `sm_dump()` and `sm_static()`, are defined to generate no code (hence the null statement). Finally, if SMARTALLOC is not defined, compilation of the file smartall.c generates no code or data at all, effectively removing it from the program even if named in the link instructions.

Thus, except for unusual circumstances, a program that works with SMAR-

TALLOC defined for testing should require no changes when built without it for production release.

The `alloc()` Function

Many programs I've worked on use very few direct calls to `malloc()`, using the identically declared `alloc()` function instead. Alloc detects out-of-memory conditions and aborts, removing the need for error checking on every call of `malloc()` (and the temptation to skip checking for out-of-memory).

As a convenience, SMARTALLOC supplies a compatible version of `alloc()` in the file alloc.c, with its definition in the file alloc.h. This version of `alloc()` is sensitive to the definition of SMARTALLOC and cooperates with SMARTALLOC's orphaned buffer detection. In addition, when SMARTALLOC is defined and `alloc()` detects an out of memory condition, it takes advantage of the SMARTALLOC diagnostic information to identify the file and line number of the call on `alloc()` that failed.

Overlays and Underhandedness

String constants in the C language are considered to be static arrays of characters accessed through a pointer constant. The arrays are potentially writable even though their pointer is a constant. SMARTALLOC uses the compile-time definition `./smartall.wml` to obtain the name of the file in which a call on buffer allocation was performed. Rather than reserve space in a buffer to save this information, SMARTALLOC simply stores the pointer to the compiled-in text of the file name. This works fine as long as the program does not overlay its data among modules. If data are overlayed, the area of memory which contained the file name at the time it was saved in the buffer may contain something else entirely when `sm_dump()` gets around to using the pointer to edit the file name which allocated the buffer.

If you want to use SMARTALLOC in a program with overlayed data, you'll have to modify smartall.c to either copy the file name to a fixed-length field added to the `abufhead` structure, or else allocate storage with `malloc()`, copy the file name there, and set the `abfname` pointer to that buffer, then remember to release the buffer in `sm_free`. Either of these approaches are wasteful of storage and time, and should be considered only if there is no alternative. Since most initial debugging is done in non-overlayed environments, the restrictions on SMARTALLOC with data overlaying may never prove a problem. Note that conventional overlaying of code, by far the most common form of overlaying, poses no problems for SMARTALLOC; you

need only be concerned if you're using exotic tools for data overlaying on MS-DOS or other address-space-challenged systems.

Since a C language "constant" string can actually be written into, most C compilers generate a unique copy of each string used in a module, even if the same constant string appears many times. In modules that contain many calls on allocation functions, this results in substantial wasted storage for the strings that identify the file name. If your compiler permits optimization of multiple occurrences of constant strings, enabling this mode will eliminate the overhead for these strings. Of course, it's up to you to make sure choosing this compiler mode won't wreak havoc on some other part of your program.

Test and Demonstration Program

A test and demonstration program, smtest.c, is supplied with SMARTALLOC. You can build this program with the Makefile included. Please refer to the comments in smtest.c and the Makefile for information on this program. If you're attempting to use SMARTALLOC on a new machine or with a new compiler or operating system, it's a wise first step to check it out with smtest first.

Invitation to the Hack

SMARTALLOC is not intended to be a panacea for storage management problems, nor is it universally applicable or effective; it's another weapon in the arsenal of the defensive professional programmer attempting to create reliable products. It represents the current state of evolution of expedient debug code which has been used in several commercial software products which have, collectively, sold more than third of a million copies in the retail market, and can be expected to continue to develop through time as it is applied to ever more demanding projects.

The version of SMARTALLOC here has been tested on a Sun SPARCStation, Silicon Graphics Indigo^2, and on MS-DOS using both Borland and Microsoft C. Moving from compiler to compiler requires the usual small changes to resolve disputes about prototyping of functions, whether the type returned by buffer allocation is `char *` or `void *`, and so forth, but following those changes it works in a variety of environments. I hope you'll find SMARTALLOC as useful for your projects as I've found it in mine.

Download smartall.zip (Zipped archive)

SMARTALLOC is provided as smartall.zip, a Zipped archive containing source code, documentation, and a `Makefile` to build the software under Unix.

Copying

by John Walker October 30th, 1998

Index