Notes on the Xvi Source Code

Chris Downey
John Downey

Xvi (pronounced *ecks-vee-eye*) is a free, portable, multi-window implementation of the popular UNIX® editor **vi**.

This document contains information on how to port **xvi** to systems not currently supported. It also explains how the **xvi** source code is arranged into modules, and explains some of the data structures which are used, so that modifications may be made if and when necessary to the editor itself.

1. INTRODUCTION

Xvi is intended to be portable to just about any system. This is one of the central reasons for its existence; the authors wish to be able to use the same editor everywhere.

The main body of the editor is (supposedly) fully portable, relying only on standard facilities defined by the White Book, and on a set of *primitives* which are provided by a set of one or more modules for each operating system. If **__STDC__** is defined, certain ANSI C facilities will be used, but the editor will compile with non-ANSI compilers.

Therefore, in order to port **xvi** to a new system, all that is necessary is to provide the defined set of *primitives*, and then build the editor. Or at least, that's the idea; we have refined the set of primitives as we port the editor to new environments, and it's getting pretty easy now.

The rest of this document is divided into sections as follows:

Section 2: System-Specific Modules
: This section deals with the layout of source files and makefiles which you will have to deal with when porting **xvi**.

Section 3: Primitives Provided by xvi
: Discusses what primitives are provided by the main body of the editor source code for use by the system interface code.

Section 4: System Interface
: Explains the primitives which need to be provided in order to make **xvi** work.

Section 5: Data Structures
: Details the internal data types used in the editor, and any functions available for operating on those types.

Section 6: Source Files
: Lists the source files comprising the editor, and explains what functionality is provided by each one.

2. SYSTEM-SPECIFIC MODULES

The system-specific code normally consists of three (or more) files; a "**.c**" file, a "**.h**" file, and a makefile. For example:

qnx.c
qnx.h
makefile.qnx

comprise the system-specific module for the QNX operating system.

In most cases, the system-specific code is divided into two or more modules, where one (called the *system interface module*) is concerned with general interactions with the operating system and the other (called the *terminal interface module*) is designed for a specific interface to a display and keyboard (and possibly, a mouse).

For example, the generic UNIX implementation has **unix.c** and **unix.h** for the system interface module, and **termcap.c** and **termcap.h** for the terminal interface module; this should work reasonably with any full-duplex terminal that can be described in the **termcap** database. On consoles with memory-mapped displays, or systems with graphic user interfaces, however, it may be possible to achieve faster display updating, and perhaps other benefits, by replacing the **termcap** module with another one that makes better use of whatever facilities are available. For instance, there is an experimental version for SunView, which allows mouse input on Sun workstations running the SunView window system.

On the other hand, the **termcap**-specific routines might conceivably be useful on some other operating systems (such as VMS), so in general it seemed a good idea to make the **termcap**-specific routines a separate module.

The current MS-DOS implementation has a separate terminal interface module, which is designed specifically for IBM PC compatible computers. This is in the files

ibmpc_a.asm
ibmpc_c.c
ibmpc.h

The first of these is written in assembly language because there are not enough routines common to the various MS-DOS C compilers which reliably access the display and keyboard at a low enough level.

The hardware-independent system interface module for MS-DOS is in

msdos_a.asm
msdos_c.c
msdos.h

The first of these is written in assembly language for the same reason as is **ibmpc_a.asm**.

Theoretically, different terminal interface modules could be written for MS-DOS systems running on hardware which is not IBM-compatible but, unfortunately, such systems seem to be virtually extinct nowadays.

Sometimes more than one makefile is provided, as in the case of UNIX, where different versions work in slightly different ways.

It is, of couse, not necessary to provide all — or any — of these files for a particular implementation; this is just a convention. The makefile(s) for each system determine what files are used in the compilation of the editor.

The following porting modules are available at present:

System	Makefile	Source Files
UNIX		
BSD	**makefile.bsd**	**unix.[ch] termcap.[ch]**
System V †	**makefile.usg**	**unix.[ch] termcap.[ch]**
AIX	**makefile.aix**	**unix.[ch] termcap.[ch]**
ULTRIX	**makefile.ult**	**unix.[ch] termcap.[ch]**
Xenix †	**makefile.xen**	**unix.[ch] termcap.[ch]**
POSIX (e.g. BSDI)	**makefile.pos**	**unix.[ch] termcap.[ch]**
SunOS	**makefile.sun**	**unix.[ch] termcap.[ch]**
SunView	**makefile.sv**	**unix.[ch] sunview.h sunfront.c sunback.c xvi.icn**
MS-DOS		**msdos_c.c msdos.h ibmpc_c.c ibmpc.h**
Microsoft C 5.* & MASM 5.*	**makefile.msc**	**8086mm.inc ibmpc_a.asm msdos_a.asm**
Microsoft Quick C & MASM 5.*	**makefile.qc**	**8086mm.inc ibmpc_a.asm msdos_a.asm**
Zortech C++ 2.* & MASM 5.*	**makefile.zc2**	**8086mm.inc ibmpc_a.asm msdos_a.asm**
Zortech C++ 3.* & MASM 5.*	**makefile.zc3**	**8086mm.inc ibmpc_a.asm msdos_a.asm**
Zortech C++ 3.* 386 protected mode	**makefile.386**	**pc386.[ch]**
OS/2 †		
Version 1, text mode Microsoft C 5.1 & MASM 5.1	**makefile.os2**	**os2vio.[ch] i286.asm**
QNX		
Version 2/3 (CII)	**makefile.qnx**	**qnx.[ch]**
Version 4 (Watcom C)	**makefile.qn4**	**unix.[ch] termcap.[ch]**
TOS †		
Lattice C	**makefile.tos**	**tos.[ch] tos.lnk**

† Versions marked with † probably do not work, as systems have not been recently available to the authors for testing.

3. PRIMITIVES PROVIDED BY XVI

3.1. General Definitions

The file **xvi.h** should be included by all system-specific modules; this file should also be edited so that a system-specific header file (or files), as determined by a predefined keyword, will be included.

For instance, under UNIX, the word **UNIX** is defined by passing the **-DUNIX** flag to the C compiler from the makefile, and **xvi.h** contains the following lines:

```
#ifdef UNIX
#   include "unix.h"
#endif
```

in order to obtain the UNIX-related definitions from that header file.

Among the definitions in **xvi.h** are the following:

bool_t
: A Boolean type having values **TRUE** or **FALSE**.

const
volatile
: These are defined out when **__STDC__** is not defined, so that it is always safe to use them.

xvi.h also includes various other header files which are needed. The following system header files are always included:

stdio.h
ctype.h
signal.h
string.h

These files are included if **__STDC__** is defined:

stddef.h
stdlib.h
limits.h

and if **__STDC__** is not defined, **xvi.h** will provide its own definitions for the following:

```
INT_MAX
INT_MIN
ULONG_MAX

FILE *fopen();
char  *malloc();
char  *getenv();
```

Finally, one of the following header files will be included:

stdarg.h
varargs.h

depending on whether **__STDC__** is defined or not. In order to make coding of **varargs** functions easier, a macro **VA_START**() is defined, which takes the same arguments as the ANSI-style **va_start**(), but which is also available in non-ANSI environments (e.g. BSD).

In order to make it possible to use ANSI-style prototypes for function declarations, but still allow compilation under non-ANSI environments, the following macro is provided:

```
#ifdef __STDC__
#  define P(args) args
#else
#  define P()    ()
#endif
```

so that function declarations may be specified thus:

```
extern FILE *fopen P((const char *, const char *));
```

Please use this facility when you provide declarations for your system primitives, unless your system always uses an ANSI compiler.

3.2. Parameters

An important facility provided for use by system-specific modules is access to the editor's parameter table. This is achieved by means of some apparent functions, and a set of **#define**d token values. The functions are:

void set_param(int n, val)
: This function sets the indicated parameter to the passed value, which must be of an appropriate type. Parameter values may be obtained by means of the following functions (actually macros):

char *Ps(int n)
: return value of string parameter

int Pn(int n)
: return value of numeric parameter

bool_t Pb(int n)
: return value of boolean parameter

char **Pl(int n)
: return value of list parameter (a **NULL**-terminated array of character pointers)

int Pen(int n)
: return numeric value (index) of enumerated parameter

char **Pes(int n)
: return string value of enumerated parameter

In all cases, the **int n** argument is the index of the parameter in the table; a set of **#define**s is provided, of the form:

```
P_name
```

which map the parameter names into integral values. Thus, for example, we might obtain the value of the **colour** parameter:

```
colour = Pn(P_colour);
```

or set the value of the **helpfile** parameter:

```
set_param(P_helpfile, "/usr/lib/xvi/help");
```

4. SYSTEM INTERFACE

4.1. Introduction

There follows a list of the primitives which must be provided either by the system interface module or by the underlying OS. Note that it is perfectly acceptable to implement functions or

external variables as macros so long as they "look the same" as the definitions below. As a guideline, anything which is (a) in capitals, or (b) is a **const** variable, will be implemented as a **#define** for most systems.

When you want to actually do the port, it is highly recommended that you copy the system-specific files for the system which seems closest to your own, and modify those files, rather than starting from scratch.

All the following symbols should be defined in the system interface module, or by standard header files already included by **xvi.h**, or by other header files explicitly included by the system-specific header file:

const unsigned int MAXPATHLEN
: The maximum number of characters in a pathname.

const unsigned int MAXNAMLEN
: The maximum number of characters in a filename.

int remove(char *filename)
: Remove the named file as per ANSI.

int rename(char *old, char *new)
: Rename the file **old** to **new** as per ANSI.

void sleep(unsigned int seconds)
: Put the process to sleep for the given number of seconds.

const char * const DIRSEPS
: The pathname separators supported for system calls (e.g. "\\/" for MS-DOS).

FILE *fopenrb(char *file)
FILE *fopenwb(char *file)
: Like the standard **fopen()** library call, but they both open files in "binary" mode (i.e. no conversion of cr/lf/crlf is done), for reading and writing respectively.

bool_t exists(char *filename)
: Returns **TRUE** if the named file exists.

bool_t can_write(char *filename)
: Returns **TRUE** if the named file can be written, i.e. if a **fopenwb(filename)** will succeed.

char *fexpand(char *filename)
: Returns a filename-expanded version of the passed filename.

#define SETVBUF_AVAIL
const unsigned int READBUFSIZ
const unsigned int WRTBUFSIZ
: If **SETVBUF_AVAIL** (or **__STDC__**) is defined, these constant values are used to set I/O buffer sizes (using the **setvbuf()** function) for reading and writing files. Note that if buffers of these sizes are unavailable at runtime, the editor will try to allocate smaller buffers by iteratively halving the buffer size until the allocation succeeds. It is therefore acceptable for these values to be quite large.

char *tempfname(const char *filename)
: Create a unique name for a temporary file, possibly using **filename** as a base (this will be used by **do_preserve()** to create a backup file for the file named by **filename**). The string returned must have been allocated using **malloc()**; **NULL** can be returned if there is no more memory available.

int call_system(char *command)

Invoke the given command in a subshell. This is used for shell escapes from **xvi**. The command string may contain metacharacters which are expected to be expanded by a command interpreter, e.g. UNIX **/bin/sh**, MS-DOS **command.com**. Return value is 0 for success. In many environments, this call may safely be **#define**d as **system(command)**.

int call_shell(char *shell)

Invoke the named shell. This is used for the **:shell** command. It may be mapped into **call_system()**, but is separate on some systems for efficiency reasons (i.e. not invoking two shells to get one). Return value is 0 for success.

bool_t
sys_pipe(char *cmd, int (*wf)(FILE *), long (*rf)(FILE *))

Used for the **!** command. The first parameter is the command to invoke, while the second and third are functions which should be called with an open file pointer in order to write out old, or read in new lines (respectively). Note that if "real" pipes are not available, it is acceptable to implement this function using temporary files, but the **wf** function must obviously be called before **rf**.

void sys_exit(int code)

Exit with given exit status. This routine must not return. The editor is considered "dead" once it has been called, and no further calls to editor functions should be made.

void delay(void)

Delay for a short time, about a fifth of a second. This is used for showing matching brackets when **showmatch** is set. It is acceptable to just return if implementing this is not easy.

4.2. Screen Control

An instance of the following structure must be defined in order to allow screen output to take place:

```
 typedef struct virtscr {
   genptr           *pv_window;
   int              pv_rows;
   int              pv_cols;
/* public: */
   VirtScr          *(*v_new)(VirtScr *);
   void             (*v_close)(VirtScr *);

   int              (*v_rows)(VirtScr *);
   int              (*v_cols)(VirtScr *);

   void             (*v_clear_all)(VirtScr *);
   void             (*v_clear_line)(VirtScr *);

   void             (*v_goto)(VirtScr *, int row, int col);
   void             (*v_advise)(VirtScr *, int row, int col,
                                            int index, char *str);

   void             (*v_write)(VirtScr *, int row, int col, char *str);
   void             (*v_putc)(VirtScr *, int row, int col, int ch);

   void             (*v_set_colour)(VirtScr *, int colour);
   int              (*v_colour_cost)(VirtScr *);

   void             (*v_flush)(VirtScr *);

   void             (*v_beep)(VirtScr *);

/* optional: not used if NULL */
   void             (*v_insert)(VirtScr *, int row, int col, char *str);

   int              (*v_scroll)(VirtScr *, int start, int end, int nlines);
 } VirtScr;
```

The first three fields in this structure are “private”, for use only within the implementation of the “public” functions. The remaining fields are all function pointers, and are described below. Note that all functions have at least one parameter, which is a pointer to the instance of the **VirtScr** in question. This is always referred to as **vs** below. Note also that the top-left-hand corner of the window is taken to be (0,0).

v_new(vs)
: Obtain a new **VirtScr**, and return a pointer to it. This is not used at present, and should return **NULL**.

v_close(vs)
: Close the window to which **vs** refers.

v_rows(vs)
: Return the number of rows in **vs**.

v_cols(vs)
: Return the number of columns in **vs**.

v_clear_all(vs)
Clear the window completely.

v_clear_line(vs, int row, int col)
Clear the specified line, from the given column to the right hand edge of the window, inclusive.

v_goto(vs, int row, int col)
Move the cursor to the specified row and column.

v_advise(vs, int row, int col, int index, char *str)
This function is called when the editor is about to produce some output on the same line as the last output, but separate from it by one or more characters. The destination position is the coordinate pair **(row, col + index)**, and **str** contains the string of characters which are in the window starting at position **(row, col)**. Where there is a cost incurred by moving the cursor to a specific screen position, the terminal interface module may decide to write the intervening characters to the screen rather than using a specific "move cursor" sequence, in order to minimise the number of characters written to the terminal.

Note that for many environments, the cost of re-positioning the cursor is nil, and under these circumstances this function need not do anything.

v_write(vs, int row, int col, char *str)
Write the specified string of characters into the window, starting at the specified row and column. The parameters will be such that the string will always fit into a single line of the window, i.e. no line-wrapping is necessary; however, it is quite possible for the string to end on the last character of a line, and some implementations will need to take special precautions to handle this correctly.

v_putc(vs, int row, int col, int ch)
This is like **v_write** but for a single character.

v_set_colour(vs, int colour)
Set the colour for all subsequent output (including clearing of lines or the whole window) to the specified colour. The meaning of the value is system-specific.

v_colour_cost(vs)
Return the number of extra characters which are taken up in the window by a colour change. This is almost always 0, but there exist some terminals for which it is not (see the "**sg**" **termcap** capability).

v_flush(vs)
Flush all screen output, and move the cursor on the screen to the correct position. The screen need not actually be updated until either this function is called, or **xvi_handle_event()** returns.

v_beep(vs)
Beep. It is acceptable to flash the screen or window if no audio facility is available.

v_insert(vs, int row, int col, char *str)
This function inserts the given string at the given position, pushing any other characters on the same row to the right. If such a facility is not available, the function pointer should be set to **NULL**.

v_scroll(vs, int start, int end, int nlines)
This function scrolls the set of lines between **start** and **end** (inclusive) by **nlines** lines. If **nlines** is positive, *normal* scrolling should be done, i.e. the lines should be moved upwards with respect to the window. If **nlines** is negative, scrolling should be in the reverse direction. The lines which are left by the scrolling should be cleared. The function should return non-zero if the scrolling was successful, otherwise 0.

If scrolling is not available, the function pointer should be set to **NULL**.

4.3. Parameters

Default values should be **#define**d for certain parameters as follows:

Parameter Name	Type	**#define** name
syscolour	numeric	**DEF_SYSCOLOUR**
colour	numeric	**DEF_COLOUR**
statuscolour	numeric	**DEF_STCOLOUR**
roscolour	numeric	**DEF_ROSCOLOUR**
helpfile	string	**HELPFILE**
format	string	**DEF_TFF**

4.4. File Formats

The functions in **xvi** which read and write text files are aware of several different newline conventions (for example, **"\n"** on UNIX, **"\r\n"** on MS-DOS, and so on), so that any version of the editor can read and write any of the supported formats. The value of the **format** parameter (which can be set to "**unix**", "**msdos**", "**macintosh**", etc.) determines which format is currently being used. If you are porting **xvi** to a system with a newline convention which isn't one of those currently supported (see the table called **tftable** in **fileio.c**) you may have to add a new entry to the table.

Unfortunately, the current design is not as general as it ought to be. If you happen to be porting to VMS, or some other system which doesn't use either a single character or a consecutive pair of characters to represent a newline, you will have quite a lot of work to do if you want to retain the facility for converting between file formats within the editor.

In any case, your system interface module should define **DEF_TFF** to be the index of the entry in **tftable** which represents the default format for your system. This is the value for **Pen(P_format)** which will be compiled into the parameter table.

4.5. Notes on Termcap Implementation

There exists a **termcap** implementation of the terminal interface, currently only used for the UNIX port. This module could quite easily be re-used for other systems if desired; the following routines would need to be defined by the system module:

void foutch(int c)
: Output a single character to the terminal. This must be implemented as a function, not a macro, because it is passed as a parameter into the **termcap** library.

void moutch(int c)
: Same as **foutch()** except that it can be implemented as a macro. This will be used by the **termcap** interface module to write characters to the screen.

void oflush(void)
: Flush buffered output to the terminal.

4.6. Entering/Leaving Visual Mode

Some facility is commonly necessary for the system interface module to be able to tell the terminal interface module to enter or exit *visual* mode. This might mean changing the terminal state between "raw" and "cooked" modes, or switching display pages. No specific interface for this is defined, although the standard UNIX and MS-DOS implementations do use such a facility, and the interface functions for both systems are identically defined.

4.7. Function Keys / Mouse Handling

Function key values are coded into a set of **#define**d constants in the file **ascii.h**; e.g. the value **K_UARROW** might be given as input when the keyboard up-arrow key has been pressed.

If the global variable **State** is not equal to **NORMAL**, all function keys except for a backspace key are invalid input. If an invalid key is pressed, the safest strategy may be to beep and wait for another key to be pressed. **NORMAL** is defined in **xvi.h**.

Another facility which may be provided is handling mouse input on systems where it is available. The strategy for interpreting mouse input is controlled by the **mouseclick()** function (in **mouse.c**); the idea is to make the strategy independent of any specific device interface. If a mouse button is pressed before a keyboard key is pressed, the following routine should be called:

mouseclick(int row, int column);

where row and column are the current co-ordinates, counted in character positions, of the mouse pointer within the screen or editing window. If the mouse is moved while a button is held down, the routine

mousedrag(int startrow, int endrow, int startcolumn, int endcolumn);

should be called with co-ordinates describing the movement. If the global variable **State** is not equal to **NORMAL**, mouse input can be ignored altogether.

All this will be considerably tidied up at a later stage, when we have proper **xvEvent** types for function keys and mouse actions.

4.8. Main

Finally, the system interface module must provide a **main()** function. This function must call **xvi_startup(vs, argc, argv, env)** at startup, with parameters as follows:

VirstScr *vs;
: This is a pointer to the **VirtScr** structure for the first window, or for the terminal screen.

int argc, char **argv;
: These are as for a **main()** function.

char *env;
: This is an environment string, normally the return value from **getenv("XVINIT")**. If the concept of environment variables does not exist, a string of the form "**source** *filename*" may be passed instead, so as to allow users to localise their usage of the editor.

The return value from **xvi_startup()** is a pointer, which will be used in future to identify the window for input events. For now, it should be stored in the **VirtScr**'s **pv_window** field.

Having called **xvi_startup()**, input events may then be passed to the editor by calling **xvi_handle_event** with a pointer to an **xvEvent** structure as the sole argument. This structure is defined as follows:

```
typedef struct event {
   enum {
      Ev_char,
      Ev_timeout
   }              ev_type;
   union {
      /* Ev_char: */
      int evu_inchar;

      /* Ev_timeout: */
   }              ev_u;
} xvEvent;

#define      ev_inchar   ev_u.evu_inchar
```

The **ev_type** field is a tag which identifies the type of event which has occurred. At present, only two events are supported: an input character from the user, and a timeout. The union which follows contains data associated with each event type; currently only the type **Ev_char** requires data, as may be seen. The **#define** for **ev_inchar** is provided purely for convenience.

The return value from **xvi_handle_event()** is a long integer value which is the time in milliseconds for which the editor is prepared to wait for more input. If no input arrives within that time, the function should be called again with an event of type **Ev_timeout**. The timeout value returned may be 0L, indicating that no timeout is necessary. It is very important that timeouts should actually be implemented because they are needed for the **preserve** facility.

Currently, if a keyboard interrupt is received, **xvi_handle_event()** need not be called (it should, in any case, never be called from an asynchronous interrupt or signal handler) but the global variable **kbdintr** should be set to a non-zero value.

5. DATA STRUCTURES

Structures used in **xvi** are all typedef'd, and all begin with a capital letter. They are defined in **xvi.h**. The following data structures are defined:

5.1. Line

This structure is used to hold a single text line. It contains forward and backward pointers which are connected together to form a two-way linked list. It also contains a pointer to an allocated text buffer, an integer recording the number of bytes allocated for the text, and the line number (an unsigned long). The text is null-terminated, and the space allocated for it may be grown but is never shrunk. The maximum size of this space is given by **MAX_LINE_LENGTH**.

The line number is used when showing line numbers on screen, but this is secondary to its main purpose of providing an ordering on lines; the ordering of two lines in a list may be established by simply comparing their line numbers (macros are available for this purpose; see later for details).

5.2. Buffer

This structure holds the internal representation of a file. It contains pointers to the linked list of lines which comprise the actual text. We always allocate an extra line at the beginning and the end, with line numbers 0 and **MAX_LINENO** respectively, in order to make the code which deals with this structure easier. The line numbers of **Line** structures in a **Buffer** are always maintained by code in **undo.c**, which is the only module which ever changes the text of a **Buffer**.

The **Buffer** structure also contains:

- flags, including readonly and modified
- current filename associated with the buffer
- temporary filename for buffer preservation
- space for the **mark** module to store information about marked lines
- space for the **undo** module to store information about the last change
- number of windows associated with the buffer

The following macros are used to find out certain information about **Line**s within **Buffers**:

lineno(Buffer *b, Line *l)
Returns the line number of the specified **Line**, which belongs to the specified **Buffer**.

earlier(Line *l1, Line *l2)
Returns **TRUE** if **l1** is earlier in the buffer than **l2**.

later(Line *l1, Line *l2)
Returns **TRUE** if **l1** is later in the buffer than **l2**.

is_lastline(Line *l1)
Returns **TRUE** if **l1** is the last line (i.e. the extra line at the end, not the last text line) of the buffer.

is_line0(Line *l1)
Returns **TRUE** if **l1** is the 0th line (i.e. the extra line at the start, not the first text line) of the buffer.

5.3. Posn

This structure is very simple; it contains a **Line** pointer and an integer index into the line's text, and is used to record a position within a buffer, e.g. the current cursor position.

These functions are available for operating on **Posn** structures:

gchar(Posn *)
Returns the character which is at the given position.

inc(Posn *)
Increments the given position, moving past end-of-line to the next line if necessary. The following type is returned:

```
enum mvtype {
   mv_NOMOVE,          /* at beginning or end of buffer */
   mv_SAMELINE,        /* still within same line */
   mv_CHLINE,          /* changed to different line */
   mv_EOL,             /* at terminating '\0' */
};
```

dec(Posn *)
As for **inc()** but decrements the position.

lt(Posn *p1, Posn *p2)
Returns **TRUE** if the position specified by **p1** is earlier in the buffer than that specified by **p2**.

5.4. Xviwin

This structure maps a screen window onto a **Buffer**. It contains:

- a pointer to the **Buffer** structure which it is mapped onto

- the cursor's *logical* position in the buffer (a **Posn** structure)
- the cursor's *physical* position in the window (row and column)
- information about size and location of screen window
- current text of status line
- forward and backward pointers to other windows

Note that there is at least one **Xviwin** for every **Buffer**.

When the editor was modified to support buffer windows, many global variables were moved into the **Buffer** and **Xviwin** structures; some were left as globals. For instance, the *undo* and *mark* facilities are obviously buffer-related, but *yank* is useful if it is global (actually static within its own module); it was decided that *search* and *redo* should also be global.

Some modules have their own internal static data structures; for instance, the **search** module remembers the last pattern searched for. Also, certain modules use data structures which are included in more global ones; e.g. each **Buffer** structure contains some data used only within **undo.c**. This is not very well structured, but in practice it's quite clean because we simply ensure that references to such structures are kept local to the module which "owns" them.

5.5. Mark

This data structure records a mark (defined by the **m** command). It contains a **Posn** and a character field to hold the letter which defines the mark. Each **Buffer** contains an array of these structures for holding alphabetic marks, plus one for the previous context mark (as used by the **''** and **``** commands). The file **mark.c** deals with marks.

5.6. Change

This structure records a single change which has been made to a buffer. It also contains a pointer, so that it may be formed into a list. See the discussion of **undo.c** below for further details.

5.7. Flexbuf

This structure is used to store text strings for which the length is unknown. The following operations are defined for this type. All functions take a Flexbuf pointer as a parameter.

flexnew(f)
: Initialise a Flexbuf; not needed for static Flexbufs.

flexclear(f)
: Truncate a Flexbuf to zero length, but don't free its storage.

flexdelete(f)
: Free all storage belonging to a Flexbuf.

flexempty(f)
: Return **TRUE** if the Flexbuf is empty.

flexlen(f)
: Return the number of characters in the Flexbuf.

flexrmchar(f)
: Remove the last character from a Flexbuf.

flexpopch(f)
: Remove the first character from a Flexbuf and return it.

flexgetstr(f)
Return a pointer to the string contained in the Flexbuf.

flexaddch(f, c)
Add the character **c** to the end of the Flexbuf.

lformat(f, fmt, ...)
A subset of **sprintf()** but for Flexbufs.

vformat(f, fmt, va_list)
A subset of **vsprintf()** but for Flexbufs.

The last two functions are especially useful, since they avoid the usual problems with the lack of bounds-checking in **sprintf()**. All code in the editor itself now uses Flexbufs to avoid the possibility of buffer overruns, and to reduce the size of the executable. Some OS-specific modules, however, may still use the **printf()** family. The subset of **printf**-like format specifiers implemented includes those for integers and strings, but not for floating-point numbers.

5.8. bool_t

A simple Boolean type; has values **TRUE** and **FALSE**, which are defined as 1 and 0 so as to be compatible with C comparison operators.

5.9. xvEvent

This type is defined in the previous section, since it forms part of the porting interface.

5.10. VirtScr

This type represents a virtual screen, and is constructed in a similar way to a *class*. It contains some function pointers which may be used to manipulate the screen in various ways, and some private data which is used by the implementation of the class.

The old terminal interface, which consisted of a set of disparate functions, is being replaced by the **VirtScr** interface; the first step in this process has been accomplished by the provision of a default **VirtScr** implementation using the old primitive functions. New, native, **VirtScr** implementations may now be coded, which will increase the efficiency of screen output.

As the final stage, a windowing implementation of the **VirtScr** class will be provided, using the underlying **VirtScr** implementations, and the window-handling code in the editor will be modified to that each occurrence of an **Xviwin** references its own **VirtScr**. It will then be possible to build a version of the editor which operates in a true windowing environment by using a separate screen window for each buffer, instead of the current vertical-split method.

A full definition of the **VirtScr** type will be found in the previous section.

5.11. Global Variables

There are only a few global variables in the editor. These are the important ones:

curbuf pointer to the current **Buffer**

curwin pointer to the current **Xviwin**

State the current *state* of the editor; controls what we do with input characters. The value is one of the following:

NORMAL The default state; **vi**-mode commands may be executed

INSERT Insert mode, i.e. characters typed get inserted into the current buffer

REPLACE Replace mode, characters in the buffer get overwritten by what is typed

CMDLINE Reading a colon-command, regular expression or pipe command

DISPLAY Displaying text, i.e. **:p** command, or **:set** or **:map** with no argument

echo This variable controls what output is currently displayable. It is used at various points within the editor to stop certain output which is either undesirable or sub-optimal. It must always be restored to its previous value after the code which changed it has finished what it is doing.

kbdintr This can be set to a non-zero value to indicate that an asynchronous user-generated interrupt (such as a keyboard interrupt) has occurred. See the discussion of event handling in the previous section.

6. SOURCE FILES

The header file **xvi.h** contains all the type definitions used within the editor, as well as function declarations etc.

The following source files form the primary interface to the editor:

startup.c Entry point for the editor. Deals with argument and option parsing and initial setup, calling module initialisation functions as necessary.

events.c Contains the routine **xvi_handle_event()**, which is the entry point for handling input to the editor; input is passed to different routines according to the **State** variable. Timeouts on input are also handled here, by calling appropriate routines in **map.c** or **preserve.c**.

edit.c Deals with insert and replace modes.

normal.c Handles normal-mode commands.

map.c This file is responsible for all input mapping (both set up by the **:map** command and internally for function-key mappings; it also implements a stuff-characters-into-the-input-stream function for use within the editor. This is used, for example, to implement command redo (but *not* to implement “undo” and “put” as in STEVIE).

Colon (**ex**-type) commands are handled by this group:

cmdline.c Decodes and executes colon commands.

ex_cmds1.c File-, **Buffer**- and **Xviwin**-related colon commands.

ex_cmds2.c Other colon commands (e.g. shell escape).

Screen updating is done within the following files:

screen.c Screen updating code, including handling of line-based entry (for colon commands, searches etc) as they are typed in, and display-mode stuff (for parameter displaying, **:g/re/p** etc).

cursor.c This file contains the single function **cursupdate()**, which is responsible for deciding where the physical screen cursor should be, according to the position of the logical cursor in the buffer and the position of the window onto that buffer. This routine is not very optimal, and will probably disappear in due course.

defscr.c This file contains the default implementation of the **VirtScr** class, on top of the old terminal/system interface.

status.c Functions to update the status line of a window; there are different functions to display file information (name, position etc.) and error/information messages.

These files deal with specific areas of functionality:

find.c Search functions: all kinds of searches, including character-based and word-based commands, sections, paragraphs, and the interface to "real" searching (which is actually done in **search.c**).

mark.c Provides primitives to record marks within a **Buffer**, and to find the marks again.

mouse.c Code to handle mice moving the cursor around and resizing windows.

param.[ch] Code to handle setting of, and access to, parameters. (These are things like **tabstops**, **autoindent**, etc.)

pipe.c Handles piping through system commands.

preserve.c File preservation routines.

search.c Code for pattern-searching in a buffer, and for substitutions and global execution. Uses **regexp.[ch]** for the actual regular expression stuff.

tags.c Routines to handle tags — for **:tag**, **-t** and **^]**.

undo.c Code to deal with doing and undoing; i.e. making and unmaking changes to a buffer. This is one of the more complex and delicate files.

yankput.c Code to deal with yanking and putting text, including named buffers.

while these files provide lower-level functions:

alloc.c Memory allocation routines.

ascii.[ch] Deals with the visual representation of special characters on the display (e.g. tabs, control chars).

buffers.c Routines dealing with the allocation and freeing of **Buffers**.

fileio.c File I/O routines; reading, writing, re-editing files. Also handling of the **format** parameter.

flexbuf.c Flexible-length character-buffer routines.

misccmds.c Miscellaneous functions.

movement.c Code to deal with moving the cursor around in the buffer, and scrolling the screen etc.

ptrfunc.[ch] Primitives to handle **Posn** structures; including various operators to compare positions in a text buffer.

regexp.[ch], regmagic.h
Regular-expression stuff, originally written by Henry Spencer (thanks Henry) and slightly hacked for use within **xvi**.

signal.c Handling of terminal-generated signals in an ANSI environment.

virtscr.h Virtual Screen interface definition. This is a new part of **xvi**, and is not yet fully completed. When it is finished, it will provide the ability to implement "native" versions of **xvi** under various windowing systems, in a clean and wholesome way. Currently there is a single instance of the **VirtScr** class, which is defined on top of the old system/terminal interface.

windows.c	Code to deal with creating, deleting, resizing windows.
version.c	Contains only the version string.