Cyclic Logs: User Manual

Version 0.2

Andrew Peter Marlow

January 1, 2004

Revision	Date	Author	Comments
0.2	01-Jan-2003	Andrew Marlow	minor updates
0.1	29-Dec-2003	Andrew Marlow	First draft

Contents

Chapter 1

Introduction

1.1 The need for cyclic logs

Servers need to record the error, warning and diagnostic output that they produce. There are a number of ways they can do this. Some write to the `syslog` facility but this is only available on POSIX platforms. Other servers write to `stdout` and `stderr`. This is platform-independent but raises the issue of the disk space required to capture the output.

Server logs are typically not examined unless something goes wrong. Old data is often not needed. When there is a problem the log data of most interest is usually what was recorded near to when the problem occurred. This means it is useful to retain recent information but not critical to retain very old information.

What is needed is a platform-independent way for a server to record `stdout` and `stderr` such that the most recent information can be retained in a logfile without such logfiles growing in size such that they might fill the disk up. One way to do this is to merge `stdout` and `stderr` and pipe the lot through a filter that writes all its input to a cyclic log. A cyclic log has a fixed size upon creation and a structure that allows older data to be overwritten on the arrival of new data to be logged.

Suppose the server is called `myserver` and we want to record the output in a cyclic log called `myserver.log`. The command needed is:
`server 2>&1 | alog myserver.log`.
This ensures that the last 10k of log data (the default log size) is always available. To view it, use the command:
`acat myserver.log`.

1.2 Cyclic log commands

The cyclic log commands are:

Command	Description
alog	Create, write or append to cyclic log
acat	Display the contents of a cyclic log to `stdout`
atail	Display the tail of a cyclic log to `stdout`, polling for the arrival of new data every second

1.3 Machine-independent structure

Cyclic logs are files with a special structure: they are created to be of a certain user-specifyable size and never grow beyond that size. They have header information at the beginning to say where the data begins and ends.

Cyclic logs may contain ASCII but on the other hand they may not. It depends what the user wants to put into them. Thus, there is no concept of a record in a cyclic log. it is just a byte buffer that wraps around.

The cyclic log header contains an identifier that identifies the file to be a cyclic logfile. Thus the commands that manipulate cyclic logs can detect when they are being asked to look at a file that is not a cyclic log (in which case they give an appropriate error message).

The cyclic log header contains a revision number. This is used to identify which version of the cyclic log software created the file. This allows the structure of a cyclic log to change in the future whilst retaining backward compatibility.

The numeric data in the cyclic log header such as the revision number and start/end of data pointers, is held in non-machine dependant form. This allows a logfile created on one architecture to be transferred read on a machine with a different architecture.

1.4 Simple API

The methods in the `cyclic_log` class are intended to be analogous to the UNIX routines read and write, which operate on byte buffers. No equivalent to an ostream object is provided for C++ users. There is a good reason for this: ostream objects allow output to be performed for those types that have an output operator defined. Output is thus type-sensitive. This is useful for ostream objects because one can define custom formatting for various types (e.g (x,y) for complex numbers). This is not particularly useful for cyclic logs since cyclic logs are just intended to capture bytes sent to standard output and standard error and it doesn't matter what those bytes represent.

read and write have return values since the analogous UNIX routines also have return values. The return value indicates how many bytes have been read or written. The return value can be used to indicate if something went wrong. The constructor does not have a return value so it throws an exception if something goes wrong.

Chapter 2

Using the cyclic log package

There are three ways to use the cyclic log package:

1. Use the commands (see chapters 3, 4 and 5)
2. Use the C API (see chapter 6)
3. Use the C++ API (see chapter 7)

Use of the either of the APIs require that the programmer observes the license that the cyclic log package is released under. The software licence is the GPL. This means that software that uses the cyclic log API is considered to be a derivative work and must therefore also be released under the GPL.

This policy does not prevent cyclic logs being used in proprietary or commercial projects. Such projects are confined to just using the cyclic log commands. This ought to be sufficient for most projects, since the need is to capture server output for later viewing without running the risk of filling up the available disk space.

Chapter 3

The alog command

3.1 Purpose

Reads from stdin and writes to a named cyclic logfile. The logfile is created if it does not already exist.

3.2 Syntax

```
alog [<options>] <filename>

The following options are supported:

-help          gives brief help
-size <size>   specify size in bytes of logfile to create
-a             append to logfile
```

3.3 Description

The `alog` command reads standard input and writes it to the specified logfile. If the file does not already exist it is created with a default size of 10240 bytes. The size can be specified via the `-size <size>` option. The `-a` option is used to append to the file.

The typical use of this command is to capture the output from a server so that it can be recorded without the output filling up the disk by writing to a conventional logfile that just grows and grows.

Chapter 4

The acat command

4.1 Purpose

Cyclic logs have a special structure which includes a header containing pointers to the first and last bytes of the file. Therefore a special command is needed to write the file contents to `stdout`. This command is `acat`.

4.2 Syntax

```
alog [<options>] <filename>

The following options are supported:

-help          gives brief help
```

4.3 Description

The `acat` command reads from the specified cyclic log and outputs the contents to `stdout`.

Suppose a server pipes its `stdout` and `stderr` through the `alog` filter. The server log will be a cyclic log. The typical use of the `acat` command is to print the log when the server has encountered an error.

Chapter 5

The atail command

5.1 Purpose

The `atail` command is analogous to the `tail` command but for cyclic logs rather than ordinary files. It is used to monitor data being written to the end of the file. The data is displayed as it is detected (polled every second).

5.2 Syntax

```
atail [<options>] <filename>

The following options are supported:

-help          gives brief help
```

5.3 Description

This command is simpler than the tail command. The tail command has many options to control from which point the data is displayed and by default, the tail command does not poll the file (the `-f` option is used for that). However, the `atail` command always polls and always starts reading from the oldest data.

Chapter 6

The C API

The external entrypoints all begin with `CYC_`. There are routines to create a log, open a log, close a log, and read from and write to a log. The open routine returns a descriptor of type `CYC_fd`, which descibes an open file. The file descriptor has to be supplied to the close, read and write routines. A cyclic log cannot be created on the fly (e.g by opening one in write or append mode): the create function must be used.

6.1 Creating a cyclic log

The routine to create a cyclic log is `CYC_create`. The function prototype is:

```
int CYC_create(const char* filename, long fileSize);
```

The file must not already exist. The return value is zero for success and -1 on error. If an error is returned then `errno` contains the error code.

6.2 Opening and closing cyclic logs

The routine to open a cyclic log is `CYC_open`. The function prototype is:

```
CYC_fd* CYC_open(const char* filename,
                 const char* openMode,
                 CYC_ErrorCode* errorCode);
```

Permitted modes are:

Mode	Description
r	read
w	write
a	append
u	synonym for append

The errorCode parameter is the address of an error code where the results of the error will be stored. The routine returns a null pointer in the event of an error.

Unlike other open routines, this open routine requires that the file already exists. If an attempt is made to open a non-existent logfile for writing or appending then an error will occur. This is because the logfile attributes (e.g size) need to be set upon creation. There is no provision for such attribute arguments in the open function.

The routine to close a cyclic log is `CYC_close`. The function prototype is:

```
void CYC_close(CYC_fd* fd);
```

The specified file descriptor must refer to an open cyclic log.

6.3 Reading and writing cyclic logs

A cyclic log file descriptor is needed for the read and write functions. This descriptor is the return value from the `CYC_open` function.

To read a maximum of `buflen` bytes into `buffer`, use the `CYC_read` routine:

```
int CYC_read(CYC_fd* fd, char* buffer, int buflen);
```

The return value is -1 on error, otherwise it is the number of bytes read. A return value of zero indicates end of file.

To write `buflen` bytes of data from `buffer`, use the `CYC_write` function:

```
int CYC_write(CYC_fd* fd, const char* buffer, int buflen);
```

The return value is -1 on error, otherwise it is the number of bytes written.

6.4 Error handling

Most cyclic log routines return zero for success and -1 to indicate an error. Upon an error, errno will indicate the error that occurred. The exception is the `CYC_open` routine. This needs to be able to detect attempts to open files that are not cyclic logs, hence the cyclic package has an error code enumeration. To map the error code to meaningful text, use the `CYC_errorText` function. The prototype is:

```
const char* CYC_errorText(CYC_ErrorCode errorCode);
```

Chapter 7

The C++ API

The class `cyclic_log` is used as the abstraction for a cyclic logfile. There is a static method, `create` to create a cyclic log. The `cyclic_log` constructors are to be used to refer to cyclic logs that have already been created. The destructor closes the cyclic log automatically. The other methods are `read` to read from the log and `write` to write to it.

7.1 Creating a cyclic log

```
static int create(const std::string& filename, long fileSize);
```

Create cyclic logfile with size specified in bytes. Zero is returned on success, otherwise -1 is returned and `errno` is set.

7.2 Opening and closing cyclic logs

7.2.1 Constructors

There are two constructors, one where the filename and open mode are of type `std::string`, the other where the types are `const char*`. Permitted open modes are:

Mode	Description
r	read
w	write
a	append
u	synonym for append

```
cyclic_log(const std::string& filename, const std::string& openMode);
```

```
cyclic_log(const char* filename, const char* openMode);
```

Open logfile for read, write or append. Any file system error encountered during the opening of the file is handled by an exception of type `std::runtime_error` being thrown. The exception is a complete report of what exactly went wrong.

7.2.2 Destructor

The destructor automically closes the open logfile. The destructor is non-virtual because the class is not designed for use as a base class.

7.3 Reading and writing cyclic logs

```
int read(char* buffer, int buflen);

int write(const char* buffer, int buflen);
```

`read` reads a block of bytes from the cyclic log. The return value is the number of bytes read. -1 is returned in the event of an error (errno will then be set).

`write` writes a block of bytes to the cyclic log. The return value is the number of bytes written.

7.4 Error handling

The functions return zero for success and -1 in the event of an error (whereon `errno` is set. The constructors throw an exception of type `std::runtime_error` in the event of an error.

Chapter 8

Design Notes

The cyclic log package is built on the C library. The C++ API is a thin layer on top. This makes the package suitable for either C or C++ programmers.

There are two important structures in the design of cyclic logs: the header that occurs at the start of every log, and the file descriptor that is used to describe an open cyclic log.

8.1 The cyclic log header

The header is exactly 100 bytes long and contains a string that identifies the file to be a cyclic log, followed by a string that indicates the version number. The version number is present in case the structure of a cyclic log ever changes and the cyclic package needs to maintain backward compatibility.

After this follows the start and end offset, including the size of the header. These are also held as strings so that the cyclic log is machine-independent. This means that a cyclic log created on a source machine may be transferred to a target machine that is a different machine architecture or running a different operating system and it will still be interpreted correctly by the cyclic package on the target machine.

The header also contains an "isEmpty" flag, as a string, which is set to "1" to indicate an empty log.

```
typedef struct logfileHeader
{
  /**
   * Identifies the file to be a cyclic logfile
   */
    char id[15];

    /// The version number of the file, as a string.
    char versionStr[15];
```

```
  /**
   * The start offset, including the size of the header
   */
    char startPtr[11];

  /**
   * The end offset, including the size of the header
   */
    char endPtr[11];

  /**
   * True ("1") if the file is empty.
   */
    char isEmpty[11];

  /**
   * Padding bytes to bring the header to 100 bytes.
   */
    char forFutureExpansion[37];
} CYC_LogfileHeader;
```

8.2 The cyclic log descriptor

The cyclic log descriptor is returned from `CYC_open` and is analogous to the `FILE*` returned by `fopen`. It contains a copy of the complete header since the header gets rewritten whenever data is added to the file (because the pointer values change).

The descriptor contains the start and end offsets and the size of the file in bytes. The read position is maintained, the write position is simply the end offset. Flags are also used to remember whether or not a read or write has wrapped. For the reader this is used to detect end of file. For the writer this is used to tell the write position to reset to just after the header. The descriptor also contains the integer file descriptor used in the file routines `read, write, lseek` and `close`.

```
typedef struct logfileDescriptor
{
    /**
     * Copy of the complete header which is rewritten at the start
     * whenever data is added to the file (because the pointer values
     * change).
     */
    CYC_LogfileHeader m_header;

    /**
```

```
     * The file descriptor on which the file is open.
     */
    int m_fd;

    /**
     * The file size in bytes, used to detect the wraparound condition.
     */
    off_t m_fileSize;

    /**
     * The file position at which the data starts.
     */
    off_t m_start;

    /**
     * The file position at which the data ends.
     * New data gets written here and the pointer advanced.
     */
    off_t m_end;

    /**
     * The file position from which to next the next bytes.
     */
    off_t m_readPosition;

    /**
     * True if the reader has wrapped during the read.
     */
    int m_wrapped;

    /**
     * True if the writer has wrapped during the write.
     */
    int m_writeWrapped;

} CYC_fd;
```

8.3 The `cyclic_log` class

The only state maintained by `cyclic_log` objects is the `CYC_fd` file descriptor returned by `CYC_open`. This enables `cyclic_log` to be a very simple wrapper around the C routines.

For example, the constructors just call $\widehat{\text{C}}$YC_open, the destructor calls $\widehat{\text{C}}$YC_close. The `read` and `write` methods just call `CYC_read` and `CYC_write` respectively.

An older design of cyclic logs had the class maintaining all its own state but this was changed in order to offer a C API as an alternative.