

\$SPAD/src/clef edible.c

The Axiom Team

July 28, 2014

Abstract

Contents

1	edible Call Graph	3
---	-------------------	---

```

/*
Copyright (c) 1991-2002, The Numerical Algorithms Group Ltd.
All rights reserved.

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

- Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in
  the documentation and/or other materials provided with the
  distribution.

- Neither the name of The Numerical Algorithms Group Ltd. nor the
  names of its contributors may be used to endorse or promote products
  derived from this software without specific prior written permission.

```

```

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

```

1 edible Call Graph

This was generated by the GNU cflow program with the argument list. Note that the line:NNNN numbers refer to the line in the code after it has been tangled from this file.

```

cflow --emacs -l -n -b -T --omit-arguments edible.c

;; This file is generated by GNU cflow 1.3. -*- cflow -*-
2 { 0} +-main() <int main () line:122>
3 { 1} +-ptyopen()
4 { 1} +-perror()
5 { 1} +-exit()
6 { 1} +-catch_signals() <void catch_signals () line:482>
7 { 2} | +-sprintf()
8 { 2} | +-getpid()

```

```

 9 { 2} | +-open()
10 { 2} | +-write()
11 { 2} | +-strlen()
12 { 2} | +-close()
13 { 2} | +-bsdSignal()
14 { 2} | +-hangup_handler() <void hangup_handler () line:374>
15 { 3} |   +-open()
16 { 3} |   +-write()
17 { 3} |   +-strlen()
18 { 3} |   +-close()
19 { 3} |   +-kill()
20 { 3} |   +-tcsetattr()
21 { 3} |   +-perror()
22 { 3} |   +-printf()
23 { 3} |   +-unlink()
24 { 3} |   \-exit()
25 { 2} | +-child_handler() <void child_handler () line:430>
26 { 3} |   +-open()
27 { 3} |   +-write()
28 { 3} |   +-strlen()
29 { 3} |   +-close()
30 { 3} |   +-Cursor_shape()
31 { 3} |   +-kill()
32 { 3} |   +-tcsetattr()
33 { 3} |   +-perror()
34 { 3} |   +-printf()
35 { 3} |   +-unlink()
36 { 3} |   \-exit()
37 { 2} | +-terminate_handler() <void terminate_handler () line:396>
38 { 3} |   +-open()
39 { 3} |   +-write()
40 { 3} |   +-strlen()
41 { 3} |   +-close()
42 { 3} |   +-sleep()
43 { 3} |   +-kill()
44 { 3} |   +-tcsetattr()
45 { 3} |   +-perror()
46 { 3} |   +-printf()
47 { 3} |   +-Cursor_shape()
48 { 3} |   +-fprintf()
49 { 3} |   +-unlink()
50 { 3} |   \-exit()
51 { 2} | +-interrupt_handler() <void interrupt_handler () line:418>
52 { 3} |   +-open()
53 { 3} |   +-write()
54 { 3} |   +-strlen()
55 { 3} |   +-close()
56 { 3} |   +-sleep()
57 { 3} |   \-kill()
58 { 2} | +-alarm_handler() <void alarm_handler () line:452>

```

```

59 { 3} | | +-getppid()
60 { 3} | | +-open()
61 { 3} | | +-write()
62 { 3} | | +-strlen()
63 { 3} | | +-close()
64 { 3} | | +-alarm()
65 { 3} | | +-tcsetattr()
66 { 3} | | +-perror()
67 { 3} | | +-Cursor_shape()
68 { 3} | | +-fprintf()
69 { 3} | | +-unlink()
70 { 3} | | \-exit()
71 { 2} | \-alarm()
72 { 1} +-strcmp()
73 { 1} +-load_wct_file()
74 { 1} +-fprintf()
75 { 1} +-skim_wct()
76 { 1} +-sprintf()
77 { 1} +-getpid()
78 { 1} +-open()
79 { 1} +-tcgetattr()
80 { 1} +-fork()
81 { 1} +-setsid()
82 { 1} +-dup2()
83 { 1} +-close()
84 { 1} +-tcsetattr()
85 { 1} +-execvp()
86 { 1} +-getenv()
87 { 1} +-strdup()
88 { 1} +-execlp()
89 { 1} +-getppid()
90 { 1} +-init_flag()
91 { 1} +-define_function_keys()
92 { 1} +-init_reader()
93 { 1} +-init_parent() <void init_parent () line:328>
94 { 2} | +-tcgetattr()
95 { 2} | +-perror()
96 { 2} | +-exit()
97 { 2} | +-tcsetattr()
98 { 2} | \-Cursor_shape()
99 { 1} +-FD_ZERO()
100 { 1} +-FD_SET()
101 { 1} +-set_function_chars() <void set_function_chars () line:575>
102 { 1} +-write()
103 { 1} +-strlen()
104 { 1} +-select()
105 { 1} +-check_flip() <void check_flip () line:502>
106 { 2} | +-tcgetattr()
107 { 2} | +-perror()
108 { 2} | +-flip_canonical() <void flip_canonical () line:547>

```

```

109 { 3} | +-tcsetattr()
110 { 3} | +-perror()
111 { 3} | +-exit()
112 { 3} | \-Cursor_shape()
113 { 2} | \-flip_raw() <void flip_raw () line:533>
114 { 3} | +-send_buff_to_child()
115 { 3} | +-tcsetattr()
116 { 3} | +-perror()
117 { 3} | \-exit()
118 { 1} +-FD_ISSET()
119 { 1} +-read()
120 { 1} +-back_up()
121 { 1} +-print_whole_buff()
122 { 1} \-do_reading()

```

— * —

```

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <termios.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <errno.h>
#include <signal.h>

#if defined (SGIplatform)
#include <bstring.h>
#endif

#include "edible.h"
#include "com.h"
#include "bsdsignal.h"

#include "bsdsignal.h1"
#include "openpty.h1"
#include "prt.h1"
#include "edin.h1"
#include "wct.h1"
#include "edible.h1"
#include "fnct-key.h1"

#ifdef AXIOM_UNLIKELY
#define log 1

```

```

#define logterm 1
#define siglog 1
#endif

#define Cursor_shape(x)

#ifdef siglog
int sigfile;
char sigbuff[256];
#endif

/* Here are the term structures I need for setting and resetting the
terminal characteristics. */

struct termios childbuf; /* the childs normal operating termio */
struct termios oldbuf; /* the initial settings */
struct termios rawbuf; /* the parents raw state, when it is doing nothing */
struct termios canonbuf; /* set it to be canonical */

/* the terminals mapping of the function keys */
unsigned char _INTR, _QUIT, _ERASE, _KILL, _EOF, _EOL, _RES1, _RES2;

int ppid; /* the parents's parent pid */
int child_pid; /* the childs process id */

short INS_MODE ; /* Flag for insert mode */
short ECHOIT = 1; /* Flag for echoing */
short PTY; /* Flag which tells me whether I should echo newlines */

int MODE; /* am I in cbreak, raw, or canonical */

char in_buff[1024]; /* buffer for storing characters read until they are processed */
char buff[MAXLINE]; /* Buffers for collecting input and */
int buff_flag[MAXLINE]; /* flags for whether buff chars
                        are printing
                        or non-printing */

char controllerPath[20]; /* path name for opening the controller side */
char serverPath[20]; /* path name for opening the server side */

int contNum, serverNum; /* file descriptors for pty's */
int num_read; /* Number of chars read */

#ifdef log
int logfd;
char logpath[30];

```

```

#endif

int
main(int argc, char *argv[])
{
    fd_set rfd;          /* the structure for the select call */
    int code;            /* return code from system calls */
    char out_buff[MAXLINE]; /* from child and stdin */
    int out_flag[MAXLINE] ; /* initialize the output flags */
    char *program;        /* a string to hold the child program invocation */
    char **pargs = 0;      /* holds parts of the command line */
    int not_command = 1;    /* a flag while parsing the command line */

    /* try to get a pseudoterminal to play with */
    if (ptyopen(&contNum, &serverNum, controllerPath, serverPath) == -1) {
        perror("ptyopen failed");
        exit(-1);
    }

    /* call the routine that handles signals */
    catch_signals();

    /* parse the command line arguments - as with the aixterm the command
       argument -e should be last on the line. */

    while(++argv && not_command) {
        if(!strcmp(*argv, "-f"))
            load_wct_file(++argv);
        else if(!strcmp(*argv, "-e")) {
            not_command = 0;
            pargs = ++argv;
        }
        else {
            fprintf(stderr, "usage: clef [-f fname] -e command\n");
            exit(-1);
        }
    }
    skim_wct();

#ifdef log
    sprintf(logpath, "/tmp/cleflog%d", getpid());
    logfd = open(logpath, O_CREAT | O_RDWR, 0666);
#endif

    /* get the original termio settings, so the child has them */

    if(tcgetattr(0,&childbuf) == -1) {
        perror("clef trying to get the initial terminal settings");
    }

```



```

    exit(-1);
}

/* start the child process */

child_pid = fork();
switch(child_pid) {
case -1 :
    perror("clef can't create a new process");
    exit(-1);
case 0:
    /* CHILD */
    /* Dissasociate form my parents group so all my child processes
       look at my terminal as the controlling terminal for the group */
    setsid();

    serverNum = open(serverPath,O_RDWR);
    if (serverNum == -1) perror("open serverPath failed");

    /* since I am the child, I can close ptc, and dup pts for all it
       standard descriptors */
    if (dup2(serverNum, 0) == -1) perror("dup2 0 failed");
    if (dup2(serverNum, 1) == -1) perror("dup2 1 failed");
    if (dup2(serverNum, 2) == -1) perror("dup2 2 failed");
    if( (dup2(serverNum, 0) == -1) ||
        (dup2(serverNum, 1) == -1) ||
        (dup2(serverNum, 2) == -1) ) {
        perror("clef trying to dup2");
        exit(-1);
    }

    /* since they have been duped, close them */
    close(serverNum);
    close(contNum);

    /* To make sure everything is nice, set off enhedit */
    /*   childbuf.c_line = 0; */

    /* reconfigure the child's terminal get echoing */
    if(tcsetattr(0, TCSAFLUSH, &childbuf) == -1) {
        perror("clef child trying to set child's terminal");
        exit(-1);
    }

    /* fire up the child's process */
    if(pargs){
        execvp( pargs[0], pargs);
        perror("clef trying to execvp its argument");
        fprintf(stderr, "Process --> %s\n", pargs[0]);
    }
}

```

```

    }
    else{
        program = getenv("SHELL");
        if (!program)
program = strdup("/bin/sh");
        else
program = strdup (program);
        execlp( program,program, 0);
        perror("clef trying to execlp the default child");
        fprintf(stderr, "Process --> %s\n", program);
    }
    exit(-1);
    break;
    /* end switch */
}
/* PARENT */
/* Since I am the parent, I should start to initialize some stuff.
   I have to close the pts side for it to work properly */

close(serverNum);
ppid = getppid();

/* I initialize some stuff for the reading and writing */
init_flag(out_flag, MAXLINE);
define_function_keys();
init_reader();
PTY = 1;
init_parent();

/* Here is the main loop, it simply starts reading characters from
   the std input, and from the child. */

while(1) {          /* loop forever */

    /* use select to see who has stuff waiting for me to handle */
    /* set file descriptors for ptc and stdin */
    FD_ZERO(&rfd);
    FD_SET(contNum,&rfd);
    FD_SET(0,&rfd);
    set_function_chars();
#ifdef log
    {
        char modepath[30];
        sprintf(modepath, "\nMODE = %d\n", MODE);
        write(logfd, modepath, strlen(modepath));
    }
#endif
#ifdef logterm
    {
        struct termio ptermio;

```

```

        char pbuff[1024];
        tcgetattr(contNum, &ptermio);
        sprintf(pbuff, "child's settings: Lflag = %d, Oflag = %d, Iflag = %d\n",
            ptermio.c_lflag, ptermio.c_oflag, ptermio.c_iflag);
        write(logfd, pbuff, strlen(pbuff));
    }
#endif

    code = select(FD_SETSIZE, (void *) &rfd, NULL, NULL, NULL);
    for(; code < 0 ;) {
        if(errno == EINTR) {
            check_flip();
            code = select(FD_SETSIZE, (void *) &rfd, NULL, NULL, NULL);
        }
        else {
            perror("clef select failure");
            exit(-1);
        }
    }

    /* reading from the child */
    if( FD_ISSET(contNum, &rfd) ) {
        if( (num_read = read(contNum, out_buff, MAXLINE)) == -1 ) {
            num_read = 0;
        }
#ifdef log
        write(logfd, "OUT<<<<<", strlen("OUT<<<<<"));
        write(logfd, out_buff, num_read);
#endif
        if(num_read > 0) {
            /* now do the printing to the screen */
            if(MODE != CLEFRAW) {
                back_up(buff_ptr);
                write(1, out_buff, num_read);
                print_whole_buff(); /* reprint the input buffer */
            }
            else write(1, out_buff, num_read);
        }
    } /* done the child stuff */
    /* I should read from std input */
    else {
        if(FD_ISSET(0, &rfd) ) {
            num_read = read(0, in_buff, MAXLINE);
#ifdef log
            write(logfd, "IN<<<<<", strlen("IN<<<<<"));
            write(logfd, in_buff, num_read);
#endif
        }
        check_flip();
        if(MODE == CLEFRAW )
            write(contNum, in_buff, num_read);
    }
}

```

```

        else
            do_reading();
    }
}
}
}

void
init_parent(void)
{

    /* get the original termio settings, so I never have to check again */
    if(tcgetattr(0, &oldbuf) == -1) {
        perror("clef trying to get terminal initial settings");
        exit(-1);
    }

    /* get the settings for my different modes */
    if ((tcgetattr(0, &canonbuf) == -1) ||
        (tcgetattr(0, &rawbuf) == -1) ) {
        perror("clef trying to get terminal settings");
        exit(-1);
    }

    canonbuf.c_lflag &= ~(ICANON | ECHO | ISIG);
    /* read before an eoln is typed */

    canonbuf.c_lflag |= ISIG;

    /* canonbuf.c_line = 0;          turn off enhanced edit */

    canonbuf.c_cc[VMIN] = 1;          /* we want every character */
    canonbuf.c_cc[VTIME] = 1;         /* these may require tweaking */

    /* also set up the parents raw setting for when needed */
    rawbuf.c_oflag = rawbuf.c_iflag = rawbuf.c_lflag /* = rawbuf.c_line */ = 0;
    rawbuf.c_cc[VMIN] = 1;
    rawbuf.c_cc[VTIME] = 1;

    if(tcsetattr(0, TCSAFLUSH, &canonbuf) == -1) {
        perror("clef setting parent terminal to canonical processing");
        exit(0);
    }

    /* initialize some flags I will be using */
    MODE = CLEFCANONICAL;
    INS_MODE = 1;

```

```

    Cursor_shape(2);
}

void
hangup_handler(int sig)
{
#ifdef siglog
    sigfile = open(sigbuff, O_RDWR | O_APPEND);
    write(sigfile, "Hangup Handler\n", strlen("Hangup Handler\n"));
    close(sigfile);
#endif
    /* try to kill my child if it is around */
    if(kill(child_pid, 0)) kill(child_pid, SIGTERM);
    if(kill(ppid, 0) >= 0) {
        /* fix the terminal and exit */
        if(tcsetattr(0, TCSAFLUSH, &oldbuf) == -1) {
            perror("clef restoring terminal in hangup handler");
        }
        printf("\n");
    }
    /* remove the temporary editor filename */
    unlink(editorfilename);
    exit(-1);
}

void
terminate_handler(int sig)
{
#ifdef siglog
    sigfile = open(sigbuff, O_RDWR | O_APPEND);
    write(sigfile, "Terminate Handler\n", strlen("Terminate Handler\n") + 1);
    close(sigfile);
    sleep(1);
#endif
    kill(child_pid, SIGTERM);
    /* fix the terminal, and exit */
    if(tcsetattr(0, TCSAFLUSH, &oldbuf) == -1) {
        perror("clef restoring terminal in terminate handler");
    }
    printf("\n");
    Cursor_shape(2);
    fprintf(stderr, "\n");
    /* remove the temporary editor filename */
    unlink(editorfilename);
    exit(0);
}

void
interrupt_handler(int sig)

```

```

{
#ifdef siglog
    sigfile = open(sigbuff, O_RDWR | O_APPEND);
    write(sigfile, "Interrupt Handler\n", strlen("Interrupt Handler\n") + 1);
    close(sigfile);
    sleep(1);
#endif
    kill(child_pid, SIGINT);
}

void
child_handler(int sig)
{
#ifdef siglog
    sigfile = open(sigbuff, O_RDWR | O_APPEND );
    write(sigfile, "Child Handler\n", strlen("Child Handler\n") + 1);
    close(sigfile);
#endif
    Cursor_shape(2);
    close(contNum);
    if(kill(ppid, 0) >= 0) {
        /* fix the terminal, and exit */
        if(tcsetattr(0, TCSAFLUSH, &oldbuf) == -1) {
            perror("clef restoring terminal in child handler");
        }
        printf("\n");
    }
    /* remove the temporary editor filename */
    unlink(editorfilename);
    exit(0);
}

void
alarm_handler(int sig)
{
    int newppid = getppid();
#ifdef siglog
    sigfile = open(sigbuff, O_RDWR | O_APPEND);
    write(sigfile, "Alarm Handler\n", strlen("Alarm Handler\n")+ 1 );
    close(sigfile);
#endif
    /* simply gets the parent process id, if different, it terminates ,
       otherwise it resets the alarm */

    if(ppid == newppid) {
        alarm(60);
        return;
    }
    else {
        /* once that is done fix the terminal, and exit */

```

```

        if(tcsetattr(0, TCSAFLUSH, &oldbuf) == -1) {
            perror("clef restoring terminal in alarm handler");
        }
        Cursor_shape(2);
        fprintf(stderr, "\n");
        /* remove the temporary editor filename */
        unlink(editorfilename);
        exit(0);
    }
}

/* a procedure which tells my parent how to catch signals from its children */
void
catch_signals(void)
{
#ifdef siglog
    sprintf(sigbuff, "/tmp/csig%d", getpid());
    sigfile = open(sigbuff, O_RDWR | O_TRUNC | O_CREAT);
    write(sigfile, "Started \n", strlen("Started \n"));
    close(sigfile);
#endif
    bsdSignal(SIGHUP, hangup_handler, RestartSystemCalls);
    bsdSignal(SIGCHLD, child_handler, RestartSystemCalls);
    bsdSignal(SIGTERM, terminate_handler, RestartSystemCalls);
    bsdSignal(SIGINT, interrupt_handler, RestartSystemCalls);
    bsdSignal(SIGALRM, alarm_handler, RestartSystemCalls);
    alarm(60);
}

/* Here is where I check the child's termio settings, and try to copy them.
   I simply trace through the main modes (CLEFRAW, CLEFCANONICAL) and
   try to simulate them */
void
check_flip(void)
{
    return;

#ifdef 0
    /*simply checks the termio structure of the child */

    if(tcgetattr(contNum, &childbuf) == -1) {
        perror("clef parent getting child's terminal in check_termio");
    }
    if(childbuf.c_lflag & ICANON) {
        if(MODE != CLEFCANONICAL) {
            flip_canonical(contNum);
            MODE = CLEFCANONICAL;
        }
    }
    else {

```

```

        if(MODE != CLEFRAW) {
            flip_raw(contNum);
            MODE = CLEFRAW;
        }
    }
    /* While I am here, lets set the echo */
    if(childbuf.c_lflag & ECHO) ECHOIT = 1;
    else ECHOIT = 0;
#endif
}

void
flip_raw(int chann)
{
    if(MODE == CLEFCANONICAL)
        send_buff_to_child(chann);

    if(tcsetattr(0, TCSAFLUSH, &rawbuf) == -1) {
        perror("clef resetting parent to raw ");
        exit(-1);
    }
}

void
flip_canonical(int chann)
{
    if(tcsetattr(0, TCSAFLUSH, &canonbuf) == -1) {
        perror("clef resetting parent to canonical ");
        exit(-1);
    }
    if(INS_MODE)
        Cursor_shape(5);
    else
        Cursor_shape(2);
}

void
etc_get_next_line(char * line,int * nr,int fd)
{
    *nr = read(fd, line, 1024);
    if(*nr == -1) {
        perror("Reading /etc/master");
    }
    if(*nr == 0) {
        fprintf(stderr, "Not found \n");
    }
}

```



```

}

#define etc_whitespace(c) ((c == ' ' || c == '\t')?(1):(0))

void
set_function_chars(void)
{
    /* get the special characters */
    _INTR = childbuf.c_cc[VINTR];
    _QUIT = childbuf.c_cc[VQUIT];
    _ERASE = childbuf.c_cc[VERASE];
    _KILL = childbuf.c_cc[VKILL];
    _EOF = childbuf.c_cc[VEOF];
    _EOL = childbuf.c_cc[VEOL];
    return;
}

```

References

- [1] nothing