

# *The Perl Journal*

## **Eryption Using *Crypt::CBC***

Julius C. Duque • 3

## **dbcoder: Generating Code For Your Database**

David H. Silber • 7

## **An Almanac in Perl**

brian d foy • 10

## **Scraping Yahoo Groups**

Simon Cozens • 13

## **Watching a Logfile in an IRC Channel**

Randal Schwartz • 16

## **PLUS**

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Russell J.T. Dyer:

***Perl Cookbook*** • 19

Source Code Appendix • 21

## LETTER FROM THE EDITOR

### Picking Perl for the Right Reasons

**L**ike a lot of us, I have to defend Perl from those who have come to the conclusion that Perl is somehow passé. I suspect this happens to all languages at some point in their lifetime, especially once they reach a stage of maturity where revolutionary leaps and bounds don't occur with every new release. There seems to be an assumption that if it's no longer in the headlines, it must be dying.

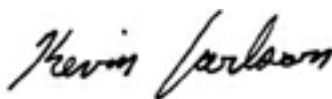
The most disturbing questions are the ones I get from project leaders who have a Perl project planned, but are getting heat from IT execs who have paid consultants to tell them that Perl is out of date. Did we all just switch to the fashion industry? Since when is it a good idea to pick your development tools based on what's "hot"? You pick the tool based on how well it accomplishes the job at hand.

I'm not saying Perl is the perfect tool for every job, although there's a good reason it has been called a "Swiss army knife." If you want to prototype a complex GUI, for example, Perl might not be the best choice. (I know some of you will disagree, but it wouldn't be my first choice.) But if you have textual data to massage or web functionality to build, Perl is a very strong contender, due in no small part to the huge library of preinvented wheels out there that Perl programmers can choose from. If you need to extract and report, then for goodness' sake use the language that was built for extraction and reporting.

I wish IT executive types would take their FUD with a grain of salt. Is Perl less popular than it was? I think that depends on how you measure it. The number of new people adopting Perl may be decreasing, but decreased growth is still growth. And what does that have to do with getting a project finished? If someone makes a case that Perl is going to limit a project in some way, then listen to their case. But none of us wants to work in a company that makes language choices based on the buzzword of the day. That's a Dilbert cartoon come horribly, frighteningly true.

Sometimes, it seems that Perl moves too slowly. By the time Perl got object-oriented features, we were good and ready for them. But the reason that Perl wasn't the first language to go OO is the same reason that it probably won't be the first language to adopt any other new methodology. I'm talking about what the letter "P" in Perl stands for: practical. Above all, it must remain easy to do easy things. And it's not always easy to keep sight of that principle while adding new features to a language. If, for instance, we have to instantiate a class just to do a quick pattern-match substitution, then I think it's fair to say we've lost some practicality. Thankfully, that hasn't happened with Perl. Perl has abstraction when we need it, and a complete lack of abstraction when we don't.

Maybe Perl gets shortchanged for some sort of perceived obsolescence. Frankly, I think reports of its demise have been greatly exaggerated.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to [editors@tpj.com](mailto:editors@tpj.com), faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

**THE PERL JOURNAL** (ISSN 1545-7567) is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



## The Perl Journal

### EXECUTIVE EDITOR

Kevin Carlson

### MANAGING EDITOR

Della Song

### ART DIRECTOR

Margaret A. Anderson

### NEWS EDITOR

Shannon Cochran

### EDITORIAL DIRECTOR

Jonathan Erickson

### COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Randal Schwartz

### CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

### INTERNET OPERATIONS

#### DIRECTOR

Michael Calderon

#### SENIOR WEB DEVELOPER

Steve Goyette

#### WEB DEVELOPER

Bryan McCormick

#### WEBMASTERS

Sean Coady, Joe Lucca

### MARKETING / ADVERTISING

#### PUBLISHER

Timothy Trickett

#### MARKETING DIRECTOR

Jessica Hamilton

#### GRAPHIC DESIGNER

Carey Perez

### THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403  
650-513-4300. <http://www.tpj.com/>

### CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, TECHNOLOGY SOLUTIONS Robert Faletta

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenber

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER ENTERPRISE

ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING

SOLUTIONS Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES

Marie Myers

# Perl News

## Now a Contest Comes The Brief and Useful Programs Turn to Poetry

ActiveState has announced a Perl Haiku Poetry Contest in two categories: valid Perl programs that adhere to the three-line haiku format, and haikus written in English on the subject of “Why I Love Perl.” Though it will be too late by the time you read this to dash off a submission (the deadline for entries was February 8), all the entries will be posted on the ActiveState web site for the enjoyment of the community. Winners will be judged by the ActiveState development team “on the basis of originality, creative imagination, characterization, artistic quality, and adherence to line limits.” First place winners in each category will receive licenses for ASPN Perl, including Komodo Professional Edition; second place winners will take “an ActiveState prize package” valued at \$50; and third place winners get ActiveState T-shirts. See <http://activestate.com/Corporate/PerlHaiku/> for full details.

## CPAN Gets Tools

Sam Tregar has added *diff* and *grep* functionality to CPAN. If you go to a distribution page, you will notice an addition to the “Links” section: an entry for “Tools.” Clicking the tools link allows you to run *diff* and *grep* on the module’s source code. “I’d like to thank Graham Barr for allowing me the opportunity to add these features,” Tregar wrote on [use.perl.org](http://use.perl.org), “and, of course, for creating and maintaining [search.cpan.org](http://search.cpan.org). CPAN just wouldn’t be the same without it!”

## The Year in Perl

The Perl conference schedule for 2004 is starting to fill up. A Dutch Perl Workshop has been announced for March 5 in the Netherlands; the talks will all be in Dutch, which “may help Dutch speaking people to absorb information faster than from books,” as organizer Mark Overmeer notes, but “also makes the meeting useless for people who do not understand Dutch.” So if <http://workshop.perlpromo.nl/> is Greek to you, then you might want to skip this one.

YAPC North America has issued a call for participation (<http://yapc.org/America/cfp.shtml>). The conference, which is now in its sixth year, will be held in Buffalo, New York, June 16–18.

Standard talks will be 20 minutes, but provisions have also been made for 5-minute lightning talks, as well as long or extra-long talks, and 3-hour tutorials. A display of poster presentations will also be set up in the main hallway.

The German Perl Workshop, scheduled for June 29–July 1 near Stuttgart, is also in its sixth year. The call for papers advises that “the conference language is mainly German, but you can easily present your talk in English if German is not your native tongue.” Proposals for short talks of 15 to 20 minutes, long talks of 40 minutes, and half-day tutorials are all welcomed. For more details, see <http://www.perlworkshop.de/2004/docs/cfp.html>.

Finally, if the whirlwind of Perl activity around the globe leaves your head spinning, Dave Cross’ Perl conference list at [http://dave.org.uk/perl\\_conf/](http://dave.org.uk/perl_conf/) may help to sort it all out.

## Perl 5.8.3 is Out

Nicholas Clark released Perl 5.8.3, “a maintenance release for Perl 5.8, incorporating various minor bugfixes, including eliminating a couple of errors in Perl’s UTF8 handling,” after a single release candidate. A *SCALAR* method is now available for tied hashes, and *find2perl* now assumes *-print* as a default action. Perl 5.8.4 should be out some time in April; after that, further 5.8.x releases will follow a roughly quarterly schedule.

## Fotango Continues to Support Perl Development

Quick on the heels of last month’s news that the UK consultancy group Fotango is sponsoring Andy Wardley in his work on Version 3 of the Template Toolkit, Fotango has announced that it will also provide a grant to the Perl Foundation to support Abhijit Menon-Sen’s work on new profiling tools for Perl. The donation will fund Menon-Sen over the next 10 months while he develops “a generalized instrumentation framework for the Perl programming language.”

Fotango hosts a page explaining the company’s interest and investment in open-source projects at <http://opensource.fotango.com/>. “Fotango started life as an Online Photo album and in fact, still operates as one, despite the dot.com crash,” the site explains. “In 2001 Canon Europe acquired Fotango, and this made it possible for us to keep our team together and become much more active as a company in the open-source community.”



# Encryption Using *Crypt::CBC*

There are two general methods of scrambling information: public-key encryption and symmetric encryption. Symmetric encryption is divided further into two major classes: stream cipher and block cipher. Of these two, the block cipher is the more popular, due mainly to the immense attention given to the Data Encryption Standard (DES), the most popular encryption algorithm in the world. Designed by IBM, with the aid of the NSA in the early 1970s, DES has been the focus of numerous studies by many mathematicians since its adoption as a Federal Standard in 1977. The lessons learned from these analyses spawned countless other block ciphers that bear many of DES's features.

## Some Block Cipher Theory

Using a block cipher, data to be encrypted (also called plaintext) is first divided in chunks (or blocks) of equal sizes. The size corresponds to the block size of the cipher. For DES, data is encrypted in chunks of 64 bits, while the Advanced Encryption Standard (AES) candidates divide data in blocks of 128 bits. Next, the first plaintext block is encrypted, using a secret key, to produce the first encrypted block (also known as the "ciphertext block"). Next, the second plaintext block is processed to create the second ciphertext block, and so on. When all plaintext blocks have been encrypted, the ciphertext blocks are strung together in the order that they were created to form the final encrypted data, called "ciphertext." This mode of operation is known as Electronic Code Book (ECB). See Figures 1 and 2 for visual descriptions of ECB mode.

As you can see from the diagrams, encryption/decryption in ECB mode need not be done sequentially. In fact, since the blocks are processed independently of each other, they are best encrypted/decrypted in parallel.

There are problems in the block cipher, however. First, plaintexts don't always divide neatly in exact multiples of the block size. The last plaintext block may end up shorter than the block size. This is a problem of all block ciphers. The second problem concerns block ciphers in ECB mode: Identical plaintext blocks always produce identical ciphertext blocks. This is a problem because a cryptanalyst can create a code book of plaintexts and corresponding ciphertexts. If he learns, for example, that the plaintext block 0x4e6a89fb encrypts to ciphertext block 0x75f0c3d1,

he can immediately decrypt that ciphertext block whenever it appears in another message.

But here's the clincher: If a certain block cipher has a block size of, say, 64 bits, the code book will have  $2^{64}$  entries—much too large to precompute and store, even with today's technology. And that's just for one key only. Now, if that block cipher uses a 128-bit key, a total of  $2^{64} \times 2^{128}$  code book entries will have to be made. Given this immense storage requirement, it is practically certain that no storage device will ever be built to hold this huge code book. So, it turns out, this problem is purely theoretical. Yet, if you're the paranoid type, you'd be happy to know that there is a Perl module that avoids this ECB weakness.

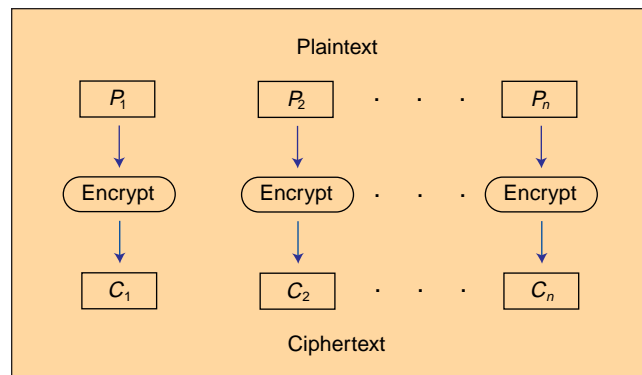


Figure 1: Encryption in ECB mode.

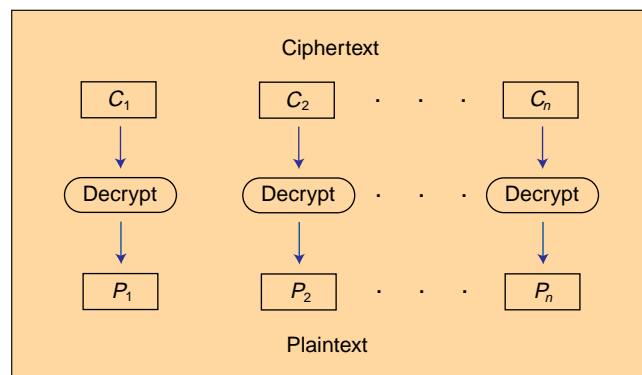


Figure 2: Decryption in ECB mode.

*Julius is a freelance network consultant in the Philippines. He can be contacted at [jcdunque@lycos.com](mailto:jcdunque@lycos.com).*

For the first problem, the solution is to pad the last plaintext block with some regular pattern—zeros, ones, alternating ones, and zeros—to make it a complete block. Figure 3 illustrates a typical padding scheme, the so-called PKCS #5. In the figure, assuming that the underlying block cipher has a block size of 64 bits (8 bytes), the partial block (consisting of five blue squares) is extended by padding it with three ASCII 0x03 characters.

For the second problem, the most common solution is to XOR (symbol:  $\oplus$ ) the current plaintext block, bit by bit, with the previous ciphertext block before it is encrypted. This way, identical plaintext blocks will not encrypt to identical ciphertext blocks. Of course, plaintexts that start the same will still encrypt to the same ciphertexts up to the first difference. To prevent this, encrypt a random block of data (called an “initialization vector” or IV), and treat it as the first ciphertext block. An IV doesn’t have to be meaningful; it’s only used to make each message unique. This mode of operation is known as Cipher Block Chaining (CBC). Figures 4 and 5 are visual descriptions of CBC mode.

CBC mode is not without its own drawbacks, however. Some of these problems are:

- The IV must be unique for every message if you wish to encrypt with the same key, and the IV must not be reused. These factors increase the complexity of implementation.
- Encryption cannot be done in parallel because the previous ciphertext block must be computed first before the current plaintext block can be processed (although decryption is parallelizable—see Figure 5).

## Perl Implementation of CBC

Lincoln Stein’s *Crypt::CBC* module is a pure Perl implementation of CBC (<http://www.cpan.org/authors/id/L/LD/LDS/Crypt-CBC-1.00.readme>). Its only requirement is that the underlying block cipher must be able to report the block size and key size it’s using when *Crypt::CBC* asks for them. Lincoln, however, made some exceptions. He accommodated Blowfish, a strong, time-tested block cipher that uses variable-length key sizes, even though it’s unable to report its own key size. When Blowfish is used, *Crypt::CBC* defaults to Blowfish’s maximum key size of 448 bits (56 bytes). On the other hand, if no block cipher is specified, *Crypt::CBC* defaults to DES.

I’ll illustrate how *Crypt::CBC* works by using it in working Perl scripts. The scripts presented here use the block ciphers Khazad and Serpent, both available from CPAN as *Crypt-Khazad* and *Crypt-Serpent*, respectively. If you wish to run these scripts, please download and install these block ciphers first.

The first Perl script, *khazad*, shown in Listing 1, shows how to encrypt simple messages. Notice that our preferred block cipher module, *Khazad*, is not *used* anywhere in the script. That’s because *Crypt::CBC* automatically loads it for us.

As with any object-oriented way of doing things, the first order of business is to instantiate an object. This is done by calling

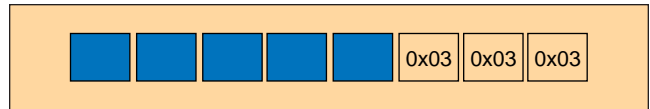


Figure 3: Padding of partial block using PKCS #5.

the *new()* function of *Crypt::CBC* (Line 10). Additionally, six hash keys need to be set before *Crypt::CBC* goes to work: *key*, *cipher*, *iv*, *regenerate\_key*, *padding*, and *prepend\_iv*. These are set in lines 10–15.

**key.** This is the password or passphrase used to encrypt or decrypt a message or file. If you’re using a block cipher that takes, for example, a 128-bit (16-byte) key (or whatever length), you must ensure that you supply a key exactly that long. A key that’s too short or too long would produce an error similar to this:

```
Uncaught exception from user code:
Key setup error: Key must be 16 bytes long! at
/usr/local/lib/perl5/site_perl/5.8.0/Crypt/CBC.pm line 95.
Crypt::CBC::new('Crypt::CBC','HASH(0x8116654)') called
at ./cbc-mode line 11
```

To take care of this problem, set the hash key *regenerate\_key* to a nonzero value. See line 13.

**regenerate\_key.** When *regenerate\_key* is set to a nonzero value, *Crypt::CBC* hashes the supplied key using MD5, a strong one-way hash function that produces a fixed, 128-bit message digest. *Crypt::CBC* then uses this message digest (a 16-byte binary string) as the actual key for the encryption.

If the underlying block cipher requires a longer key, *Crypt::CBC* hashes the 16-byte key, thereby producing another 16-byte string, and appends it to the old 16-byte key. The resulting 32-byte key is again hashed, producing another 16-byte string, and appends it to the previous 32-byte key. This hashing process is repeated as often as needed until the final string is at least as long as the cipher’s key size. The final string is then truncated to the desired length if it exceeds the required key size. For block ciphers with key sizes less than 128 bits (DES, for example, uses only 64 bits), the hashes of their keys are just truncated to the required lengths.

On Line 7, it is perfectly all right to use a key of arbitrary length because *regenerate\_key* is set to 1.

On the other hand, if *regenerate\_key* is set to 0, *Crypt::CBC* uses the user-supplied key as is. Just make sure that your key conforms to the key size of your block cipher.

**cipher.** This is the name of the block cipher to be used. There’s no need to use it in your script because *Crypt::CBC* automatically loads it for you.

**iv.** The IV must be as long as the block size. Line 8 uses a static IV, which is really not a good idea.

**padding.** When padding an incomplete plaintext block, *Crypt::CBC* employs a variety of padding options. These are:

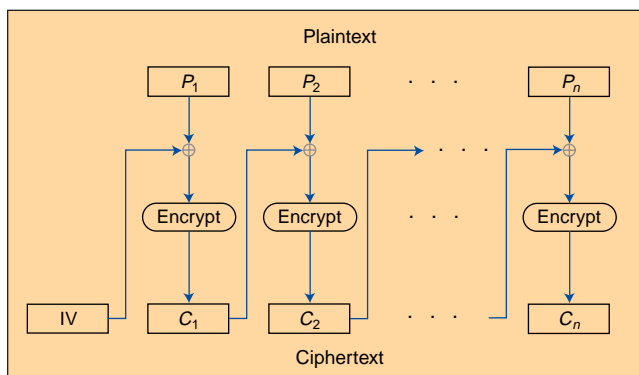


Figure 4: Encryption in CBC mode.

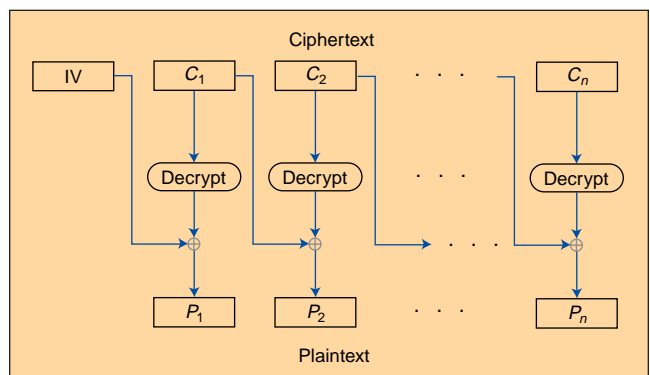


Figure 5: Decryption in CBC mode.

- Null padding  
Usage: 'padding' => 'null'  
Pad the last plaintext block with as many ASCII 0x00 characters necessary to fill the block. If the last block is a full block, an extra block of all ASCII 0x00 characters is still appended. See Figure 6 for a diagram of null padding. This padding option is recommended only for encrypting text data.
- Space padding  
Usage: 'padding' => 'space'  
Same as null padding, but uses ASCII 0x20 instead. An extra block containing all ASCII 0x20 characters is still appended if the last block happens to be complete already. See Figure 7 for a diagram of space padding. This padding option is recommended only for encrypting text data.
- One and zeroes padding  
Usage: 'padding' => 'oneandzeroes'  
Pad the last plaintext block with one ASCII 0x80 character followed by zero or more ASCII 0x00 characters necessary to fill the block. If the last block is already full, append an extra block. This extra block starts with ASCII 0x80 followed by as many ASCII 0x00 characters necessary to complete a block. See Figure 8 for a diagram of one and zeroes padding. The name oneandzeroes comes from the fact that 0x80 is 10000000 in binary representation. The last block is appended with a “1” bit followed by as many “0” bits as necessary to complete a block. This padding option is recommended for all types of data.
- Standard padding  
Usage: 'padding' => 'standard'  
This padding scheme is also known as PKCS #5. An example would clarify this padding option. Suppose that the last block is 5 bytes short of becoming a full block. With this padding scheme, append five ASCII 0x05 characters to that block. See Figure 3 for another visual description of this padding scheme. If, on the other hand, the block happens to be complete already, we still append an extra block. If the block size is, say, 8 bytes, this extra block contains eight ASCII 0x08 characters. See Figure 9 for a diagram of standard padding. This padding option is recommended for all types of data. If no padding method is specified, standard is assumed.

**prepend\_iv.** When *prepend\_iv* is set to a nonzero value, the resulting ciphertext will start with a 16-byte prefix. The first eight characters consist of the string “RandomIV” followed by another eight-character string. During decryption, if *Crypt::CBC* sees the regular expression */RandomIV{8}/*, the eight characters following “RandomIV” will be used as the IV. Obviously, this is unacceptable for block ciphers with block lengths longer than eight bytes. New block ciphers, like Rijndael, Serpent, RC6, Twofish, and Anubis, use block sizes of 16 bytes. To disable this feature, set *prepend\_iv* to 0.

## A Sample File Encryption Script

The next Perl script illustrates the use of Serpent, a 128-bit block cipher that uses a 128-bit key. The whole Serpent script, appropriately called *serpent*, is shown in Listing 2.

Suppose that you want to encrypt the file *sample.txt*. To encrypt, type

```
./serpent -encrypt sample.txt > sample.enc
```

You will be asked to enter a password twice. When this is done, a new file, *sample.enc*, is created. To decrypt *sample.enc*, type

```
./serpent -decrypt sample.enc > sample.dec
```

Just like in the preceding example, you will be asked to enter a password twice. After this, the decrypted file, *sample.dec*, is cre-

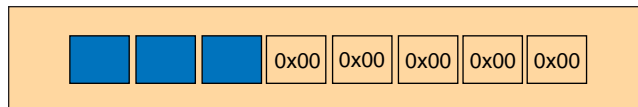


Figure 6: Null padding of a partial block.

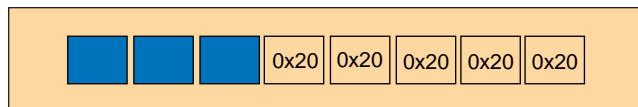


Figure 7: Space padding of a partial block.

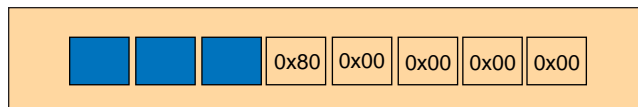


Figure 8: One and zeroes padding of a partial block.

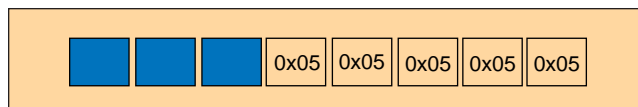


Figure 9: Standard padding of a partial block.

ated. The original file, *sample.txt*, should be identical to *sample.dec*.

In this script, the IV is generated by reading */dev/urandom* (line 8, and lines 47–49), and the IV is saved for future use in the file *iv.rand* (line 9, and lines 50–52). *get\_input()* (lines 16–37) simply asks for a key twice while *echo* is switched off.

Taking a string argument, *start()* (lines 62 and 77) is a *Crypt::CBC* method that initializes the encryption or decryption process. For encryption, the argument must be “E” or any word that begins with “e.” For decryption, the argument must be “D” or any word that begins with “d.”

During decryption, the file *iv.rand* is read to retrieve the IV used in previous encryption (lines 66–68).

Lines 80–82 tell *Crypt::CBC* to call the method *crypt()* for as long as there is data to be processed. The *crypt()* method performs the encryption and should be called after *start()*.

Finally, on line 85, the method *finish()* is called to process the last plaintext block.

## Creating a *Crypt::CBC*-Compliant Module

Suppose you’d like to make your own block cipher module that is *Crypt::CBC*-compliant. Your block cipher must be able to return the block size it is using, as well as the length of its key when *Crypt::CBC* asks for them. Suppose that your block cipher is named *MyBlockCipher*, and uses a block size of 64 bits (8 bytes) and a key size of 128 bits (16 bytes), and further suppose that you’d like to use XS programming. Edit your *.xs* file and add the following lines:

```
int
keysize(...)
CODE:
    RETVAL = 16;
OUTPUT:
    RETVAL

int
blocksize(...)
CODE:
    RETVAL = 8;
OUTPUT:
    RETVAL
```

Your .xs file should now contain the following:

```
1 #include "EXTERN.h"
2 #include "perl.h"
3 #include "XSUB.h"
4 #include "ppport.h"
5
6 /* some code here */
7
8 MODULE = Crypt::MyBlockCipher    PACKAGE = Crypt::MyBlockCipher
9
10 int
11 keysize(...)
12     CODE:
13         RETVAL = 16;
14     OUTPUT:
15         RETVAL
16
17 int
18 blocksize(...)
19     CODE:
```

```
20         RETVAL = 8;
21     OUTPUT:
22         RETVAL
23
24 /* some more code here */
```

The keyword *int* must be on a line by itself. *RETVAL* stands for “return value,” and this is the data that *Crypt::CBC* needs. Read the *perlxs* man pages for more details.

In the .xs file above, the *keysize()* function returns a value of 16 bytes (the 128-bit key), while *blocksize()* returns 8 bytes (the 64-bit block length).

## References

*Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Second Edition. Bruce Schneier, John Wiley & Sons, 1997, ISBN 0-471-12845-7

*Crypt::CBC* Module, Lincoln D. Stein, <http://www.cpan.org/authors/id/L/LD/LDS/Crypt-CBC-1.00.readme>.

**TPJ**

(Listings are also available online at <http://www.tpj.com/source/>.)

### Listing 1

```
1 #!/usr/local/bin/perl
2 use diagnostics;
3 use strict;
4 use warnings;
5 use Crypt::CBC;
6
7 my $key = "hello, there!";
8 my $IV = pack "H16", "0102030405060708";
9
10 my $cipher = Crypt::CBC->new({'key' => $key,
11                               'cipher' => 'Khazad',
12                               'iv' => $IV,
13                               'regenerate_key' => 1,
14                               'padding' => 'standard',
15                               'prepend_iv' => 0
16                               });
17
18 my $plaintext1 = pack "H32", "0123456789abcdeffedcba9876543210";
19 print "plaintext1 : ", unpack("H*", $plaintext1), "\n";
20
21 my $ciphertext1 = $cipher->encrypt($plaintext1);
22 print "ciphertext1 : ", unpack("H*", $ciphertext1), "\n";
23
24 my $plaintext2 = $cipher->decrypt($ciphertext1);
25 print "plaintext2 : ", unpack("H*", $plaintext2), "\n";
```

### Listing 2

```
1 #!/usr/local/bin/perl
2 use diagnostics;
3 use strict;
4 use warnings;
5 use Getopt::Long;
6 use Crypt::CBC;
7
8 my $SRC_RANDOM = "/dev/urandom";
9 my $IV_FILE = "iv.rand";
10 my $BLOCKSIZE = 16;
11 my $BLOCK_CIPHER = "Serpent";
12 my $IV;
13 my ($encrypt, $decrypt) = ();
14 GetOptions("encrypt" => \$encrypt, "decrypt" => \$decrypt);
15
16 sub get_input
17 {
18     my ($message) = @_;
19     local $| = 1;
20     local *TTY;
21     open TTY, "</dev/tty";
22     my ($key1, $key2);
23     system "stty -echo </dev/tty";
24     do {
25         print STDERR "Enter $message: ";
26         chomp($key1 = <TTY>);
27         print STDERR "\nRe-type $message: ";
28         chomp($key2 = <TTY>);
29         print STDERR "\n";
```

```
30         print STDERR "\nThe two $message, 's don't match. ",
31             "Please try again.\n\n" unless $key1 eq $key2;
32     } until $key1 eq $key2;
33
34     system "stty echo </dev/tty";
35     close TTY;
36     return $key1;
37 }
38
39 my $key = &get_input("password");
40
41 chomp $ARGV[0];
42 open INFILE, "<$ARGV[0]";
43
44 my $cipher;
45
46 if ($encrypt) {
47     open RANDSRC, "<$SRC_RANDOM";
48     read(RANDSRC, $IV, $BLOCKSIZE);
49     close RANDSRC;
50     open SRC_IV, ">$IV_FILE";
51     print SRC_IV $IV;
52     close SRC_IV;
53
54     $cipher = Crypt::CBC->new({'key' => $key,
55                               'cipher' => $BLOCK_CIPHER,
56                               'iv' => $IV,
57                               'regenerate_key' => 1,
58                               'padding' => 'standard',
59                               'prepend_iv' => 0
60                               });
61
62     $cipher->start('encrypt');
63 }
64
65 if ($decrypt) {
66     open RANDSRC, "<$IV_FILE";
67     read(RANDSRC, $IV, $BLOCKSIZE);
68     close RANDSRC;
69
70     $cipher = Crypt::CBC->new({'key' => $key,
71                               'cipher' => $BLOCK_CIPHER,
72                               'iv' => $IV,
73                               'regenerate_key' => 1,
74                               'padding' => 'standard'
75                               });
76
77     $cipher->start('decrypt');
78 }
79
80 while (read(INFILE, my $buffer, 1048576)) {
81     print $cipher->crypt($buffer);
82 }
83
84 close INFILE;
85 print $cipher->finish;
```

**TPJ**



# dbcoder: Generating Code For Your Database

This article will discuss dbcoder, a tool that generates code based on a MySQL database schema and a code *template*. It can be used, for example, to generate a set of Perl classes to provide object-oriented access to a database.

Information about an arbitrary database's schema can be embedded into the generated code.

## Usage

```
$ dbcoder -database=payroll -template=program
```

will produce a version of the *program* template customized to match the structure of the *payroll* database. This presumes that there is an available template called “program” and a database named “payroll.” A selection of useful and demonstration templates are included as part of the dbcoder distribution package. In addition, users may write their own templates to fulfill any particular need.

One of the primary products of dbcoder is code generated from the “Perl” template. This generates a set of Perl-language modules that provide classes specific to the named database and its tables. The “Perl” template is distributed as part of the dbcoder package.

dbcoder currently works with MySQL databases. I have started work toward support for PostgreSQL databases, but have since deferred that project to make time for other dbcoder enhancements.

Even though dbcoder is written in Perl, the templates can be written in any language—in fact, if a schema report is all you want, dbcoder can generate that from a template as easily as it generates program code. Note that dbcoder users do not have to know the details about how templates are written. End users can use one of the templates provided with the dbcoder software distribution, or templates written by others within their organization or by third parties. dbcoder will search out templates from various places on the local computer and can produce reports regarding them upon request.

## Basic Template Structure

Each template resides within a directory containing at least an information file and a template subdirectory. The information file contains meta-information, which describes the template. The

---

*David writes custom software and lives in the Washington, D.C. area. He can be contacted at david@SilberSoft.com.*

name of the top-level directory is the default name of the template. The template subdirectory contains the actual template files themselves.

Within the template files, a set of data is made available by means of *template variables*, which need only be mentioned within a set of special delimiters. For example, the following line in a template:

```
# This code generated for use with the [[ Database.Name ]] database.
```

could result in this similar line of generated code:

```
# This code generated for use with the payroll database.
```

Information more complicated than a simple variable can also be referenced. For example, this template snippet produces a list of tables in the database:

```
# This database contains the following tables:
[[ FOREACH table = Database.Tables ]]
#   - [[ table.Name ]]
[[ END ]]
```

Code is normally evaluated in “database context.” Within the above loop, information is evaluated within the context of the current table. It is possible to set up a template file to be processed in “table context,” in which case that template file is used as the pattern for one generated file per database table.

As an example of how dbcoder might be used, let us say that you have a class for your program that (among other things) accesses data from a specific database table. (This would be generated in table context, as otherwise, there is no value that can be set for the top-level Table variable.)

You might have something like:

```
Package [[ Database.Name ]]:[[ Table.Name ]];
.
.
.

[[ FOREACH column = Table.Columns ]]
sub get[[ column.Name ]] {
...
}
```



```

}
[[ END ]]

```

This would result in a set of files of generated code similar to:

```

Package payroll::timesheetentry;
.
.
.

sub getemployeeid {
...
}

sub getprojectid {
...
}

sub getstarttime {
...
}

sub getduration {
...
}

sub getgoal {
...
}

sub getdone {
...
}

sub getcomment {
...
}

.
.
.

```

But this example is pretty boring. Suppose you wanted to make the accessor methods that deal with strings pass their results through a translation function before returning them. You could make a template that includes:

```

[[ FOREACH column = Table.Columns ]]
sub get[[ column.Name ]] {
...
[[ IF column.Type == DBCoder.ColumnType.String ]]
return babblefish( $result );
[[ ELSE ]]
return $result;
[[ END ]]
}
[[ END ]]

```

This would result in generated code similar to:

```

Package payroll::timesheetentry;
.
.
.

sub getemployeeid {
...

```

```

return $result;
}

sub getprojectid {
...
return $result;
}

sub getstarttime {
...
return $result;
}

sub getduration {
...
return $result;
}

sub getgoal {
...
return babblefish( $result );
}

sub getdone {
...
return babblefish( $result );
}

sub getcomment {
...
return babblefish( $result );
}

.
.
.

```

The possibilities are limited only by the imagination of the template author.

## Directory Variables

The variable data provided to templates is also available to be used within the file names of generated files. The file names of the template files need to be constructed with underscores surrounding the name of the desired variable in the place where the variable values should be substituted.

The use of the variable name *Database.Tables* marks the template file to be evaluated in table context. If the variable name is used in a directory's name, all files within that directory are likewise evaluated in table context.

## Parameters

The template author can specify a set of parameters that will be recognized by the template. Parameters are declared (and generally defined) in a parameter's file in the top-level directory of a template. A parameter can be referenced from within a template just like any built-in template variable, except that parameters are contained in simple variables with only one value.

End users may choose to set values for these parameters, and the files they generate from templates that use the named parameters will include that information.

## The Perl Template Toolkit

Originally, templates contained embedded Perl code and the db-coder program used *eval* to process that code. The intention was to leverage the power of the Perl *eval* statement to evaluate code with database information embedded in Perl code.

While this was a nifty hack, it had some major problems. It meant that the templates had to be written by someone who was fluent in Perl—even when the language of the template code was something other than Perl. It also not only exposed the internals of dbcoder to the templates, but it required that template authors be very knowledgeable about those internals. This made it very difficult to change the dbcoder program. Almost any part of it might be used by a template that would then be broken if that part changed. Finally, the template code itself was ugly. It was necessary for the template author to protect any text that should not be evaluated by the dbcoder program. Perl programs in particular tended to turn into exercises in creative string escapes.

But then we discovered the Perl Template Toolkit. Template Toolkit is a CPAN module that encapsulates the general idea of processing a template file and generating a result file. It allows for a set of variable information to be provided to the template. It was not a hard decision to use it instead of the dbcoder template-parsing code.

This brought us a variety of benefits:

- The dbcoder code base is reduced in size, which eases maintenance of the dbcoder project.
- The template syntax became much more regular.
- Many users will already be familiar with the syntax.
- While we could have again exposed the internal dbcoder objects to the template, we chose instead to provide the Template Toolkit (thus, the templates) with a very specific set of data. The interface to the templates is now totally separate from the internal layout of dbcoder. This opened the way for a lot of code cleanup, which in turn led to a less cluttered and more understandable code base.

## Data Collection

Option setting can come from the command line or from a user's project file. If the user has set parameter values, they are collected from the project file as part of this process. dbcoder collects metadata about all the templates that it can find. Templates are searched for in a *template path*, which by default contains *\$HOME/dbcoder* and the directory under which the templates distributed with dbcoder were installed.

This collection of data is searched to find the requested template(s). If dbcoder was invoked to produce a report instead of to generate code, it might be that all of the templates' data will be used.

The template metadata is made available to the template by way of the *template* variable. Specific data within the template information will be available through variables such as *Template.Author* or *Template.Version*. The full set of template metadata can be found in the "dbcoder Template Authors' Reference."

The complete tree of database information (database, table, column) is collected from the database engine. This information includes the name of each entity, their relative position within the data hierarchy, and—for columns—their type. Relation and index information are read from the project file. (When dbcoder was written, MySQL did not support references.)

## Generating Code from Templates

When it is time to generate result files, the template files from each requested template are processed. (The metainformation was already collected early on.) The hierarchy of template files is traversed depth-first. Each template file is used to generate a result file with the same name. Certain characters embedded in the template file name can be used to substitute information from the template variables. If that information is the name of a table, the contents of that template file or directory are used to generate one result file or directory for each table in the database.

For the most part, template files are processed in database context, but if the template is being processed for a specific table (table

context), the information about that table is made available as a top-level template variable.

When each template file is handed off to the Template Toolkit for processing, the name for the target file (possibly modified to contain variable data) and the set of templates variables data is also provided. Variable information includes dbcoder metainformation (version, timestamp of this invocation, etc.), template metainformation (author, version, versions of dbcoder it can work with, etc.), database schema information, and parameters.

## Lessons Learned in Development

**Testing.** dbcoder was conceived and written by John Romkey in 1997. I took over the maintenance and further development (under John's supervision) in January 2001. Coming in to the dbcoder project, I was confronted with a large chunk of code that had some problems and for which we had many goals. After studying the code for a while, I could see that I needed some way to test the existing functionality. There was a "t" directory, which is commonly used for testing by Perl modules, but it had a series of empty files and one file with a few lines of code in it. There was no documentation, and I was unfamiliar with that testing setup. Rather than get side-tracked learning that testing methodology, I chose to create my own tests.

In retrospect, I think that was only a partial mistake. On the one hand, I should have gone with the standard test setup. On the other hand, many of the tests I wrote to fit into my specially constructed test framework would not have worked in the standard test framework. I likely could have saved myself a great deal of effort, though, by using the standard test framework for the simpler tests.

**Language Features.** As I was wrapping up work on the dbcoder project, I realized that I had not taken as much advantage of inheritance as I could have. The *MySQL* and *PostgreSQL* classes should inherit from *Database*, *Table* and *Column* should perhaps inherit from an *Element* class, and *TemplateList* and *ParameterHash* should descend from a common *List* ancestor.

**Documentation.** I put a lot of effort into the dbcoder documentation. I added POD to each module so that manual pages could be generated. The POD was interspersed with the code rather than being stuck at the bottom so that it also served as the per-subroutine headers. This worked out well, as documentation that became out of date tended to get noticed and fixed. I think that if the documentation had been separate it would have tended to drift much more than it did.

In addition to the program code documentation, I wrote a set of books targeted to the various roles people take on as they use software. In this case, the roles are "End User," "Administrator," "Template Author," and "dbcoder Developer." I also turned the POD documentation into sections in the appendix of the Developers' Manual. I think this was a good thing for the program modules. It was a bit of overkill, though, to do the same for the test scripts. For a project with more people working on it, that might have made more sense. The books were written in DocBook, which was a good choice, I think, as there are a good assortment of tools for it.

All told, I think the work was successful. The project is in a much better state now than when I started and there is documentation to show how to use the program and to enlighten the next person to work on it, or even to remind me, should I come back to it after a long absence.

More information regarding dbcoder may be found at <http://www.dbcoder.org/>.

# An Almanac in Perl

*brian is currently on active duty with the United States Army as a military policeman in Iraq. You can e-mail him at [comdog@panix.com](mailto:comdog@panix.com), if you don't mind waiting a couple months for a reply.*

I have been living in the middle of the desert for several months while on active duty with the U.S. Army and, occasionally, we take advantage of our night vision technology. In any conflict, the ability to see at night has been important. Night vision can be limited by weather and climate effects such as cloud cover or fog, but over here it is mostly affected by the phase of the moon. We consider that full or new moons give off 100 percent illumination, a half-moon 50 percent, and so on; and whenever we plan any night operation, we try to schedule it for the lowest illumination possible to make maximum use of our night vision devices. This is nothing new—soldiers in any war realize that the darker it is, the easier it is to sneak around unseen. All of this information is available in almanacs, but I am going to create my own since Perl makes it so easy.

The lunar cycle is 28 days, so the same phase comes at different days of each month. If I marked a known phase on my calendar, I could count the right number of days to the next phase, and through that process, create my own almanac. Indeed, you can buy such books. That way, however, does not let me play with Perl and, eventually, I want to create a program to give me a summary of the day's information.

In Listing 1, lines 3 to 6 pull in the modules I need. The `Astro::MoonPhase` module does most of the hard work for me, and some of the Time and Date modules handle the rest. I want to make a graph of luminosity as a function of the date. The `GD::Graph` module takes care of visuals as long as I come up with the data.

Lines 8 and 9 look at the command-line arguments to get the time zone offset and year for which I want to create the graph; otherwise, I assume I am in Greenwich during the current year. On line 12, I do a little trickery to get the first Sunday in the year

because I want that to be my first data point. When I label the dates on the graph, I will make a mark for each Sunday. The seventh element that `localtime()` returns gives me the day of the week, starting at 0 for Sunday, so I keep looping through the days of the year until `localtime()` gives me a false value, taking into account the time zone difference.

On line 22, I get the latest time in the year for which I want to calculate the luminosity, and I will use that as a limit when I start generating data points. The `for()` loop on line 25 is the interesting work. I start at the first Sunday time, stored in `$now`, and go to the end of the year, stored in `$then`, and I create a data point every three hours. I keep the list of times in `@times` and the luminosity in `@illum`. I could have put these in a hash, but I am going to take them right back out to plot them, so I just use named arrays.

On line 35, I create the `@data` array to pass off to `GD::Graph`. The `GD::Graph` modules expect the data points to be in an array of arrays. The first anonymous array element is the horizontal axis value and the rest are the vertical axis values.

On line 37, I create a new graph object by telling `GD::Graph` that I want a line graph that is 800 pixels wide and 300 pixels long. On line 39, I set the options for the graph. The interesting options include the `x_number_format`, which I give an anonymous subroutine to translate the time data to an English date, and the `x_tick_number`, which tells `GD::Graph` to make 52 x ticks, one for each week (remember that I started at the first Sunday). The rest of the options specify various colors and labels.

Once I specify what I want, on line 58, I tell `GD::Graph` to make it happen by plotting the data. `GD::Graph` creates the graph and puts everything where it should be (if I have set my options right—sometimes I have to tweak it a bit). Although `GD::Graph` created the graph, it did not actually give me an image. On line 61, I tell the module that I want the graph as a PNG image and I save it to a file.

Once I run the program, I open the image to make sure it is what I want, but it usually is not, so I change a few `GD::Graph` options and try again. You can play with the options yourself to make the image pleasing to you if you do not like my choices. The `GD::Graph` module can do quite a bit more than I have shown.

I want to know all sorts of other things, too. I want to know the various levels of sunrises and sunsets. The sun's relationship to

---

*brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com.*

the horizon defines the different types. The civil twilight is when it becomes too dark to read outside, and that is when we like to start moving around at night.

For my almanac, I want to use Perl's formats, which saves me a lot of work in displaying the data and making it suitable for printing. Formats fell out of favor a while ago, largely because they require package variables, but they are meant for what I want to do—create multiple pages of columnar data. Formats handle pagination, which I would have to do myself if I used `printf()` or something similar.

On line 11 of Listing 2, I read in my personal configuration settings found in the file `.almanacrc` in my home directory. The configuration file is very simple. When I move around, I can change the configuration and generate a new almanac:

```
longitude 38.5
latitude 45.5
time_zone 2
```

On line 19, I start a `for()` loop that goes from the current time until one month later (there are  $3.15e7$  seconds in one year, so that many divided by 12 in one month. Remember that number in case you run out of things to talk about at a party—there are approximately  $\pi$  times  $10^7$  seconds in a year). I advance one day for each iteration.

On lines 21 to 25, I get the current date from `localtime()` and format it with `Date::Format` to look nice for me. You may want to change the format to suit yourself. I put the formatted date in the global variable `$date` because formats work with global variables.

In the `foreach()` loop that starts on line 29, I get the sunrise and sunset times for the various altitudes of the sun using the values from the `Astro::Sunrise` documentation. I add each pair of values, the sunrise and sunset, to the global array `@sunrises`. I get the moon illumination from `Astro::MoonPhase's phase()` and store that in the global `$moon_phase`.

Once I have all the variables in place, I `write()` the format. Perl prints the `STDOUT_TOP` if I am at the top of the page (and when I get to the next page, it will do it again, so my columns always have headings). The `STDOUT` format puts the values of the `$date`, `@sunrises`, and `$moon_phase` data in the right places.

On line 35, I get the moon phase data by passing the date-time in `$i`, adjusted for the time zone, to `Astro::MoonPhase's phase()`.

Date	Upper	Civil	Nautical	Astro	Illum
Sun Oct 26	5:52 17:59	5:31 18:20	5:07 18:44	4:43 19:08	2
Mon Oct 27	5:52 17:58	5:31 18:19	5:07 18:43	4:43 19:07	7
Tue Oct 28	5:51 17:58	5:31 18:19	5:07 18:43	4:43 19:07	15
Wed Oct 29	5:51 17:58	5:30 18:19	5:06 18:43	4:42 19:07	25
Thu Oct 30	5:51 17:58	5:30 18:18	5:06 18:42	4:42 19:06	35
Fri Oct 31	5:51 17:57	5:30 18:18	5:06 18:42	4:42 19:06	46
Sat Nov 01	5:51 17:57	5:30 18:18	5:06 18:42	4:42 19:06	57

Example 1: The output of the almanac script.

The number I get back is a fractional value, and I turn it into a percentage by multiplying by 100. To make sure I end up with an integer, I use `sprintf()` to format the number as a decimal integer. I do not mind losing precision after the third decimal place—atmospheric effects make that meaningless anyway.

Now, I have all of the information I want to put into my almanac, and all of the values are in package variables. I call `write()`, which invokes the current format, which is the same name as the

*The civil twilight is when it  
becomes too dark to read outside,  
and that is when we like to start  
moving around at night*

current default output filehandle name. Since I have not messed with any of that, both names are still `STDOUT`. I do not need to pass any arguments to `write()` because the format already knows which package variables to use.

The output (Example 1) looks kind of dry, but it is perfect for my needs. I can refer quickly to any of the columns to get the information I want. When is the morning going to be light enough that it wakes me up? I look in the first column of the “Upper” set of times. When will I be able to see most of the stars? I look at the second column under the “Astro” heading. In the army, we consider the day over (or the night beginning) at nautical twilight, which is the second column of the “Nautical” column.

Pages and pages of this sort of data may be too long for me to carry around, so I can make a graph, like I did before. All I have to do is put the right values in the `@data` array I used in my `GD::Graph` program.

With a couple of Perl modules from CPAN, I was able to quickly churn out as much astrological data as I wanted. If I want to calculate data that does not already have a module, I only need to turn to the instructions in *Practical Astronomy With Your Calculator*, by Peter Duffett-Smith (Cambridge University Press, 1989; ISBN 0521356997), into Perl programs. Some of the `Astro::` modules recommend other sources as well. Good luck with your own astrological projects.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

## Listing 1

```
1 #!/usr/bin/perl
2
3 use Astro::MoonPhase;
4 use Date::Format;
5 use GD::Graph::lines;
6 use Time::Local;
7
8 my $tz_offset = $ARGV[1] || 0;
9 my $year      = $ARGV[0] || ( (localtime)[5] + 1900 );
10
11 # get the first Sunday in the year
```

```
12 my $now = do {
13     for( $day = 1; ; $day++ )
14     {
15         $time = timegm( 0, 0, 0, $day, 0, $year );
16         last unless (localtime($time + $tz_offset))[6];
17     }
18     $time;
19 };
20
21
22 my $then = timegm(59, 59, 23, 31, 11, $year);
23
24 # calculate the moon phase for every three hours
25 for( my $i = $now; $i < $then; $i += 60 * 60 * 3 )
26 {
```



```

27 my @data = phase( $i + $tz_offset );
28
29 push @time, $i;
30 push @illum, $data[1];
31 }
32
33 my @data = ( \@time, \@illum);
34
35 my $my_graph = new GD::Graph::lines( 800, 300 );
36
37 $my_graph->set(
38     x_label      => 'Day',
39     y_label      => 'Moon Illumination',
40     title        => 'Moon Illumination',
41     x_number_format => sub { time2str( "%h %e", $_[0] ) },
42     line_width   => 1,
43     x_label_position => 1/2,
44     r_margin     => 15,
45     x_min_value  => $time[0],
46     x_max_value  => $time[-1],
47     x_tick_number => 52,
48     transparent => 0,
49     x_labels_vertical => 1,
50     tick_clr     => 'gray',
51     border_clr  => 'dgray',
52     x_long_ticks => 1,
53     y_long_ticks => 0,
54     border      => 1,
55     border_clr  => 'black',
56 );
57
58 my $gd = $my_graph->plot(\@data);
59
60 open IMG, "> moon.png" or die "$!\n";
61 print IMG $gd->png;
62 close IMG;

```

## Listing 2

```

1  #!/usr/bin/perl
2  use strict;
3
4  use Astro::MoonPhase;
5  use Astro::Sunrise;
6  use ConfigReader::Simple;
7  use Date::Format;
8
9  use vars qw( $date @sunrises $moon_phase );
10
11 my $config = ConfigReader::Simple->new( ".almanacrc" );
12
13 my $long = $config->longitude;
14 my $lat  = $config->latitude;
15 my $tz   = $config->time_zone;
16
17 my $now = time;
18
19 for( my $i = $now; $i < $now + 3.15e7/12 ; $i += 24 * 60 * 60 )
20 {
21     my @date = localtime( $i );
22
23     print "\n" unless $date[6];
24
25     $date = time2str( "%a %b %d", $i );
26
27     @sunrises = ();
28
29     foreach my $altitude ( -0.833, -6, -12, -18 )
30     {
31         push @sunrises, sunrise( @date[5,4,3],
32                                 $long, $lat, $tz, $date[8], $altitude );
33     }
34
35     $moon_phase = sprintf "%3d", 100 * ( phase( $i + $tz ) )[1];
36
37     write;
38 }
39
40 format STDOUT =
41 @<<<<<<<< @>>>> @>>>> @>>>> @>>>> @>>>> @>>>> @>>>> @>>>>
42 $date,      @sunrises,          $moon_phase
43 .
44
45 format STDOUT_TOP =
46 Date          Upper          Civil          Nautical          Astro          Illum
47 .

```

TPJ

# 101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobb's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.

Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

**\$9.95** For subscribers to  
*The Perl Journal*

**\$12.95** For nonsubscribers to  
*The Perl Journal*

**\$24.95** To subscribe to  
*The Perl Journal* and  
receive *101 Perl Articles*

Go to

**<http://www.tpj.com/>**  
now!



# Scraping Yahoo Groups

*Simon Cozens*

**Y**ou can learn a lot about a man from what he writes about. In my previous articles, ostensibly about Perl both here and at perl.com, I've talked about poker, Ruby, church work, Japanese literature, housemates, and linguistics. This month, I'm going to talk about obscure Japanese pop groups. No, really.

The great Pizzicato Five began life in 1984 as a four-piece band made up of Konishi Yasuharu, Takanami Keitaro, Ryo Kamamiya, and Sasaki Mamiko, and went through a series of line-up changes and hit albums. Eventually becoming a duo of Konishi Yasuharu and Nomiya Maki, they split up in March 2001. I got interested in Pizzicato Five (P5) in March 2001. Life is hard sometimes.

The P5 have always been more popular outside Japan than inside it, although "popular" is something to be taken relatively—there's a vibrant English-language discussion group, P5ML, at Yahoo Groups. Unfortunately, coming late into the P5 scene, I missed a lot of the initial mail. I'd love to have a copy of all the old messages in my mailbox, but I don't really want to have to go trolling through the Yahoo Groups web interface to get them.

Maybe Perl can help.

## Grabbing Mails

The first port of call in cases like this is search.cpan.org, and asking it about "Yahoo Groups" points us directly to the *WWW::Yahoo::Groups* module. This is a module based on *WWW::Mechanize*, a tool we've looked at before for scraping web sites.

Once we've read, marked, and inwardly digested the documentation, we can start working on our program to download the mail:

```
use WWW::Yahoo::Groups;

my $scraper = WWW::Yahoo::Groups->new();
$scraper->login($username => $password);
$scraper->list("p5ml");
```

Now we have logged in, all being well, and selected our group. We're ready to see what mail is available to be downloaded.

---

*Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumping. Simon can be reached at [simon-cozens.org](mailto:simon-cozens.org).*

*WWW::Yahoo::Groups* gives us functions to tell us the first message in the archive and the last one.

```
my $lwm = $scraper->first_msg_id();
my $hwm = $scraper->last_msg_id();
```

These IDs form a sequence internal to Yahoo Groups, and are not actually related to the message ID specified in the e-mail. We could take a simple-minded approach to grabbing all the messages, and simply walk from the low water mark to the high water mark, downloading and storing each message in turn:

```
open OUT, ">> mail/p5ml" or die $!;
for ($lwm...$hwm) {
    print OUT $scraper->fetch_message($_)."\n";
}
```

However, there are two problems with this idea. The first is that it's dangerous—any mail arriving due to new posts to the P5ML mailing list will be clobbered by our overzealous output. The second is that it is wasteful—we may already have downloaded many of the messages in the archive. Let's take these problems one at a time.

## Storing Mail

The first thing we want is a safe way of storing mail into a mailbox. We'll assume that we're going to be using the UNIX mbox format for the purposes of this article since, well, that's what I use. There are, as I've mentioned in the past, a large number of mail folders handling libraries on CPAN, but we're going to use the simplest one, *Email::LocalDelivery*.

Its job is to take an e-mail from the "wire," and store it in some kind of file on disk. It's a very simple module with one method: *deliver*. It takes the e-mail as a plain string, which is quite handy because that's what *WWW::Yahoo::Groups* gives us. Hence, we can make our code a lot safer by rewriting it like so:

```
for ($lwm...$hwm) {
    Email::LocalDelivery->deliver($scraper->fetch_message($_),
                                "~/mail/p5ml");
}
```

This makes sure we don't overwrite incoming messages, but it doesn't necessarily help us to pull down messages we already

have. Of course, for that, we need to know what messages we've already got.

## Reading Folders

Again, there is a plethora of modules for dealing with mail folders, but the best one, which does nothing other than split a folder into separate mail messages, is Richard Clamp's *Email::Folder*:

```
use Email::Folder;
my $folder = Email::Folder->new("mail/p5ml");
```

Once we've opened the folder, we can get its messages as individual *Email::Simple* objects:

```
my @ids = map { $_->header("Message-ID") } $folder->messages;
```

---

*Walking a tree-like data  
structure, such as a message  
thread, should suggest to you a  
recursive algorithm*

---

Now we know what message IDs we have, but we're going to find it much more convenient to have that list as a hash rather than an array. Did that immediately jump out at you? Don't worry if it didn't; just like learning the idioms and constructions of the human language, the hash formulation is just one of those things you sense as you become more fluent in Perl.

We want to know if an incoming message is one that we already hold a copy of. Another way to phrase this is that the message exists in the set of messages in our folder. Once you start thinking about sets and existence in Perl, you should immediately start thinking about hashes because, like sets, they provide a data structure that doesn't necessarily hold an order, but can be used to test easily whether or not something is a member of the set. Perl's *exists* function gives us a quick and simple test for set membership.

It's a Perl rule of thumb—if you want to know if something is part of a list of values, that list should become a hash and you should use *exists*.

So, I'd naturally have written the above line as:

```
my %ids = map { $_->header("Message-ID") => 1 } $folder->messages;
```

Notice that we don't really care about the value attached to the message ID in the hash is because all we're going to use it for is to test for existence.

Now we can look again at the messages we're downloading via *WWW::Yahoo::Groups*. If we see a message that's already in the folder, we can assume that we've caught up with the archives.

The only slight nit is that to match the message ID from the downloaded message against our set from the folder, we need to turn that into an *Email::Simple* object, too. But thankfully, by definition, that's simple!

```
for ($lwm...$tmm) {
    my $message = $scraper->fetch_message($_);
```

```
my $id = Email::Simple->new($message)->header("Message-ID");
last if exists $ids{$id}; # Caught up with archive.
```

```
Email::LocalDelivery->deliver($message, "~/mail/p5ml");
}
```

Hoorah! Now we should have a mail box full of old Pizzicato Five messages. What could be a better use for Perl?

## Threading Mail

I suppose, though, that we might want to have a fast way of catching up with these hundreds of old messages. In next month's article, we're going to look in more depth about summarizing mailing-list archives, but for this month, we'll take a quick look at how to organize these messages into threads. We'll again start with *Email::Folder* and *Email::Simple* to split off the messages from the folder, and now we want to thread them.

Thankfully, CPAN is there for us again, and *Email::Thread* does all the work in arranging a collection of *Email::Simple* objects into threads. It could hardly be much easier. To kick off the threading process, we create a new threader object with all the messages we want organized, and then call the obviously named thread method:

```
use Email::Folder;
use Email::Thread;

my $threader = Email::Thread->new(
    Email::Folder->new("mail/p5ml")->messages
);
$threader->thread;
```

This goes away and populates a bunch of internal data structures—quite remarkably quickly—and allows us to ask for the set of root messages that form the initial messages in threads:

```
my @roots = $threader->root_set;
```

Each of these root messages returned from *root\_set* is an *Email::Thread::Container* object; it may have a parent, child, or sibling, each of which is another *Email::Thread::Container*, and a message, which is the *Email::Simple* object representing the message. The child is a container that refers to a reply to the current node; the sibling is another reply to the same parent.

Sometimes the message will be empty (if we have missed out some posts in the thread), but the threader can still deduce that a message should be there. Let's begin by looking at the subjects of each thread:

```
for my $root (@roots) {
    print $root->message->header("Subject")."\n";
}
```

This gives us a good overview of the various topics that come up on the mailing list, but how popular is each one? To count the number of replies to a thread-starter, we'll have to traverse the child and sibling links of the thread.

Walking a tree-like data structure, such as a message thread, should suggest to you a recursive algorithm: Do something for the node, then perform this algorithm on its sibling, then perform it on its child. I was once told two golden rules for dealing with recursive procedures. First, make sure there's a termination condition, then have faith that it'll do the right thing. It usually will. Here's a basic thread walker:

```
sub walk_thread {
    my $node = shift;
    do_something($node);
```

```

    walk_thread($node->sibling) if $node->sibling;
    walk_thread($node->child)   if $node->child;
}

```

Will this terminate? Eventually, we'll see a node that has no siblings or no replies. Will it do the right thing? It's easier to see this if you assume a thread with no siblings, just one reply to each message:

```

sub walk_thread {
    my $node = shift;
    do_something($node);
    walk_thread($node->child) if $node->child;
}

```

Now we can see that this will cause each child to be visited in turn, and then terminate. Putting the sibling link back is exactly the same but makes the walk two-dimensional.

But how do we go from here to finding out the number of messages in a thread? Imagine an army squad in a field. It's almost in a line, but some people are ahead of others. The commander at the west end of the field wants to know how many soldiers are still alive and responding. He can only see the sergeant, so he stealthily runs over to him and says, "Find out how many soldiers are east of you, add one for yourself, and tell me the answer." The soldier can see two corporals to his east, so asks both of them "Find out how many soldiers are east of you, add one for yourself, and tell me the answer." The corporals go and do exactly the same thing for the men that they can see, and report back the answer. The sergeant gets two answers back, adds them together, adds one for himself and reports it to the commander, who adds one for himself, and knows how many messages, uh, men he's got.

This works because the men at the east end can see no other men, so they add one for themselves, and report back "one," and the answer bubbles up until you've summed everyone. Here's what that looks like in Perl.

```

sub count_offspring {
    my $node = shift;
    my $count = 0;
    if ($node->next) { $count += count_offspring($node->next); }
    if ($node->child) { $count += count_offspring($node->child); }
    # And one for mynode
    $count++;
    return $count;
}

```

And now, we can sort the threads by the number of messages they contain:

```

my @rootset = $thead->rootset;
my %counts = map { $_ => count_offspring($_) } @rootset;
for my $node (sort { $counts{$b} <> $counts{$a} } @rootset) {
    print $node->message->header("Subject"), ":", $counts{$node} . "\n";
}

```

When we do this, we find that, unsurprisingly, the most prominent thread is the one about the band splitting up...

## Providing Links

Now let's change the subject a tiny bit. Getting back to these Yahoo Groups, you'll remember I said there was no correlation between a Yahoo Group message ID and a message's Message-ID header. Wouldn't it be nice, then, if we could insert a header into each message giving the URL where the message came from?

We can do this because we know that *WWW::Yahoo::Groups* uses the *WWW::Mechanize* robot to do its fetching, that *WWW::Mechanize* allows us to get at the *HTTP::Response* object

for the last fetch, and that *HTTP::Response* has a base method. And thankfully, we can get at the underlying *WWW::Mechanize* object by calling *agent*. So our modified scraper code looks like this:

```

for ($lwm...$hwm) {
    my $message = $scraper->fetch_message($_);
    my $uri = $scraper->agent->res->base;

```

Next, we need to modify the headers of this e-mail. Since we're parsing the mail into *Email::Simple* anyway to get the Message ID, we can add an X-Yahoo-URL header in there as well:

```

    my $simple = Email::Simple->new($message);
    my $id = $simple->header("Message-ID");
    $simple->header_set("X-Yahoo-URL", $uri);

    last if exists $ids{$id}; # Caught up with archive.

    Email::LocalDelivery->deliver($simple->as_string, "~/mail/p5ml");
}

```

There's only one slight problem with this—the URLs you get back are really horrifically long in some cases. Thankfully, there's a nice solution we can slot in here—another CPAN module called *WWW::Shorten*. This uses the various URL-shortening services out there, like the well-known MakeAShorterLink—but many others, too—to produce a more palatable URL. Let's support the home side, and use *Ask Bjorn Hansen's metamark*. (*Ask* runs most of the perl.org services.)

```

use WWW::Shorten 'metamark';

for ($lwm...$hwm) {
    my $message = $scraper->fetch_message($_);
    my $uri = makeashorterlink($scraper->agent->res->base);
    my $simple = Email::Simple->new($message);
    my $id = $simple->header("Message-ID");
    $simple->header_set("X-Yahoo-URL", $uri);

    last if exists $ids{$id}; # Caught up with archive.

    Email::LocalDelivery->deliver($simple->as_string, "~/mail/p5ml");
}

```

And there we are! Job done!

## Dedication

Some of you may have cottoned on to something strange about this article: As it progressed from a relatively plausible (for me, at least) premise, it got more and more contrived. My apologies for that, but there is a good reason for it.

This month, I wanted to particularly showcase a special set of modules and a special module author. Three of the modules we discussed in this article, *WWW::Yahoo::Groups*, *Email::Thread*, and *WWW::Shorten* were all written by the same man, Iain Truskett.

Iain not only wrote these modules, but many others, too. He also contributed fixes and advice to a large number of other people's modules, to the Perl 5 core documentation, and to the Date-Time project. Additionally, he ran the Perl books site at <http://books.perl.org/>, and was a regular contributor to many Perl mailing lists, newsgroups, and IRC channels, always known for his patience and calmness at all times.

He was a tireless benefactor to the Perl community, even though he largely remained humbly behind the scenes. We lost a good man when Iain passed away on New Year's Eve, 2003. This article is respectfully dedicated to him.

TPJ





# Watching a Logfile In an IRC Channel

*Randal Schwartz*

The other day, I was hanging out on IRC when Keven Lenzo (creator of the YAPC conference and current director of The Perl Foundation) popped in and asked if anyone had written a tool to dribble a slowly growing file into an IRC channel. I jumped right in and said, “That can be done rather simply with POE,” then realized that I hadn’t actually done anything quite like that yet.

In this column last year (May 2003), I had written a file watcher that provided a web interface, but not a file watcher that would write to IRC. Since I hadn’t yet played much with the POE IRC component, I grabbed the ball and ran with it.

As an interesting part of the problem, I had to figure out how to deal with IRC’s antiflood provisions. Most IRC servers these days will boot off any IRC client that sends too much text at once, and the POE IRC component throttles messages by default, dribbling them out at a rate that appears to be about three messages every 10 seconds. But a logfile that is being watched might grow faster than that. It wouldn’t make sense for my bot to just spew all the lines blindly, getting more and more behind as time went on, so I had to have my code figure out when I was being throttled. Luckily, since the code is all open source, I was able to peer inside and figure out how to adapt. The result is in Listing 1.

Lines 1 through 3 are my standard preamble, turning on warnings, enabling “big program” compiler restrictions, and disabling buffering on standard output.

Lines 5 through 22 delimit the “user serviceable parts” from the rest of the file. Line 7 is the IRC nickname to be used by this bot. Lines 8 through 12 define to which server the bot connects, and the bot’s real name in a manner consistent with the POE IRC component’s initialization parameters. Line 13 defines the channel that the bot will join, sending public messages to that channel. Line 14 defines an internal alias for the virtual IRC client created by the POE IRC Component.

Lines 16 to 20 define the files that the bot will be watching. Each key should be a short identifying string that will show up in the bot’s output and the value of the corresponding file to be tailed. In this case, I’m watching my web server’s access and error log files.

Lines 24 and 25 are used in the throttling process, described later. They have to be defined outside the handler that uses them so that they will persist between handler invocations.

Line 27 pulls in POE, as well as the POE components for IRC and for following a file.

---

*Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems, and Effective Perl Programming, as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.*

Line 29 creates the virtual IRC client, giving the session the name defined in `$IRC_ALIAS`.

Lines 31 through 119 define a second session. If the first session is like the IRC client, the second session being created here is like the virtual human running the client. As such, we’ll have steps here to cause the IRC client to connect to the server, then react as the IRC client notices various conditions. This session doesn’t have a name. For the most part, the session’s behavior is defined by a series of inline states, beginning in line 33.

Lines 33 to 36 define the actions taken by this session as it is starting up. The session pokes at the virtual IRC client to have it respond to all IRC events, then it tries to connect the IRC client to its server.

If the connection is successful, eventually an `irc_255` event is posted back to the virtual-human session (at the end of all the connecting messages), triggering the handler beginning in line 37. This handler tells the IRC client to join a particular channel (line 38), and then start up the heartbeat and listeners (lines 39 and 40). The details of the startup are given later.

Lines 42 through 49 cause null handlers to be installed for many common but uninteresting IRC events. This list started out empty, but after running my bot for a while, I figured out more things that I should ignore (because they were making noise in my default handler) and added them here.

Similarly, line 50 keeps `_child` events from hitting the default handler because I have nothing to do for that either.

Lines 51 to 57 define a `_default` event handler, mostly so that I can diagnose all the events being sent to my virtual human, including dumping the data in a nice way.

Up to this point, we have a fairly generic but stupid IRC bot that knows how to connect to a server and join a channel. The remaining events give the bot its personality. The `my_add` handler (defined starting in line 58) tells the bot to start watching a particular file. We’ll do this by adding a watcher session using `POE::Wheel::FollowTail`.

First, lines 59 and 60 set up two closure variables: the key into the `%FOLLOWS` hash, and the session ID of the virtual-human session. Lines 61 to 76 create the watcher session, watching the filename from the `%FOLLOWS` hash (in line 67) and handling events at each new noticed line via the `got_line` handler (line 68).

The `got_line` handler posts an event back to the virtual-human session to the `my_tailed` handler, including the timestamp, short identifying word, and the line of text that was seen. As each new line is noticed by the POE wheel, we’ll get another event in sequence, carefully timestamped and labeled appropriately.

Although this strategy makes some logical sense (having a separate session for each wheel), at the cost of some complexity, I could have also done the work entirely with one session. I usually go for the dirt-simple methods first, though, and it worked for me this way.

Now that we have line events flowing in, it's time to handle them using the handler starting in line 79. This handler is normally called with the timestamp (integer), the file, and the line of

---

*If the first session is like the IRC client, the second session being created here is like the virtual human running the client*

---

text, which are extracted in line 80. However, under circumstances described shortly, this handler may also be called with no parameters, leaving these three variables *undef*.

Because I wanted this virtual human to be aware of the built-in throttling of the virtual IRC client (to avoid the heavy-handed throttling of the real IRC server), I have to reach under the hood in lines 85 and 86. By staring at the source code for *POE::Component::IRC*, I was able to determine that the *arrayref* stored in its heap at *send\_queue* contained one element per deferred IRC message. If the array is empty, then the next message I send will most likely be sent immediately to the IRC server. I cache this *arrayref* in the outer *\$SEND\_QUEUE* variable just to avoid the bizarre sequences of steps to find it each time.

Lines 89 to 111 implement the throttling-aware transmission. I chose to implement a bit of hysteresis in the design. When the queued elements reach three or more, I start discarding and keep discarding until the queue is completely empty. With this strategy, the discarded lines tend to come in clumps, rather than throwing away every other line or every third line. I chose three as the threshold because it appears that it takes about 10 seconds to dump three average-sized messages.

The variable *\$SKIPPING* keeps track of this state. If it's 0, we're just pushing things through as they arrive. Otherwise, it's the number of messages that we've had to discard because we're skipping.

Lines 89 to 93 handle the skipping to nonskipping transition. If we're currently skipping, but the queue is now empty, then it reports how many messages were discarded by sending a message to the channel (lines 90 and 91) and then resets the *\$SKIPPING* variable.

Lines 96 to 106 execute only if we have a line to display, verified by a nonfalse *\$time* value in line 96.

Lines 97 to 98 handle the transition from nonskipping to skipping, if needed. If we're already skipping, or the queue has now exceeded three messages, then we discard the line and note that it has been discarded (line 98).

Otherwise, it's time to say something. The timestamp is expanded to the time components (seconds, minutes, hours, and so on) in line 100. Line 101 sends a public message to the channel, labeled with the timestamp in HH:MM:SS 12-hour format and the file from which the line comes. The fancy bit of math in line 103 converts 0–23 into 12, 1–12, 1–11, as needed.

Lines 109 to 111 cause this handler to be reinvoked automatically every half second if we're currently skipping. The purpose of this action is to display the discarded *N* messages message as quickly as possible, instead of waiting for more input to come from the file. Note that we reinvoke the handler but pass no parameters, which will make *\$time* false and will cause the middle portion of this handler to be skipped. Using *\$\_[STATE]* here is a shortcut meaning, "The same handler as the one we're in."

Finally, lines 114 to 117 define the heartbeat task. Every 10 seconds, a line that looks like "HH:MM:SS: heartbeat: beep" is shown in the channel. I did this for testing before I had the file-tailing added, but I like how it keeps channel occupants aware that there's stuff coming in from the bot.

Once the POE event handlers and sessions are created, the final line of this program (line 121) tells POE to start the event loop, and away we go. This program never exits because there are always sessions that are alive, but a quick INT character (usually Control-C) takes care of the appropriate termination. The bot rudely disconnects from the IRC server: If I had been more concerned, I could have set up a *\_signal* handler to send a proper quit message to the IRC server with an explanation.

Hopefully, you've seen how easy it is to construct an IRC bot. I must caution you, though, that many people (myself included) despise noisy or spooky bots, so have some common sense when you set these up. Until next time, enjoy!

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

## Listing 1

```
=1=    #!/usr/bin/perl -w
=2=    use strict;
=3=    $|++;
=4=
=5=    ## CONFIG
=6=
=7=    my $NICK = 'weblogger';
=8=    my $CONNECT =
=9=        {Server => 'irc.some.borg',
=10=         Nick => $NICK,
=11=         Ircname => 'weblogger: see merlyn@stonehenge.com',
=12=        };
=13=    my $CHANNEL = '#weblogger';
=14=    my $IRC_ALIAS = "irk";
=15=
=16=    my %FOLLOWS =
=17=        (
=18=            ACCESS => "/var/log/access_log",
```

```
=19=        ERROR => "/var/log/error_log",
=20=        );
=21=
=22=    ## END CONFIG
=23=
=24=    my $SKIPPING = 0;          # if skipping, how many we've done
=25=    my $SEND_QUEUE;           # cache
=26=
=27=    use POE qw(Component::IRC Wheel::FollowTail);
=28=
=29=    POE::Component::IRC->new($IRC_ALIAS);
=30=
=31=    POE::Session->create
=32=        (inline_states =>
=33=            {_start => sub {
=34=                $_[KERNEL]->post($IRC_ALIAS => register => 'all');
=35=                $_[KERNEL]->post($IRC_ALIAS => connect => $CONNECT);
=36=            },
=37=            irc_255 => sub {          # server is done blabbing
=38=                $_[KERNEL]->post($IRC_ALIAS => join => $CHANNEL);
=39=                $_[KERNEL]->yield("heartbeat"); # start heartbeat
=40=                $_[KERNEL]->yield("my_add", $_[0]) for keys %FOLLOWS;
```

```

=41=     },
=42=     (map
=43=     {
=44=         ;"irc_$_" => sub { })
=45=     qw(join public
=46=         connected snotice ctcp_action ping notice mode part quit
=47=         001 002 003 004 005
=48=         250 251 252 253 254 265 266
=49=         332 333 353 366 372 375 376)),
=50=     _child => sub {},
=51=     _default => sub {
=52=         printf "%s: session %s caught an unhandled %s event.\n",
=53=             scalar localtime(), $_[SESSION]->ID, $_[ARG0];
=54=         print "The $_[ARG0] event was given these parameters: ",
=55=             join(" ", map({"ARRAY" eq ref $_[_] ? "[$_]" : "$_"
@{$_[ARG1]})), "\n";
=56=         0; # false for signals
=57=     },
=58=     my_add => sub {
=59=         my $trailing = $_[ARG0];
=60=         my $session = $_[SESSION];
=61=         POE::Session->create
=62=             (inline_states =>
=63=                 (_start => sub {
=64=                     $_[HEAP]->{wheel} =
=65=                         POE::Wheel::FollowTail->new
=66=                         (
=67=                             Filename => $FOLLOWS{$trailing},
=68=                             InputEvent => 'got_line',
=69=                         );
=70=                     },
=71=                     got_line => sub {
=72=                         $_[KERNEL]->post($session => my_tailed =>
=73=                             time, $trailing, $_[ARG0]);
=74=                     },
=75=                     ),
=76=                 );
=77=     },
=78=     },
=79=     my_tailed => sub {
=80=         my ($time, $file, $line) = @_[ARG0..ARG2];
=81=         ## $time will be undef on a probe, or a time value if a real
line
=82=
=83=         ## PoCo::IRC has throttling built in, but no external
visibility
=84=         ## so this is reaching "under the hood"
=85=         $$SEND_QUEUE ||=
=86=             $_[KERNEL]->alias_resolve($IRC_ALIAS)->get_heap-
>{send_queue};
=87=
=88=         ## handle "no need to keep skipping" transition
=89=         if ($$SKIPPING and @$SEND_QUEUE < 1) {
=90=             $_[KERNEL]->post($IRC_ALIAS => privmsg => $CHANNEL =>
=91=                 "[discarded $$SKIPPING messages]");
=92=             $$SKIPPING = 0;
=93=         }
=94=
=95=         ## handle potential message display
=96=         if ($time) {
=97=             if ($$SKIPPING or @$SEND_QUEUE > 3) { # 3 msgs per 10#
seconds
=98=                 $$SKIPPING++;
=99=             } else {
=100=                 my @time = localtime $time;
=101=                 $_[KERNEL]->post($IRC_ALIAS => privmsg => $CHANNEL =>
=102=                     sprintf "%02d:%02d:%02d: %s: %s",
=103=                         ($time[2] + 11) % 12 + 1, $time[1],
$time[0],
=104=                         $file, $line);
=105=             }
=106=         }
=107=
=108=         ## handle re-probe/flush if skipping
=109=         if ($$SKIPPING) {
=110=             $_[KERNEL]->delay($_[STATE] => 0.5); # $time will be undef
=111=         }
=112=
=113=     },
=114=     my_heartbeat => sub {
=115=         $_[KERNEL]->yield(my_tailed => time, "heartbeat", "beep");
=116=         $_[KERNEL]->delay($_[STATE] => 10);
=117=     },
=118=     },
=119=     );
=120=
=121=     POE::Kernel->run;

```

TPJ

# Sign Up For BYTE.com!

**DON'T GET LEFT BEHIND!**  
**Register for BYTE.com today**  
**and have access to...**

- Jerry Pournelle's "Chaos Manor"
- Moshe Bar's "Serving with Linux"
- Martin Heller's "Mr. Computer Language Person"
- David Em's "Media Lab"
- and much more!...

**BYTE.com will keep you up-to-date on emerging trends and technologies with even more rich technical articles and opinions than ever before! Expert opinions, in-depth analysis, trusted information...find all this and more on BYTE.com!**



**ONLY \$19.95  
FOR ANNUAL  
ACCESS!**



**As a special thank you for signing up, receive the BYTE CD-ROM and a year of access for the low price of \$29.95!**

**Registering is easy...go to [www.byte.com](http://www.byte.com) and sign up today! Don't delay!**



**BYTE**



# Perl Cookbook

*Russell J.T. Dyer*

As a Perl programmer, you probably own or have access to O'Reilly's *Perl Cookbook*. It's an essential book for the advanced development of Perl skills. Thanks to the stability of Perl, this is one computer book that doesn't become obsolete very quickly. It was first published in the Summer of 1998 and is still useful today. If your copy of the *Cookbook*, however, is anything like some that I've seen belonging to senior Perlists—filled with post-it notes, notes written in the margins, highlighting and dog-eared pages throughout, and looking more than a little worn—you're probably due for a replacement. Fortunately, you now have an additional excuse to buy a fresh copy of the *Perl Cookbook*, since the second edition is out.

The *Perl Cookbook* is filled with dozens of common and somewhat uncommon dilemmas that one might encounter with Perl in daily life. Scenarios are laid out in clear language, then resolved with excellent explanations. Sometimes the solutions (or recipes) are straightforward and limited. Often times, though, the authors give more than one solution depending on what they imagine the reader may be seeking or may need. The result is a deeper understanding for the reader by way of more examples, and a greater likelihood that the nuances of your particular problem are addressed.

## What's Different

The first edition of the *Perl Cookbook* is based on Perl 5.004.04. Because of the stability and the reverse compatibility of Perl, just about all of the first edition still applies. The authors have updated the text in the second edition for Perl 5.8.1. Many of the changes to old recipes are based on the newer version of Perl. But many of the changes were made to give greater clarity through expanded discussions, and to give the reader more examples since there's always more than one way to solve a problem in Perl. The 200 additional pages in the new edition are composed of changes to

---

*Russell is a Perl programmer, MySQL developer, and web designer living and working on a consulting basis in New Orleans. He is also an adjunct instructor at a local college where he teaches Linux and other open-source software. He can be reached at [russell@dyerhouse.com](mailto:russell@dyerhouse.com).*

*Perl Cookbook, Second Edition*  
Tom Christiansen  
and Nathan Torkington  
O'Reilly & Associates, 2003  
1000 pp., \$49.95  
ISBN 0-596-00313-7

more than 100 recipes, as well as the inclusion of 80 new recipes. With that many changes, I can't list them all here. However, I will highlight several of them briefly.

Math fans will be pleased to find that a new recipe on named Unicode characters (1.5) has been added, as well as a recipe on normalizing similar Unicode characters (1.9) and treating them as octets (1.10). There's a new recipe to format text as title case (1.14). The recipes for trimming blank spaces from the end of strings (1.19), as well as the one for parsing comma-separated data (1.20) were expanded. Chapter 2 on numbers has some recipes that have been reworded and reworked, as well—including 2.2 on rounding floating-point numbers and 2.3 on comparing them. The recipes on converting binary, octal, and hexadecimal numbers have been rewritten and combined into one lengthy recipe (2.15).

There are new recipes in Chapter 8 on dealing with a file's contents: treating a file as an array (8.18); setting the default I/O layers (8.19); converting Microsoft text files into Unicode (8.21); comparing the contents of files (8.22); treating strings as files (8.23); and dealing with flat file indexes (8.27). In Chapter 9 on directories, a recipe has been included on how to handle symbolic file permissions instead of their octal values (9.11). In Chapter 10 on subroutines, a recipe has been introduced for creating a *switch* statement using the *Switch* module with the *case* command—a very



handy way to consolidate multiple *if* and *elsif* statements into a clean format.

Chapter 11 on references has a new recipe for dealing with memory problems common in self-referential data structures (11.15). There's also a new recipe on using program outlines (11.16). Chapter 12 on packages, libraries, and modules has a new recipe that provides a solution for making a function private (12.5). There's another one on customizing warnings in your own Perl module (12.15). And Chapter 13 on objects has a new recipe using the *dclone()* function to give the user a *copy* method for a class.

In Chapter 14 on database accessing, there are several new recipes: escaping embedded quotes (14.10); handling database errors (14.11); setting up database queries within a *loop* statement (14.12); determining the number of rows returned by a database query (14.14); and displaying data retrieved one page at a time (14.16). Chapter 15 is on user interfaces, or rather, interactivity. There are several new recipes on this topic as well: graphing data (15.18); creating thumbnails of images (15.19); and adding text to an image (15.20).

Chapter 17 on sockets has an additional recipe on handling multiple clients from within a process using an operating system's threads (17.14). There's another on managing multiple inputs from

unpredictable sources (17.19). Chapter 20 on web automation has many new recipes—one using cookies (20.14), another two on retrieving password-protected pages using LWP (20.15) and *https* pages (20.16), and two particularly good ones on parsing HTML (20.18) and on extracting data from an HTML table (20.19).

Finally, two new chapters have been added: Chapter 21 on mod-perl contains 17 recipes from authenticating to dealing with cookies to redirection. It has recipes on Apache logs, migrating from CGI to mod-perl, and working with the *HTML::Mason* perl module. Chapter 22 on XML, another new chapter, includes a quick introduction to XML, as well as a few lengthy recipes on parsing XML and validating XML gracefully. It also provides advice on searching an XML tree.

### Conclusion

In summary, the authors and editors of the *Perl Cookbook* have managed to retain what is good and of value in the first edition. They've managed to fine tune the existing recipes for the latest version of Perl and have added many more recipes to keep up with the developing needs of Perl programmers. This was quite an undertaking on their part, and they've succeeded nicely.

TPJ

---

*The authors give more than one solution depending on what they imagine the reader may be seeking or may need*

---

**Subscribe now to**

## **Dr. Dobb's E-mail Newsletters**

**They're Free!** <http://www.ddj.com/maillists/>

- ✓ **AI Expert Newsletter.** Edited by Dennis Merritt; the AI Expert Newsletter is all about artificial intelligence in practice.
- ✓ **Dr. Dobb's Linux Digest.** Edited by Steven Gibson, a monthly compendium that highlights the most important Linux newsgroup discussions.
- ✓ **Dr. Dobb's Software Tools Newsletter.** Having a hard time keeping up with new developer tools and version updates? If so, Dr. Dobb's Software Tools e-mail newsletter is just the deal for you.
- ✓ **Dr. Dobb's Data Compression Newsletter.** Mark Nelson reports on the most recent compression techniques, algorithms, products, tools, and utilities.
- ✓ **Dr. Dobb's Math Power Newsletter.** Join Homer B. Tilton and expand your base of math knowledge.
- ✓ **Dr. Dobb's Active Scripting Newsletter.** Find out the most clever Active Scripting techniques from Mark Baker.

**Sign up now at** <http://www.ddj.com/maillists/>

---

# Source Code Appendix

---

## Julius C. Duque “Encryption Using *Crypt::CBC*”

### Listing 1

```
1 #!/usr/local/bin/perl
2  use diagnostics;
3  use strict;
4  use warnings;
5  use Crypt::CBC;
6
7  my $key = "hello, there!";
8  my $IV = pack "H16", "0102030405060708";
9
10 my $cipher = Crypt::CBC->new({'key' => $key,
11                               'cipher' => 'Khazad',
12                               'iv' => $IV,
13                               'regenerate_key' => 1,
14                               'padding' => 'standard',
15                               'prepend_iv' => 0
16                               });
17
18 my $plaintext1 = pack "H32", "0123456789abcdeffedcba9876543210";
19 print "plaintext1 : ", unpack("H*", $plaintext1), "\n";
20
21 my $ciphertext1 = $cipher->encrypt($plaintext1);
22 print "ciphertext1 : ", unpack("H*", $ciphertext1), "\n";
23
24 my $plaintext2 = $cipher->decrypt($ciphertext1);
25 print "plaintext2 : ", unpack("H*", $plaintext2), "\n";
```

### Listing 2

```
1 #!/usr/local/bin/perl
2  use diagnostics;
3  use strict;
4  use warnings;
5  use Getopt::Long;
6  use Crypt::CBC;
7
8  my $SRC_RANDOM = "/dev/urandom";
9  my $IV_FILE = "iv.rand";
10 my $BLOCKSIZE = 16;
11 my $BLOCK_CIPHER = "Serpent";
12 my $IV;
13 my ($encrypt, $decrypt) = ();
14 GetOptions("encrypt" => \$encrypt, "decrypt" => \$decrypt);
15
16 sub get_input
17 {
18     my ($message) = @_;
19     local $| = 1;
20     local *TTY;
21     open TTY, "/dev/tty";
22     my ($tkey1, $tkey2);
23     system "stty -echo </dev/tty";
24     do {
25         print STDERR "Enter $message: ";
26         chomp($tkey1 = <TTY>);
27         print STDERR "\nRe-type $message: ";
28         chomp($tkey2 = <TTY>);
29         print STDERR "\n";
30         print STDERR "\nThe two $message", "s don't match. ",
31             "Please try again.\n\n" unless $tkey1 eq $tkey2;
32     } until $tkey1 eq $tkey2;
33
34     system "stty echo </dev/tty";
35     close TTY;
36     return $tkey1;
37 }
38
39 my $key = &get_input("password");
40
41 chomp $ARGV[0];
42 open INFILE, $ARGV[0];
43
44 my $cipher;
45
46 if ($encrypt) {
47     open RANDSRC, $SRC_RANDOM;
48     read(RANDSRC, $IV, $BLOCKSIZE);
49     close RANDSRC;
50     open SRC_IV, "> $IV_FILE";
51     print SRC_IV $IV;
52     close SRC_IV;
53 }
```

```

54     $cipher = Crypt::CBC->new({ 'key' => $key,
55                                 'cipher' => $BLOCK_CIPHER,
56                                 'iv' => $IV,
57                                 'regenerate_key' => 1,
58                                 'padding' => 'standard',
59                                 'prepend_iv' => 0
60                                 });
61
62     $cipher->start('encrypt');
63 }
64
65 if ($decrypt) {
66     open RANDSRC, $IV_FILE;
67     read(RANDSRC, $IV, $BLOCKSIZE);
68     close RANDSRC;
69
70     $cipher = Crypt::CBC->new({ 'key' => $key,
71                                 'cipher' => $BLOCK_CIPHER,
72                                 'iv' => $IV,
73                                 'regenerate_key' => 1,
74                                 'padding' => 'standard'
75                                 });
76
77     $cipher->start('decrypt');
78 }
79
80 while (read(INFILE, my $buffer, 1048576)) {
81     print $cipher->crypt($buffer);
82 }
83
84 close INFILE;
85 print $cipher->finish;

```

## brian d foy “An Almanac in Perl”

### Listing 1

```

1  #!/usr/bin/perl
2
3  use Astro::MoonPhase;
4  use Date::Format;
5  use GD::Graph::lines;
6  use Time::Local;
7
8  my $tz_offset = $ARGV[1] || 0;
9  my $year      = $ARGV[0] || ( ( localtime )[5] + 1900 );
10
11 # get the first Sunday in the year
12 my $now = do {
13     for( $day = 1; ; $day++ )
14     {
15         $time = timegm( 0, 0, 0, $day, 0, $year );
16         last unless ( localtime($time + $tz_offset) )[6];
17     }
18
19     $time;
20 };
21
22 my $then = timegm(59, 59, 23, 31, 11, $year);
23
24 # calculate the moon phase for every three hours
25 for( my $i = $now; $i < $then; $i += 60 * 60 * 3 )
26 {
27     my @data = phase( $i + $tz_offset );
28
29     push @time, $i;
30     push @illum, $data[1];
31 }
32
33 my @data = ( \@time, \@illum,);
34
35 my $my_graph = new GD::Graph::lines( 800, 300 );
36
37 $my_graph->set(
38     x_label      => 'Day',
39     y_label      => 'Moon Illumination',
40     title        => 'Moon Illumination',
41     x_number_format => sub { time2str( "%h %e", $_[0] ) },
42     line_width   => 1,
43     x_label_position => 1/2,
44     r_margin     => 15,
45     x_min_value  => $time[0],
46     x_max_value  => $time[-1],
47     x_tick_number => 52,
48     transparent => 0,
49     x_labels_vertical => 1,
50     tick_clr     => 'gray',
51     border_clr   => 'dgray',
52     x_long_ticks => 1,

```

```

53     y_long_ticks      => 0,
54     border            => 1,
55     border_clr        => 'black',
56 );
57
58 my $gd = $my_graph->plot(\@data);
59
60 open IMG, "> moon.png" or die "$!\n";
61 print IMG $gd->png;
62 close IMG;

```

## Listing 2

```

1  #!/usr/bin/perl
2  use strict;
3
4  use Astro::MoonPhase;
5  use Astro::Sunrise;
6  use ConfigReader::Simple;
7  use Date::Format;
8
9  use vars qw( $date @sunrises $moon_phase );
10
11 my $config = ConfigReader::Simple->new( ".almanacrc" );
12
13 my $long = $config->longitude;
14 my $lat  = $config->latitude;
15 my $tz   = $config->time_zone;
16
17 my $now = time;
18
19 for( my $i = $now; $i < $now + 3.15e7/12 ; $i += 24 * 60 * 60 )
20 {
21     my @date = localtime( $i );
22
23     print "\n" unless $date[6];
24
25     $date = time2str( "%a %b %d", $i );
26
27     @sunrises = ();
28
29     foreach my $altitude ( -0.833, -6, -12, -18 )
30     {
31         push @sunrises, sunrise( @date[5,4,3],
32                                 $long, $lat, $tz, $date[8], $altitude );
33     }
34
35     $moon_phase = sprintf "%3d", 100 * ( phase( $i + $tz ) )[1];
36
37     write;
38 }
39
40 format STDOUT =
41 @<<<<<<<< @>>>> @>>>> @>>>> @>>>> @>>>> @>>>> @>>>> @>>>>
42 $date,      @sunrises,          $moon_phase
43 .
44
45 format STDOUT_TOP =
46 Date          Upper          Civil          Nautical      Astro      Illum
47 .

```

## Randal Schwartz “Watching a Logfile in an IRC Channel”

### Listing 1

```

=1= #!/usr/bin/perl -w
=2= use strict;
=3= $|++;
=4=
=5= ## CONFIG
=6=
=7= my $NICK = 'weblogger';
=8= my $CONNECT =
=9=     {Server => 'irc.some.borg',
=10=      Nick => $NICK,
=11=      Ircname => 'weblogger: see merlyn@stonehenge.com',
=12=      };
=13= my $CHANNEL = '#weblogger';
=14= my $IRC_ALIAS = "irk";
=15=
=16= my %FOLLOWS =
=17= (
=18=     ACCESS => "/var/log/access_log",
=19=     ERROR => "/var/log/error_log",
=20= );
=21=
=22= ## END CONFIG
=23=
=24= my $SKIPPING = 0;          # if skipping, how many we've done
=25= my $SEND_QUEUE;          # cache

```



```

=26=
=27= use POE qw(Component::IRC Wheel::FollowTail);
=28=
=29= POE::Component::IRC->new($IRC_ALIAS);
=30=
=31= POE::Session->create
=32= (inline_states =>
=33=   {_start => sub {
=34=     $_[KERNEL]->post($IRC_ALIAS => register => 'all');
=35=     $_[KERNEL]->post($IRC_ALIAS => connect => $CONNECT);
=36=   },
=37=   irc_255 => sub { # server is done blabbing
=38=     $_[KERNEL]->post($IRC_ALIAS => join => $CHANNEL);
=39=     $_[KERNEL]->yield("heartbeat"); # start heartbeat
=40=     $_[KERNEL]->yield("my_add", $_[0]) for keys %FOLLOWS;
=41=   },
=42=   {map
=43=     {
=44=       ;"irc_$_" => sub { }}
=45=     qw(join public
=46=       connected snotice ctcp_action ping notice mode part quit
=47=       001 002 003 004 005
=48=       250 251 252 253 254 265 266
=49=       332 333 353 366 372 375 376)),
=50=     _child => sub {},
=51=     _default => sub {
=52=       printf "%s: session %s caught an unhandled %s event.\n",
=53=         scalar localtime(), $_[SESSION]->ID, $_[ARG0];
=54=       print "The $_[ARG0] event was given these parameters: ",
=55=         join(" ", map({"ARRAY" eq ref $_[0] ? "[$_]" : "$_" } @$_[ARG1]}), "\n";
=56=       0; # false for signals
=57=     },
=58=     my_add => sub {
=59=       my $trailing = $_[ARG0];
=60=       my $session = $_[SESSION];
=61=       POE::Session->create
=62=         (inline_states =>
=63=           {_start => sub {
=64=             $_[HEAP]->{wheel} =
=65=               POE::Wheel::FollowTail->new
=66=                 (
=67=                   Filename => $FOLLOWS{$trailing},
=68=                   InputEvent => 'got_line',
=69=                 );
=70=             },
=71=             got_line => sub {
=72=               $_[KERNEL]->post($session => my_tailed =>
=73=                 time, $trailing, $_[ARG0]);
=74=             },
=75=           ),
=76=         );
=77=     },
=78=   },
=79=   my_tailed => sub {
=80=     my ($time, $file, $line) = @_[ARG0..ARG2];
=81=     ## $time will be undef on a probe, or a time value if a real line
=82=
=83=     ## PoCo::IRC has throttling built in, but no external visibility
=84=     ## so this is reaching "under the hood"
=85=     $SEND_QUEUE ||=
=86=       $_[KERNEL]->alias_resolve($IRC_ALIAS)->get_heap->{send_queue};
=87=
=88=     ## handle "no need to keep skipping" transition
=89=     if ($$SKIPPING and @$SEND_QUEUE < 1) {
=90=       $_[KERNEL]->post($IRC_ALIAS => privmsg => $CHANNEL =>
=91=         "[discarded $$SKIPPING messages]");
=92=       $$SKIPPING = 0;
=93=     }
=94=
=95=     ## handle potential message display
=96=     if ($time) {
=97=       if ($$SKIPPING or @$SEND_QUEUE > 3) { # 3 msgs per 10# seconds
=98=         $$SKIPPING++;
=99=       } else {
=100=        my @time = localtime $time;
=101=        $_[KERNEL]->post($IRC_ALIAS => privmsg => $CHANNEL =>
=102=          sprintf "%02d:%02d:%02d: %s: %s",
=103=            ($time[2] + 11) % 12 + 1, $time[1], $time[0],
=104=            $file, $line);
=105=        }
=106=      }
=107=
=108=      ## handle re-probe/flush if skipping
=109=      if ($$SKIPPING) {
=110=        $_[KERNEL]->delay($_[STATE] => 0.5); # $time will be undef
=111=      }
=112=    },
=113=    my_heartbeat => sub {
=114=

```

---

```
=115=         $_[KERNEL]->yield(my_tailed => time, "heartbeat", "beep");
=116=         $_[KERNEL]->delay($_[STATE] => 10);
=117=     }
=118=     },
=119=     );
=120=
=121= POE::Kernel->run;
```