Projet TEX-Talk
Rapport de spécifications

Nicolas Bernard
Sujet proposé par Nicolas Ollinger

20 mai 2002

Le but de ce projet est de réaliser un programme ressemblant au *talk* classique des systèmes Unix mais dans lequel l'utilisateur pourrait taper des commandes TEX (ou plutôt LATEX) qui seraient interprétées et mises en forme à la volée.

1 Vague ébauche de solution

Nous allons procéder en plusieurs étapes:

Les premières versions seront principalement des interfaces entre différents éléments: *Qemacs* pour la saisie du texte, *Xdvi* pour l'affichage du texte mis en forme, *TEX* pour faire cette mise en forme, ainsi que *dvichop* pour traiter la sortie de *TEX* avant de l'envoyer à *Xdvi*. Ce sera la première phase.

Dans la phase 1-A, *Qemacs* ne sera pas utilisé, la saisie étant directement gérée par le programme.

La phase 1-B introduira *Qemacs*.

La partie communication entre personnes ne sera pas implémentée dans les phases 1-A et 1-B.

Elle le sera par contre dans la phase 1-C. Elle pourra ou non réutiliser les sources du Talk classique. Eventuellement, il se peut que cette partie ne fonctionne qu'entre personnes connectées sur une même machine

et non à travers un réseau, cette dernière étape faisant alors l'objet d'une **phase 1-D** séparée.

Dans la phase deux l'idée est de s'affanchir des services de *Qemacs* et *Xdvi*, c'est-à-dire de faire un programme graphique (En utilisant une bibliothèque qui reste à définir, *cf* la partie consacrée aux détails techniques) intégrant la zone de saisie et la zone d'affichage.

La phase 2-A est l'équivalent des phases 1-A et 1-B, c'est-à-dire qu'il n'y a ici pas de communications possibles.

Dans la phase 2-B on ajoute ladite gestion des communications, avec ou sans réutilisation du code de *Talk*

Vient alors la phase trois: Le problème des versions précédentes est qu'un utilisateur peut utiliser une commande interdisant toute saisie ultérieure: dans cette phase, il s'agit de modifier le programme de la version deux de manière à interdire ce genre de manipulations, ou du moins à fournir un moyen de les réparer.

Si jamais il reste du temps (on peut toujours réver!) on peut imaginer de remplacer *dvichop* par un programme dédié faisant exactement ce que l'on veut et intégré au programme principal.

2 "Norme" de codage

Afin de garder le code aussi lisible que possible nous allons essayer de nous en tenir aux règles de développement du noyau de NetBSD telles qu'énoncées dans l'annexe A (connues sous le nom de KNF, pour *Kernel Normal Form*).

3 Quelques détails techniques

3.1 Pour les premières versions:

Les premières versions étant principalement des interfaces entre différents éléments, elles seront probablement réalisées en *shell-script* ou en *C*.

3.2 Pour la phase deux:

Les programmes de la phase deux seront sans doute écrits en *C*. On utilisera pour réaliser l'interface graphique une bibliothèque spécialisée qui reste à préciser (probablement *GTK+*). La gestion et l'affichage du format *dvi* seront réalisés là aussi avec une bibliothèque dédiée.

3.3 La phase trois:

La phase trois est encore très floue. On peut imaginer avoir un programme *lex* qui filtre les entrées, ou une fonction de *reset* de TEX.

3.4 Configuration requise:

Tout cela nécessitera, pour la partie 1, les logiciels mentionnés, et pour la partie deux les bibliothèques et logiciels utilisés. Moyennant l'existence de ces programmes, on peut raisonnablement espérer un fonctionnement sur tout système Unix bénéficiant d'un serveur *X-window*. Notons également que pour toutes les versions, il faudra avoir une version de TEX supportant l'option *-ipc* (cela nécessite généralement une recompilation de TEX). Du point de vue matériel, il faudra une machine capable d'exécuter TEX et d'afficher le fichier *dvi* rapidement, donc un processeur avec une fréquence d'horloge d'au moins 200 MHz.

4 Qui fait quoi?

Etant seul sur ce projet, la distribution des tâches va être simple. Une aide de l'auteur du sujet serait appréciée pour aider à la définition des commandes dangereuses pour la phase trois... si jamais on arrive jusque là...

La progression de la programmation suivra plus où moins les étapes mentionnées ci-dessus; des parties de la phase 1 pourront être sautées, et la phase deux étant sans doute la plus grosse partie, il n'est pas certain que l'on arrive à la phase trois.

A La norme de codage choisie

```
/* $NetBSD: style,v 1.20 2001/10/23 18:51:05 kleink Exp $ */

/*
 * The revision control tag appears first, with a blank line after it.
```

```
 * Copyright text appears after the revision control tag.
 */

/*
 * The NetBSD source code style guide.
 * (Previously known as KNF - Kernel Normal Form).
 *
 *      from: @(#)style        1.12 (Berkeley) 3/18/94
 */
/*
 * An indent(1) profile approximating the style outlined in
 * this document lives in /usr/share/misc/indent.pro.  It is a
 * useful tool to assist in converting code to KNF, but indent(1)
 * output generated using this profile must not be considered to
 * be an authoritative reference.
 */

/*
 * Source code revision control identifiers appear after any copyright
 * text.  Use the appropriate macros from <sys/cdefs.h>.  Usually only one
 * source file per program contains a __COPYRIGHT() section.
 * Historic Berkeley code may also have an __SCCSID() section.
 * Only one instance of each of these macros can occur in each file.
 */
#include <sys/cdefs.h>
#ifndef __lint
__COPYRIGHT("@(#) Copyright (c) 2000\n\
        The NetBSD Foundation, inc. All rights reserved.\n");
__RCSID("$NetBSD: style,v 1.20 2001/10/23 18:51:05 kleink Exp $");
#endif /* !__lint */

/*
 * VERY important single-line comments look like this.
 */

/* Most single-line comments look like this. */

/*
 * Multi-line comments look like this.  Make them real sentences.  Fill
 * them so they look like real paragraphs.
 */

/*
```

```
 * Attempt to wrap lines longer than 80 characters appropriately.
 * Refer to the examples below for more information.
 */

/*
 * EXAMPLE HEADER FILE:
 *
 * A header file should protect itself against multiple inclusion.
 * E.g, <sys/socket.h> would contain something like:
 */
#ifndef _SYS_SOCKET_H_
#define _SYS_SOCKET_H_
/*
 * Contents of #include file go between the #ifndef and the #endif at the end.
 */
#endif /* !_SYS_SOCKET_H_ */
/*
 * END OF EXAMPLE HEADER FILE.
 */

/*
 * Kernel include files come first.
 */
#include <sys/types.h>          /* Non-local includes in brackets. */

/*
 * If it's a network program, put the network include files next.
 * Group the includes files by subdirectory.
 */
#include <net/if.h>
#include <net/if_dl.h>
#include <net/route.h>
#include <netinet/in.h>
#include <protocols/rwhod.h>

/*
 * Then there's a blank line, followed by the /usr include files.
 * The /usr include files should be sorted!
 */
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
```

```
/*
 * Global pathnames are defined in /usr/include/paths.h.  Pathnames local
 * to the program go in pathnames.h in the local directory.
 */
#include <paths.h>

/* Then, there's a blank line, and the user include files. */
#include "pathnames.h"          /* Local includes in double quotes. */

/*
 * ANSI function declarations for private functions (i.e. functions not used
 * elsewhere) and the main() function go at the top of the source module.
 * Don't associate a name with the types.  I.e. use:
 *      void function(int);
 * Use your discretion on indenting between the return type and the name, and
 * how to wrap a prototype too long for a single line.  In the latter case,
 * lining up under the initial left parenthesis may be more readable.
 * In any case, consistency is important!
 */
static char *function(int, int, float, int);
static int dirinfo(const char *, struct stat *, struct dirent *,
                   struct statfs *, int *, char **[]);
static void usage(void);
int main(int, char *[]);

/*
 * Macros are capitalized, parenthesized, and should avoid side-effects.
 * If they are an inline expansion of a function, the function is defined
 * all in lowercase, the macro has the same name all in uppercase.
 * If the macro is an expression, wrap the expression in parenthesis.
 * If the macro is more than a single statement, use ``do { ... } while (0)'',
 * so that a trailing semicolon works.  Right-justify the backslashes; it
 * makes it easier to read. The CONSTCOND comment is to satisfy lint(1).
 */
#define MACRO(v, w, x, y)                                               \
do {                                                                    \
        v = (x) + (y);                                                      \
        w = (y) + 2;                                                    \
} while (/* CONSTCOND */ 0)

#define DOUBLE(x) ((x) * 2)

/* Enum types are capitalized.  No comma on the last element. */
```

```
enum enumtype {
        ONE,
        TWO
} et;

/*
 * When declaring variables in structures, declare them organized by use in
 * a manner to attempt to minimize memory wastage because of compiler alignment
 * issues, then by size, and then by alphabetical order. E.g, don't use
 * ``int a; char *b; int c; char *d''; use ``int a; int b; char *c; char *d''.
 * Each variable gets its own type and line, although an exception can be made
 * when declaring bitfields (to clarify that it's part of the one bitfield).
 * Note that the use of bitfields in general is discouraged.
 *
 * Major structures should be declared at the top of the file in which they
 * are used, or in separate header files, if they are used in multiple
 * source files.  Use of the structures should be by separate declarations
 * and should be "extern" if they are declared in a header file.
 *
 * It may be useful to use a meaningful prefix for each member name.
 * E.g, for ``struct softc'' the prefix could be ``sc_''.
 */
struct foo {
        struct foo *next;        /* List of active foo */
        struct mumble amumble;        /* Comment for mumble */
        int bar;
        unsigned int baz:1,        /* Bitfield; line up entries if desired */
                     fuz:5,
                     zap:2;
        u_int8_t flag;
};
struct foo *foohead;            /* Head of global foo list */

/* Make the structure name match the typedef. */
typedef struct BAR {
        int level;
} BAR;

/*
 * All major routines should have a comment briefly describing what
 * they do.  The comment before the "main" routine should describe
 * what the program does.
 */
```

```
int
main(int argc, char *argv[])
{
        long num;
        int ch;
        char *ep;

        /*
         * At the start of main(), call setprogname() to set the program
         * name.  This does nothing on NetBSD, but increases portability
         * to other systems.
         */
        setprogname(argv[0]);

        /*
         * For consistency, getopt should be used to parse options.  Options
         * should be sorted in the getopt call and the switch statement, unless
         * parts of the switch cascade.  Elements in a switch statement that
         * cascade should have a FALLTHROUGH comment.  Numerical arguments
         * should be checked for accuracy.  Code that cannot be reached should
         * have a NOTREACHED comment.
         */
        while ((ch = getopt(argc, argv, "abn")) != -1) {
                switch (ch) {                /* Indent the switch. */
                case 'a':                /* Don't indent the case. */
                        aflag = 1;
                        /* FALLTHROUGH */
                case 'b':
                        bflag = 1;
                        break;
                case 'n':
                        num = strtol(optarg, &ep, 10);
                        if (num <= 0 || *ep != '\0')
                                errx(1, "illegal number -- %s", optarg);
                        break;
                case '?':
                default:
                        usage();
                        /* NOTREACHED */
                }
        }
        argc -= optind;
        argv += optind;
```

```
        /*
         * Space after keywords (while, for, return, switch).  No braces are
         * used for control statements with zero or only a single statement,
         * unless it's a long statement.
         *
         * Forever loops are done with for's, not while's.
         */
        for (p = buf; *p != '\0'; ++p)
                continue;               /* Explicit no-op */
        for (;;)
                stmt;

        /*
         * Parts of a for loop may be left empty.  Don't put declarations
         * inside blocks unless the routine is unusually complicated.
         */
        for (; cnt < 15; cnt++) {
                stmt1;
                stmt2;
        }

        /* Second level indents are four spaces. */
        while (cnt < 20)
                z = a + really + long + statement + that + needs + two lines +
                    gets + indented + four + spaces + on + the + second +
                    and + subsequent + lines;

        /*
         * Closing and opening braces go on the same line as the else.
         * Don't add braces that aren't necessary except in cases where
         * there are ambiguity or readability issues.
         */
        if (test) {
                /*
                 * I have a long comment here.
                 */
#ifdef zorro
                z = 1;
#else
                b = 3;
#endif
        } else if (bar) {
```

```
                stmt;
                stmt;
        } else
                stmt;

        /* No spaces after function names. */
        if ((result = function(a1, a2, a3, a4)) == NULL)
                exit(1);

        /*
         * Unary operators don't require spaces, binary operators do.
         * Don't excessively use parenthesis, but they should be used if
         * statement is really confusing without them, such as:
         * a = b->c[0] + ~d == (e || f) || g && h ? i : j >> 1;
         */
        a = ((b->c[0] + ~d == (e || f)) || (g && h)) ? i : (j >> 1);
        k = !(l & FLAGS);

        /*
         * Exits should be 0 on success, and 1 on failure.  Don't denote
         * all the possible exit points, using the integers 1 through 300.
         * Avoid obvious comments such as "Exit 0 on success."
         */
        exit(0);
}

/*
 * The function type must be declared on a line by itself
 * preceding the function.
 */
static char *
function(int a1, int a2, float fl, int a4)
{
        /*
         * When declaring variables in functions declare them sorted by size,
         * then in alphabetical order; multiple ones per line are okay.
         * Function prototypes should go in the include file "extern.h".
         * If a line overflows reuse the type keyword.
         *
         * DO NOT initialize variables in the declarations.
         */
        extern u_char one;
        extern char two;
```

```
        struct foo three, *four;
        double five;
        int *six, seven;
        char *eight, *nine, ten, eleven, twelve, thirteen;
        char fourteen, fifteen, sixteen;

        /*
         * Casts and sizeof's are not followed by a space.  NULL is any
         * pointer type, and doesn't need to be cast, so use NULL instead
         * of (struct foo *)0 or (struct foo *)NULL.  Also, test pointers
         * against NULL.  I.e. use:
         *
         *        (p = f()) == NULL
         * not:
         *        !(p = f())
         *
         * Don't use '!' for tests unless it's a boolean.
         * E.g. use "if (*p == '\0')", not "if (!*p)".
         *
         * Routines returning void * should not have their return values cast
         * to any pointer type.
         *
         * Use err/warn(3), don't roll your own!
         */
        if ((four = malloc(sizeof(struct foo))) == NULL)
                err(1, NULL);
        if ((six = (int *)overflow()) == NULL)
                errx(1, "Number overflowed.");
        return (eight);
}

/*
 * Use ANSI function declarations.  ANSI function braces look like
 * old-style (K&R) function braces.
 * As per the wrapped prototypes, use your discretion on how to format
 * the subsequent lines.
 */
static int
dirinfo(const char *p, struct stat *sb, struct dirent *de, struct statfs *sf,
        int *rargc, char **rargv[])
{       /* Insert an empty line if the function has no local variables. */

        /*
```

```
         * In system libraries, catch obviously invalid function arguments
         * using _DIAGASSERT(3).
         */
        _DIAGASSERT(p != NULL);
        _DIAGASSERT(filedesc != -1);

        if (stat(p, sb) < 0)
                err(1, "Unable to stat %s", p);

        /*
         * To printf 64 bit quantities, use %ll and cast to (long long).
         */
        printf("The size of %s is %lld\n", p, (long long)sb->st_size);
}

/*
 * Functions that support variable numbers of arguments should look like this.
 * (With the #include <stdarg.h> appearing at the top of the file with the
 * other include files).
 */
#include <stdarg.h>

void
vaf(const char *fmt, ...)
{
        va_list ap;

        va_start(ap, fmt);
        STUFF;
        va_end(ap);
                                /* No return needed for void functions. */
}

static void
usage(void)
{

        /*
         * Use printf(3), not fputs/puts/putchar/whatever, it's faster and
         * usually cleaner, not to mention avoiding stupid bugs.
         * Use snprintf(3) or strlcpy(3)/strlcat(3) instead of sprintf(3);
         * again to avoid stupid bugs.
         *
```

```
             * Usage statements should look like the manual pages.  Options w/o
             * operands come first, in alphabetical order inside a single set of
             * braces.  Followed by options with operands, in alphabetical order,
             * each in braces.  Followed by required arguments in the order they
             * are specified, followed by optional arguments in the order they
             * are specified.  A bar (`|') separates either/or options/arguments,
             * and multiple options/arguments which are specified together are
             * placed in a single set of braces.
             *
             * Use getprogname() instead of hardcoding the program name.
             *
             * "usage: f [-ade] [-b b_arg] [-m m_arg] req1 req2 [opt1 [opt2]]\n"
             * "usage: f [-a | -b] [-c [-de] [-n number]]\n"
             */
            (void)fprintf(stderr, "usage: %s [-ab]\n", getprogname());
            exit(1);
}
```