MINIX.XINIM,
Towards a Bi-Directional, Bi-Lingual UNIX™ Operating System

Gil Allon
Daniel M. Berry

Faculty of Computer Science, Technion
Haifa 32000, Israel

Abstract

This paper describes the design and construction of MINIX.XINIM, a bi-directional, bi-lingual version of MINIX, a mini-UNIX operating system. MINIX.XINIM is constructed from MINIX by modifying some of MINIX's device drivers so that they reverse all right-to-left text that passes through them. From this simple change, the entire kernel and all line-mode applications become bi-directional. While the version of MINIX.XINIM described here is for English and Hebrew, it can be easily used for any pair of left-to-right and right-to-left languages supported by the local input-output devices.

MINIX.XINIM,
לקראת מערכת הפעלה UNIX
דו־כיוונית ודו־שפתית

גיל אלון
דניאל ברי

הפקולטה למדעי המחשב, הטכניון
חיפה 32000, ישראל

תקציר

מאמר זה מתואר את תכנון ובנית MINIX.XINIM, שהיא גירסה דו־כיוונית ודו־שפתית של MINIX, מערכת הפעלה מיני־UNIX. MINIX.XINIM נבנה מ־MINIX באמצעות שינוי של כמה מתוכנות הממשק להתקן באופן שהם הופכים כל טקסט הנכתב מימין לשמאל שעובר דרכם. מהשינוי הפשוט הזה, הגרעין השלם וכל הישומים במודת שורה נעשים דו־כיוונית. עבור כל זוג שפות הנכתבות האחת משמאל לימין והאחרת מימין לשמאל ושנתמכות על־ידי התקני קלט־פלט מקומים.

MINIX.XINIM,
К двунаправленной двуязычной операционной системе UNIX

Гиль Алон
Даниэль М. Бэри

Факультет компьютерных наук, Технион
Хайфа 32000, Израиль

Аннотация

Настоящая статья описывает проектирование и построение MINIX.XINIM, двунаправленной и двуязычной версии MINIX'а (мини-UNIX'а). MINIX.XINIM построен из MINIX'а путём модификации драйверов ряда устройств MINIX'а так, что они обращают любой текст, проходящий через них. В результате этого простого преобразования ядро в целом и все действия в строковом режиме становятся двусторонними. Хотя описанная здесь версия MINIX.XINIM предназначена для пары английского языка-иврита, её легко можно использовать для любой пары языков, поддерживаемых локальными устройствами ввода-вывода, один из которых использует запись справа налево, а второй – слева направо.

1 INTRODUCTION

Computers, operating systems, and applications were developed primarily in English speaking countries. Hence the computers, the operating systems, and the applications were all geared to English; the alphabet supported by the machines is that used for English, the commands of the operating systems are English words, and e.g., applications dealing with names alphabetize them according to English rules. In the word processing area in particular, the first programs knew about English hyphenation rules, spelling rules, and punctuation rules.

As computing spreads throughout the world, the need for multi-lingual machines, operating systems, and applications grows. The problems begin as heavily accented Latin alphabets, with possibly additional letters are needed for many European countries. The problems grow as non-Latin alphabets are needed for countries in Eastern Europe, around the Mediterranean, and in Asia. This problem is compounded as the alphabets for countries in the Far East prove too big for one byte to uniquely encode each letter. The problems increase as it is observed that some of the Mediterranean and Asian languages are not written from left to right; some are written from right to left and others are written from top to bottom.

The focus of this work is on computers, operating systems and applications for English-Hebrew work. English is written from left to right with the Latin alphabet, for which there is a seven-bit standard encoding, called ASCII. Hebrew is written from right to left with the Hebrew alphabet, for which there is a seven-bit standard encoding. The encodings for each alphabet assign consecutive numbers to letters in alphabetical order. While this paper is focused on English-Hebrew work, many of its principles apply to the general multi-lingual situation and many of its details apply to any pair of languages with alphabets small enough for seven-bit encodings and whose writing directions oppose, e.g., French or Russian and Arabic or Farsi.

This paper first describes the requirements for a bi-directional, bi-lingual, English-Hebrew UNIX system. Then the lessons that were learned from past related work on formatters, editors, and other applications are examined in order to obtain a strategy for building this system. Based on this strategy, the goals of this work are stated, first in the most ambitious form, and then scaled down to be completable by one person as a prototype for the ambitious goals. The steps of the construction of the *completed*, running prototype are recounted with an emphasis on lessons learned and minds changed during the construction. Outputs from acutal sessions with the running prototype are exhibited to demonstrate that the prototype is behaving the way intended. Finally plans for a full bi-directional, multi-lingual UNIX system and for a bi-directional, multi-lingual X-windows system are described.

This paper is derived from a longer one of the same title obtainable from the second author at the address given above.

2 REQUIREMENTS FOR ENGLISH-HEBREW OPERATING SYSTEM

A single code is needed for representing all required characters. There is already such a code, a standard called ESCII in Israel. It is the standard Latin ASCII code in the first 128 codes and it is the seven-bit Hebrew code with the eighth bit turned on in the second 128 codes. Thus, there is a simple way to tell from each character in which language it is; look at its eighth bit. If it is 0, the character is Latin, and if it is 1, then the character is Hebrew. Many hardware devices made or adapted in Israel, especially terminals and line printers, accept this code.

It is necessary to add to the English-based operating system and the applications that run on top of it the ability to have (1) Hebrew text in files, (2) Hebrew file names, (3) Hebrew command names, (4) Hebrew error messages, (5) Hebrew prompts, (6) Hebrew input accepted, (7) Hebrew output generated, and (8) bi-directional output. Moreover, all of the above should co-exist with the extant English version. Having a character code that supports Hebrew takes care of Requirement 1 and 2 if the operating system does not balk at having characters with the eighth bit on in file names. If Requirement 2 is met, then Requirement 3 is met, because command names are interpreted as the name of the file that contains the program that does the command. Requirements 4, 5, 6, and 7 require changing the contents of string constants used by programs, in addition to, in the case of Requirement 6, not balking at input with the eighth bit on. Recall that accepting input may require the program to match possible input strings and generating output may require copying constant strings.

There are two methods of changing the contents of string constants within the program. If the string constants are hard-wired into the code, then the contents of the strings must be translated into Hebrew and the program must be recompiled with the new strings. If the strings are all stored in external files, and are read in by the program at start up or when they are needed, then only the strings in the files must be changed. The latter solution is preferred, as it means keeping only one copy of each stringless program and one copy of the strings file for each program and each language, as opposed to one copy of each whole program for each language. If the latter solution is adopted, then the ability to link a file to several names, one in English and one in Hebrew, allows the same program to be invoked with either an English or a Hebrew name.

The remaining requirement is Number 8. Arranging that an operating system and its applications meet Requirement 8 is the subject of this paper. Note that Requirement 8 is really language independent in that no particular knowledge of Hebrew is needed to implement it and it is needed for languages other than Hebrew. Even the other requirements, aside from the fact that one needs to know enough Hebrew to write good file names, command names, error messages, prompts, possible matches to inputs, and output messages in Hebrew, are also language independent. The needed changes have nothing to do with Hebrew, and they are needed for other languages as well.

3 LESSONS LEARNED FROM PAST WORK

The brute force way to adapt computer systems and software to a multi-lingual environment is to change each program individually. Students and system programmers at the Technion and the Hebrew University have started to do this with some more popular programs and have developed hcat, hmail, hmore, htroff, and vi.iv as bi-directional versions of cat, mail, more, the original troff, and vi, respectively, and ded, a bi-directional full screen editor for the DEC Hebrew vt100 terminals. They have also developed troffh and vih, as uni-directional Hebrew versions of the original troff and vi. However, the effort seems to have lost steam, as there remain many other applications for which it would be useful to have bi-directional or Hebrew versions, e.g., grep, gres, and sort, as well as databases. Examination of what happened shows that similar changes were being made to all the programs. These changes were not quite similar enough to be automatable, but similar enough to become a tedium for those starting the project. Basically, the project failed from boredom and the realization that the changes would have to be made to *every* program. The good software engineer, who is of course lazy, begins to wonder if there is a better way, say to do these changes once in such a way that all programs can share the effects.

3.1 Universal Lessons

Lessons can be learned from the published literature about the adaptation of word-processing software to the bi-horizontal-direction environment, in general, and to the English-Hebrew environment, in particular.

Becker's multi-lingual Xerox ViewPoint™ system [Becker1987], Buchman and Berry's ffortid™ [Buchman1985], using ideas from Gonczarowski's htroff [Gonczarowski1980], Knuth and MacKay's TEX/XET [Knuth1987], Tayli and Al-Salamah's Intelligent Arabic Workstation [Tayli1990], Habusha and Berry's vi.iv [Habusha1990] all have the following properties in common despite that the first is a WYSIWYG word-processor, the next two are batch formatters, and the third is a full-screen editor, all for handling bi-horizontal-direction text.

1. The storage of all files is in what is called *time*, *logical*, or *input order*, that is, the characters are stored in the order in which they are pronounced by a person reading the text.

2. Text is rearranged so that each language is printed in its own direction only at the time of output, whether on a screen or on some hard-copy printing device. The order of the text after layout, as seen on, e.g., a screen, is called *visual order*.

3. There are two independent directions involved in layout, the direction of the individual character and the direction of the document. A Latin character is generally considered a left-to-right character while a Hebrew or Arabic character is generally considered a right-to-left character. In a left-to-right document, the beginning of a printed line is its leftmost character and the end is its rightmost character. The concept of document has a temporal nature; one speaks of the current document direction. Note that while the character direction is implicit in the character and is therefore encoded in the file, the current document direction is a function of the application and is *not* encoded in the file. To see this, observe that a quoted passage of Shakespeare's work in an Arabic book may be considered a right-to-left document, containing left-to-right characters, if one does not mind that the paragraphs are indented from the right.

4. The algorithm to rearrange the text upon output, called the *layout* algorithm works on a line-by-line basis and can be applied locally to each line. The layout algorithm assumes a device that prints from left to right. The left-to-right language is called LR and the right-to-left language is called RL.

    for each line in the file **do**
        if the current document direction is left-to-right **then**

```
        reverse each contiguous sequence of RL characters in the line
    else (the current document direction is right-to-left)
        reverse the whole line about;
        reverse each contiguous sequence of LR characters in the line
    fi
od
```

As an example of the above concepts, consider the two lines in time order:

```
הוא_אמר_Hi Gil
and Dan_לגיל_ודן.
```

The underscored spaces and the period are Hebrew and the blank spaces are Latin. The two lines mean "*He said* Hi Gil and Dan *to Gil and Dan.* In a left-to-right document, these lines in visual order are:

```
_רמא_אוהHi Gil
and Dan.ןדו_ליגל_
```

In a right-to-left document, these lines in visual order are:

```
                 Hi Gil_רמא_אוה
            .ןדו_ליגל_and Dan
```

Storing the text in time order and reversing the right-to-left text only upon output flies in the face of the more common approach of reversing right-to-left text upon input and storing the file in visual order. Layout upon input is followed by alef-bet [Alef-Bet19??], Einstein [Einstein19??], MacInHebrew [Weinstein1986], Multi-Lingual Scribe [Gamma1984], vih [David19??], and WORDMILL [Intersoft1984]. However, layout upon output is more general because it allows changes to line lengths without having to reconstruct the input order first.

3.2 Formatters

In the case of the formatters, the layout algorithm is applied to an intermediate representation produced by the underlying left-to-right formatter after all line-breaking decisions have been made. The intermediate representation thus shows exactly what text is on what line based on a standard left-to-right formatting of text in time order. It is required that the intermediate representation show where the line boundaries are.

In the case of ditroff/ffortid, the standard device-independent output of the left-to-right formatter ditroff is a suitable intermediate representation, and the layout algorithm is embodied in a separate program, ffortid. The visual order output of ffortid is in the language of the intermediate representation. Thus the composition ditroff | ffortid is indistinguishable on both ends from ditroff. Thus, all ditroff pre- and post-processors work without change with ditroff|ffortid. In particular, one post-processor called a device-driver, whose job it is to print the formatted output on one particular device, works none the wiser of the true source of its input.

3.3 Editors

Most full-screen editors these days are divided into two parts, the command processor and the screen manager. The command processor obeys the commands and updates the file or the internal representation of the file. The screen manager keeps the screen up to date as an accurate depiction of one or more portions of the edited file. If the editor, e.g. vi, is divided this way, then the layout algorithm can be applied by the screen manager line-for-line on all lines affected by any change. Since the file is in time order, apart from a few new commands necessary to change language and document direction, no change is needed to the command processor. All that is required for the layout algorithm to be able to display each language in its own direction is that it can distinguish right-to-left text from left-to-right text in the file. This can be done by having special, non-printing, escape characters marking language changes, or by having the eighth bit off for one direction and the eighth bit on for the other direction. The former is more general, but requires minor changes to the command processor to treat the escape character properly. The latter requires no change to the command processor, other than to be *eighth-bit clean*, i.e., to leave the eighth bit alone, but is limited to handling only two directions of text with languages each of whose alphabets is smaller than 127 characters. Moreover, the solution adopted by one editor on a system must really be adopted by all other editors and applications. A file produced by one editor may be edited with another, and it may be submitted to applications.

Input files for applications are generally produced with the help of editors. It seems easier to make all applications eighth-bit clean than to make them all ignore language identifying escape characters. Finally, there is the problem of choosing an escape character that does not already have a meaning.

To be able to handle more than two alphabets or alphabets with more than 127 characters requires moving to 16 or more bit characters [Becker1984, Beebe1990]. With such large characters there is room in the character for the language code and thus no need for language-identifying escape characters

As mentioned, after the command processor executes each user-visible step of any command, the layout algorithm is applied by the screen manager to each line that is changed by the step just executed. Obviously, the screen manager is invited to use the terminal capabilities data base in order to find a minimal sequence of terminal commands that will cause the terminal to update itself to the correct appearance. If the screen manager is successful in this venture, it can avoid having to send over the full text of all lines that were modified by the step. However, logically it has applied the layout algorithm to each modified line and has sent these lines over to the screen.

3.4 Line Mode

The common denominator running through all of this software is that the layout algorithm is applied line-by-line to time-ordered text already broken into lines and for which it is possible to determine for each character its direction of printing. If the eighth bit method is used to mark the direction of printing for each character, and all programs are eighth-bit clean, then the only remaining requirement for applicability of the layout algorithm is that the text be broken into lines. The standard conventions of using the new-line character, the carriage-return character, or both to mark line boundaries can be followed. We call all software whose output is a sequence of lines so marked *line-mode* software. Many programs, e.g., compilers, filters, stream editors, etc., in general, cat, more, sort, etc. in specific, are line-mode. On the other hand, many programs are not. The most common classes of non-line mode programs are full-screen editors and windowing systems. They output commands that address specific points on the screen. The layout algorithm can be applied *externally* to the output of any line-mode program without change to the program. The layout algorithm cannot be applied externally to the output of non-line-mode programs. Instead, the layout algorithm must be incorporated internally, as was done to the vi full-screen editor to make vi.iv.

Therefore, it should be possible to change a whole operating system and any line-mode software that runs on top of it to be bi-directional simply by rewriting output device drivers to reorganize each output line as it is output using the layout algorithm. Of course, any programs that affect the behavior of these drivers, e.g., stty, will have to be changed. Figure 1 illustrates the structure of such a system.

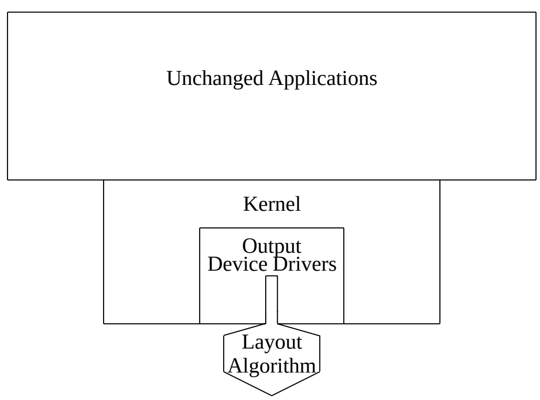

Figure 1: System Structure

For any program that requires no new features as a result of being applied in a bi-directional situation, then there is no more to do. If a program does require new features, then these must be implemented specifically. For example, if users are happy with the collating sequence of the new alphabets both internally and with respect to the Latin alphabet, then sort does not require any new features; if not, then sort requires new features to implement the desired ordering. Regardless of whether or not the new collation order options are added to sort, relative to the collation order, sort works correctly on multi-lingual lines because all lines are in time order. The most significant character of all lines is at the same end of the line. After correct sorting, the external layout algorithm takes care of displaying each output line correctly in visual order.

3.5 What is Left?

Even if this proposal works, there is still a lot to be done to make an entire system fully bi-lingual and bi-directional. First, from the original requirements, it is clear that the various string constants used for prompts, input comparison, output messages, error messages, etc. have be available in Hebrew. The proposal to put these strings in a file and make the program select strings from language-selected files solves this problem. Secondly, the non-line-mode programs have to be made bi-directional by brute force.

4 GOALS

4.1 Ultimate Goal

The ultimate goal of the research reported herein is to build a UNIX.XINU (no connection either to Mt. Xinu or to Comer's Xinu), a bi-directional, full-function UNIX system with the property that the kernel and *all* line-mode applications running on top of it are bi-directional with *no* change to the application software. It is assumed that all files, input, and output are in a code that permits the distinction between the languages of the two directions, with the right-to-left language having codes with the eighth bit on. Moreover, it is assumed that all files and input are in time order, and human-readable output is to be in visual order. If the terminals attached to the system handle Hebrew, then the system should support Hebrew input and output with *no* change to any application software, except if it should be desired that command names, prompts, acceptable input, output and error messages be in Hebrew as well! The approach will be to re-write the output device drivers for human-readable devices to apply the layout algorithm to each line of output that passes through them. On the other hand, device drivers for disk drives, tapes, diskettes, etc. should not be modified, because these devices are used for storage of files, which must be kept in time-order!

4.2 Realities

While the idea is nice and sounds like it should work, harsh experience has taught us that these sorts of things *never* work quite the way they are intended. There are things that could have been overlooked in getting the abstraction of the kernel and application software that was used to decide the validity of the approach. The use of the layout algorithm on every line of output could affect the performance of the system so much as to make the approach useless. Therefore, the first build of the system will be an experiment to find the corners in the reasoning, to evaluate the performance of the system, and, in general, to verify the useability of the resulting system.

UNIX systems, at least the ones for which we have the sources, are too big and too messy to expect that this modification will go without a hitch. Therefore, it would be useful to build a first version using a mini-UNIX system, which is both small enough and clean enough to work with easily and large enough to be a faithful model of UNIX. In other words, build a prototype first. If the prototype works as expected, then it is justified to begin a project to build a production quality, full UNIX.XINU.

Of all the mini-UNIX systems available, e.g., MINIX [Tanenbaum1987], Xinu [Comer1984], and XENIX™, MINIX is just right! It has a kernel fully compatible with the Version 7 UNIX system that used to be available from AT&T and from which System V and Berkeley variants were derived. It has a full complement of all the usual applications including a shell. The applications include both line-mode programs and interactive programs. Finally, the sources are available. A plus is the fact that MINIX runs on PCs. Thus, we can afford to dedicate an entire computer to the project without affecting a user population; indeed, a machine with *no* user population at all except the authors can be chosen.

4.3 Limited Goal

Therefore, the first sentence of the ultimate goal was modified into something more limited. The more limited goal for the project reported herein was to build MINIX.XINIM, a bi-directional, full-function MINIX system with the property that the kernel and *all* line-mode applications running on top of it are bi-directional with *no* change to the application software. The rest of the goals concerning files, input, and output, and not handling Hebrew text stand.

5 CONSTRUCTION

This section describes the steps followed to build MINIX.XINIM and the problems encountered. The topics are covered in the order that we encountered them to allow the reader to participate in our discovery of these problems and their solutions. In particular, the reader will see how goals were modified as a result of what we learned trying to meet the goals. Before jumping in, it is necessary to point out that we were using an IBM PC-AT™ to which a Hebrew character generator chip was added. This chip causes Hebrew characters to be generated on the screen in place of the funny faces in response to character codes greater than 127.

5.1 Making Code Eighth-Bit Clean

The first problem encountered was one encountered in the building of vi.iv. Hebrew uses character codes with eighth bit on so that it is possible to distinguish Hebrew characters from Latin characters, whose character codes have the eighth bit off. Many Latin-Hebrew terminals are built with this assumption, especially the newer ones that can display both lower-case Latin and Hebrew at the same time.

Many programs in and on UNIX and MINIX systems use the eighth bit of a character to keep Boolean flags associated with individual characters, and many programs destroy or erase the eighth bit even if they do not use it. Therefore, it was necessary to change a number of programs supplied with MINIX. These include grep, od, and sh. Among those that did not clobber the eighth bit were cat, cd, cp, gres (interestingly even though grep does clobber the eighth bit!), head, ls, mkdir, more, mv, pwd, rm, and tail.

The process of modifying programs so that they do not clobber the eighth bit is called making the software *eighth-bit clean*.

Note that AT&T started doing this with System V [AT&T1986] some time ago, SCO [SCO1988] now offers eighth-bit clean XENIX. Sun™ Microsystems' SunOS™ systems are all eighth-bit clean. In fact, these and all systems based on the X/Open Portability Guide [X/OPEN1987] and on the POSIX standard [POSIX1988] are all required to be eighth-bit clean specifically to allow international use.

5.2 Modifying the Shell

After having written one draft of the new device drivers, it became clear to the authors that it does not suffice to just rewrite output device drivers, because then only line-mode programs are affected. There are a few interactive programs that are very difficult to live without, e.g., editors and the shell. Without an editor, it is very difficult, but not impossible, to prepare files to test the main functions of MINIX.XINIM. One is reduced to doing

```
cat < > newfile
This is text that is going into newfile. The only editing function
available is backspacing WITHIN the current line.
^D
```

Without a shell it is almost impossible to run programs to test whether or not they continue to work. Since a colleague of ours, Uri Habusha has developed vi.iv, a bi-directional version of the vi full-screen editor, we accepted the limitation of no editor on MINIX (with the intention of using vi.iv via the network to prepare really big test files), but working without a full shell is well nigh impossible! So we had to produce a bi-directional version of the MINIX shell.

5.3 New Features

When designing the bi-directional shell, it behooves us to put as much of the new functionality that the new shell might need into the device drivers for no other reason to make these features available to other programs.

Once it is decided to modify shell to be bi-directional, many concepts from vi.iv find their way into the shell, because what is a shell other than an editor of commands? Thus, there are the concepts of *sessions*, the analogy of documents in an editing situation, and current input language, identifying the language of the next input character. As with vi.iv, it should be possible to set the session direction to be either left-to-right or right-to-left. Currently all screen properties of command editing sessions are set by the stty program, which sets flags and other data that are used by the screen device driver. All such features become available to any program writing to the screen. Thus, setting session direction should be done using the stty program. Therefore, two new mutually exclusive command line options were added to the stty program, `lr` and `rl`, with the former being the default. Saying "`stty rl`" makes the current session right-to-left, and saying “`stty lr`” makes the current session left-to-right.

Again, as with vi.iv, it should be possible to indicate in which language or direction is the next character. We adopted the vi.iv solution of having a global switchable setting that indicates the direction of all incoming characters until further notice. Once in, the direction of a character is encoded in the character and cannot be changed. The current input direction is changed by hitting the three keys, ALT, CTRL, and `x` simultaneously. The character `x` is used in this three-key code in order to be similar to vi.iv’s use of simultaneous hitting of CTRL and `x` for the same purpose. As in vi.iv it is possible to request to use a character, *c*, other than `x` in the three-key code by saying “`stty language` *c*”.

5.4 Echoing

We discovered that if the device-driver layout routines written for line-mode software are used during the operation of the shell, then the echoing behavior is not what is expected. When inputting text in the language whose direction opposes that of the current session, the text does not show up on the screen *until* either the end of the line is reached, by typing CR or LF, or the language is changed to agree with the direction of the current session. For example if the user is in a left-to-right session, and he or she switches to Hebrew from English, then the Hebrew input is not echoed to the screen until the end of the line or until he or she switches back to English. Obviously, the user would prefer to see each character as it is being typed, to avoid typing blindly. Upon careful reflection, we realized that the shell is not a line-mode program as the term was originally defined. It echoes as it receives input. Thus, it is outputting individual characters rather than whole lines. However, its outputting of individual characters is rather controlled compared to, say, a full-screen editor, which is outputting characters anywhere on the screen. A program that echoes input continues to output on the same line *until* the user hits CR or LF. As each character is input, the echoed output grows only by one new character. Unlike in an editor, in which editing commands can be applied to anywhere on the screen, an echoing program has a very limited repertoire of editing commands, i.e., backspace one character, backspace one word, and erase the whole line. Even with these commands, the echoing output stays within the current line. The editing commands do not allow backspacing through the beginning of the current line to the previous line. Given this very controlled flexibility that keeps the echoing on one line until the next CR or LF is given, it is not hard to conceive of applying the layout algorithm to the current state of the input line *after* each and every character of input, even if that input is an editing command that shrinks the input. It is also not hard to conceive of an incremental version of the layout algorithm that adjusts the echoed line after each input character according to what the character was, always able to fall back on building the whole current line if things get too complicated to construct incrementally.

A whole new class of application programs for which an external layout algorithm works has been found. The layout algorithm is made incremental so that it works for echoing programs, and the class of programs called line-mode include any whose output do not erase any character other than those, in reverse order of generation, between the last issued character and the last issued CR or LF. To handle this new class of line-mode programs, it is necessary to modify treatment of echo mode in device drivers so that as each character is printed the appearance of the line is calculated and if necessary redrawn in its entirety. Normally, it is necessary only to add the latest character to one side of the cursor, but then either the cursor moves or the line moves under the cursor. If the terminal has hardware commands to do these movements, then these commands should be issued by the device driver. The terminal capabilities data base, or termcap file, can be used to tell the device driver what commands the current terminal has. If such commands are not available, then it may be necessary for the device driver to redraw the entire line. However, at 9600 baud and more, this redrawing is just not noticeable by the human user.

5.5 Turning Layout On and Off

Finally, we observed in our own work with MINIX.XINIM, that the user may not wish to see the output in visual order. There are occasions, especially to answer picky questions about file contents, in which it is useful to be able to see files in time order. Therefore it was decided to be able to request that the layout algorithm not be invoked at all. One says “`stty -lrrl-flip`” to turn off layout, to see files in time order, and one says “`stty lrrl-flip`” to turn layout back on in order to see files in visual order.

5.6 Changed MINIX Modules

The number of modules of MINIX that had to be changed was surprisingly small, only four! The list below indicates the modules and the routines within them that had to be changed to make MINIX.XINIM as described in this paper.

1. `tty.c`: declaring variables for session state, scancodes, cooked mode controller, echo mode controller, echo, do cancel, setting session variables, cursor control, shift line back, output character, output conversion, input conversion, scroll screen, and flush
2. `klib88.s`: assembly routines for working with screen
3. `stty.c`
4. `sgtty.c`

It is a remarkably small list and gives hope that it will be quite easy to build a UNIX.XINU. With permission from Andrew Tanenbaum, the author of MINIX, the modified modules, instructions and diffs for eighth-bit cleaning, and instructions for builing MINIX.XINIM are available from the second author.

6 RESULTS

This section shows the outputs of several sequences of commands on files with mixed language contents.

6.1 more

The first examples show two files, lr1 and rl1, in various configurations. The contents of lr1 are (with translations from Hebrew indicated by oblique font text):

> `This paragraph is written in LR session.`
> *`This paragraph is written in-`* `LR session.`
> `This sentence begins with English,` *`continues in Hebrew`* `and ends with English.`
> *`This sentence begins in Hebrew`* `continues with English` *`and ends in Hebrew`*`.`
> `This sentence begins with English` *`and ends in Hebrew`*`.`
> *`This sentence begins in Hebrew`* `and ends with English.`

The punctuation and spaces are all in English. The contents of rl1 are:

> *`This paragraph is written in-`* `RL session.`
> `This paragraph is written in RL session.`
> *`This sentence begins in Hebrew`* `continues with English` *`and ends in Hebrew`*`.`
> `This sentence begins with English,` *`continues in Hebrew`* `and ends with English.`
> *`This sentence begins in Hebrew`* `and ends with English.`
> `This sentence begins with English` *`and ends in Hebrew`*`.`

The punctuation and spaces are all in Hebrew. The files are shown first in time-order and then in visual order. For each order, both session directions are illustrated. The application more is used for these demonstrations. Figure 2 shows the time-ordered contents of lr1 using more in a left-to-right session.

```
# more lr1
This paragraph is written in LR session.
הקסיפ וז תבתכנ -ב LR session.
This sentence begins with English, ךישממ תירבעב and ends with English.
טפשמ הז ליחתמ תירבעב continues with English םייתסמו תירבעב.
This sentence begins with English םייתסמו תירבעב.
טפשמ הז ליחתמ תירבעב and ends with English.
```

Figure 2: File lr1 in time order under left-to-right session with more

When MINIX.XINIM is booted, it comes up with the session left-to-right, the language English, and layout in effect. Thus, to get Figure 2, it is first necessary to execute “`stty -lrrl_flip`”. Then “`more lr1`” yields the figure. The words flow in a left-to-right order. The letters of the English words are printed in the correct order, but the letters of the Hebrew words are printed backwards.

Figure 3 shows the time-ordered contents of rl1 using more in a right-to-left session.

```
                                         1lr erom #
                    .noisses LR פיסקה זו נכתבת ב–
          .noisses LR ni nettirw si hpargarap sihT
```

```
                         .ומסתיים בעברית hsilgnE htiw seunitnoc משפט זה מתחיל בעברית
      .hsilgnE htiw sdne dna ממשיך בעברית ,hsilgnE htiw snigeb ecnetnes sihT
                                       .hsilgnE htiw sdne dna משפט זה מתחיל בעברית
                                   .ומסתיים בעברית hsilgnE htiw snigeb ecnetnes sihT
```

Figure 3: File rl1 in time order under right-to-left session with more

From the current state of MINIX.XINIM, it is necessary to change the session direction with "`stty rl`". Then "`more rl1`" yields the figure. The words flow in a right-to-left order. The letters of the Hebrew words are printed in the correct order, but the letters of the English words, including those of the command, are printed backwards.

To show files in visual order, it is then necessary to execute "`stty lrrl_flip`". Then same two files under the same session directions with the same command are shown in Figures 4 and 5.

```
# more lr1
This paragraph is written in LR session.
פיסקה זו נכתבת ב- LR session.
This sentence begins with English, ממשיך בעברית and ends with English.
משפט זה מתחיל בעברית continues with English ומסתיים בעברית.
This sentence begins with English ומסתיים בעברית.
משפט זה מתחיל בעברית and ends with English.
```

Figure 4: File lr1 in visual order under left-to-right session with more

```
                                                            # more rl1
                                                .RL session פיסקה זו נכתבת ב-
                                        .This paragraph is written in RL session
                        .ומסתיים בעברית continues with English משפט זה מתחיל בעברית
    .and ends with English ממשיך בעברית This sentence begins with English,
                                  .and ends with English משפט זה מתחיל בעברית
                              .ומסתיים בעברית This sentence begins with English
```

Figure 5: File rl1 in visual order under right-to-left session with more

In all of these figures, the words flow in the session's order *and* each word is printed in the correct direction for its language.

The full paper on which this paper is based shows examples using an unchanged cat, an eighth-bit-cleaned grep, and an unchanged gres.

6.2 sort

It was most gratifying to discover, as we had hoped, that the sort application did not have to be changed at all! The next set of examples illustrate the use of the sort program. It is able to sort files that contain mixed English and Hebrew. Because all Hebrew characters have the eighth bit set to 1 and English characters have the eighth bit set to 0, Hebrew characters appear negative to sort. As a consequence Hebrew sorts lower than English! Figure 6 shows the result of "`sort lr1`" in a left-to-right session.

```
# sort lr1
משפט זה מתחיל בעברית and ends with English.
משפט זה מתחיל בעברית continues with English ומסתיים בעברית.
פיסקה זו נכתבת ב- LR session.
This paragraph is written in LR session.
This sentence begins with English ומסתיים בעברית.
This sentence begins with English, ממשיך בעברית and ends with English.
```

Figure 6: Lines of lr1 sorted in visual order, under left-to-right session

Figure 7 shows the result of “`sort rl1`” in a right-to-left session.

```
                                          # sort rl1
                 .and ends with English משפט זה מתחיל בעברית
ומסתיים בעברית. continues with English משפט זה מתחיל בעברית
                                .RL session פיסקה זו נכתבת ב-
                      .This paragraph is written in RL session
              ומסתיים בעברית. This sentence begins with English
.and ends with English ממשיך בעברית This sentence begins with English,
```

Figure 7: Lines of rl1 sorted in visual order, under right-to-left session

To understand how the lines ended up the way they are, recall the time-ordered view of these files from Figures 2 and 3. The sort is applied to these lines to obtain the new ordering. Then each line is laid out according to the languages of its text and the current session direction.

6.3 Hebrew File Names

Hebrew file names happen! They are available with *no* special treatment. MINIX allows any character to be in a file name. Sometimes that can be a problem as if one of the characters in a file name has special meaning to the shell, it becomes very hard to get rid of the file! In the present case, this flexibility is useful. So, the command sequence, in time order

```
mkdir וךירדמ
cd וךירדמ
../cat > ץבוק
This is תירבעב םושר ומשש ץבוק.
ˆD
../ls
../ls -la
../ls ק*
../pwd
../od -cx ץבוק
```

gives rise to the time-ordered, left-to-right session output shown in Figure 8.

```
# mkdir מדריךו
# cd מדריךו
# ../cat > קובץ
This is קובץ ששמו רשום בעברית.
ˆD

# ../ls
קובץ
file

# ../ls -la
total 3
-rw-rw-rw-  1   root       31 Jan  1 00:41 קובץ
drwxrwxrwx  2   root       64 Jan  1 00:41 .
drwxrwxrwx  4   root      496 Jan  1 00:40 ..
-rw-rw-rw-  1   root        0 Jan  1 00:40 file

# ../ls ק*
קובץ

# ../pwd
/user/מדריךו
```

```
# ../od -cx קובץ
0000000    6854    7369    6920    2073    e5f7    f5e1    f9a0    eef9
0000000   T   h   i   s       i   s       ק   ו   ב   ץ       ש   ש   מ
0000020    a0e5    f9f8    ede5    e1a0    e1f2    e9f8    2efa    000a
0000020   ו       ר   ש   ו   ם       ב   ע   ב   ר   י   ת   .  \n
0000037
```

Figure 8: Working with Hebrew file names

The output of od shows that the contents of the file named קובץ is in time order and that the codes for the Hebrew letters have the eighth bits on. Note that the unchanged mkdir, cd, and ls are painlessly shown to have no problems with Hebrew and bi-directional text.

7 FUTURE WORK

The next step, of course, is to build UNIX.XINU. The steps to be followed are the following.

1. Get the sources of a UNIX system that meets the POSIX standard for nation independence. That is, all of its software is eighth-bit clean and uses external files to contain string constants.

2. Put a team of linguists to work to create versions of the string constants files in all languages to be supported by the system.

3. Modify the device drivers of all hard and soft copy, human readable output devices in the manner described in this paper. This involves adding various variants of the layout algorithm and dealing with echoing when the device supports it.

4. Add variants of the layout algorithm to all other screen output packages, e.g., curses, X-windows, SunView, NeWS, etc.

5. Modify any non line-mode, non interactive line-mode program that is deemed important enough to have a bi-directional version of it. We have already done this for the Berkeley version of vi.

6. Add Arabic as an alternative to Hebrew. This would mean altering the layout algorithm to determine the form of the letters based on their positions in the word, as is described by Becker, Mahjoub and Mandurah, and Tayli and Al-Salameh [Becker1987, Mahjoub19??, Tayli1990].

The full paper on which this one is based has a complete discussion of how the desired UNIX.XINU can be built from the Multi-National Language Supplement (MNLS) of UNIX System V produced by AT&T UNIX Software Operation Pacific in Tokyo with the help of AT&T UNIX Software Operation U.S. and AT&T UNIX Software Operation Europe in London [AT&T1987, Kogure1987] or from Sun Microsystems's addition to the Sun Operating System called HLE [SUN1990]. The full paper also discusses work that has been done by Gilad Granot, Eliad Klein, Amit Reisman, Haim Roman, and Amijai Shulman towards building a bi-directional X-windows [Scheifler1986] called X-windows.swodniw-X.

8 CONCLUSION

If one is willing to accept the changes that make software eighth-bit clean as nonsignificant changes, then the goal of building MINIX.XINIM as a bi-directional, full-function MINIX system so that the kernel and all line-mode applications running on top of it are bi-directional with no *significant* change to the applications has been achieved. This is not quite the goal that we started with; it specified *no* change to the applications. However, we went a bit beyond the goal in a significant way. More than just the line-mode programs are made bi-directional. By carefully dealing with echoing, also *interactive* line-mode programs were made bi-directional with no significant change to the application.

A natural question to ask is whether the performance of the system and the applications suffers. After all, now *every* line of output is subject to the overhead of layout. However, the fact is that we simply did not feel any difference. Probably the time to wait until the device is ready for the next character so dominates the time spent in the device driver that the layout overhead is just not observable.

On the other hand, note that we worked with an IBM PC clone, which has a memory-mapped terminal. A memory-mapped terminal is about a thousand times faster than a serial terminal. Note also that there is a considerable difference between line-mode output and interactive output. The first will be fast on almost any kind of terminal, but the latter may be slowed down on serial terminals, especially ones that do not have the ability to shift lines. Still one the other, other hand, note that in another project, we used the same layout algorithm in the screen manager of vi to make vi.iv. When vi.iv is used on DEC VT220 Hebrew terminals operating at 9600 baud, it feels no slower than vi itself. There too, the time to send the characters of a line to the terminal completely dominates the time to compute the view order appearance of the line.

The testing of MINIX.XINIM was a pleasant surprise! After thoroughly testing the changes to the device drivers through the first applications and the shell, we slowly began to test and use other applications. In the first few of these test uses we were apprehensive; would the application work correctly? Usually they did, and they worked the first time. We began to notice that in setting up for one test we inadvertently tested other programs, e.g,. after making a directory or a file, we did ls from habit. Only after seeing the results did we realize that we had just tested ls. We finally arrived at the point that we were just doing commands and *expecting* them to work. It is really a pleasure when a nice, theoretically sound idea really works when put to practice on real software!

Acknowledgements

The authors thank Lev Finkelshtein for the translation of the abstract into Russian.

ffortid is a trademark of Berry Computer Scientists. NeWS, Sun, SunOS, and SunView are trademarks of Sun Microsystems, Inc. PC-AT is a trademark of IBM Corporation. TEX is a trademark of the American Mathematical Society. UNIX is a trademark of AT&T Bell Laboratories. ViewPoint is a trademark of Xerox Corporation. X-windows is a trademark of the X Consortium. XENIX is a trademark of Microsoft Corporation.

References

[Alef-Bet19??] *Alef-Bet Manual*, 19??.

[AT&T1986] *System V Interface Definition*, Indianapolis, IN: Volume I, Issue 2, AT&T Customer Information Center, 1986.

[AT&T1987] *UNIX System V Multi-National Language Supplement*, Tokyo, Japan: Release 3.2, Product Overview, AT&T UNIX Software Operation Pacific, 1987.

[Becker1984] J.D. Becker, Multilingual Word Processing, *Scientific American*, 251(1): 96–107, July, 1984.

[Becker1987] J.D. Becker, Arabic Word Processing, *Communications of the ACM*, 30(7): 600–611, July, 1987.

[Beebe1990] N.H.F. Beebe, Character Set Encoding, *TUGboat*, 11(2): 171–175, 1990.

[Buchman1985] C. Buchman, D.M. Berry, and J. Gonczarowski, *DITROFF/FFORTID*, An Adaptation of the UNIX *DITROFF* for Formatting Bi-Directional Text, *ACM Transactions on Office Information Systems*, 3(4): 380–397, October, 1985.

[Comer1984] D. Comer, *Operating System Design: the Xinu Approach*, Englewood-Cliffs, NJ: Prentice-Hall, 1984.

[David19??] I. David, *Vih Manual Page*, Haifa, Israel: Technion, 19??.

[Einstein19??] *Einstein Manual*, 19??.

[Gamma1984] *Multi-Lingual Scribe*, Santa Monica, CA: Gamma Productions, 1984.

[Gonczarowski1980] J. Gonczarowski, *HNROFF/HTROFF Hebrew Formatters based on NROFF/TROFF*, Jerusalem, Israel: Computer Science Department, Hebrew University, 1980.

[Habusha1990] U. Habusha and D.M. Berry, vi.iv, a Bi-Directional Version of the vi Full-Screen Editor, *Electronic Publishing*, 3(2): 3–29, 1990.

[Intersoft1984] *WORDMILL User's Guide*, Jerusalem, Israel: Intersoft Software Engineering, Ltd., 1984.

[Knuth1987] D.E. Knuth and P. MacKay, Mixing Right-to-left Texts with Left-to-right Texts, *TUGboat*, 8(1): 14–25, 1987.

[Kogure1987] H. Kogure and R. McGowan, A UNIX® System V STREAMS TTY Implementation for Multiple Language Processing, *Proceedings of the Summer 1987 USENIX Conference*: 323–336, June, 1987.

[Mahjoub19??] A.H. Mahjoub and M.M. Mandurah, Current Issues and Future Directions in Computer Arabization, Technical Report, College of Computer and Information Science, King Saud University, 19??.

[POSIX1988] IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988, Technical Committee on Operating Systems of the IEEE Computer Society, 1988.

[Scheifler1986] R.W. Scheifler and J. Gettys, The X Window System, *ACM Transactions on Computer Graphics*, 5(2): 110–141, 1986.

[SCO1988] 8-bit Compatible Updates, *Discover—The Technical Bulletin of Santa Cruz Operation, Inc.*: 11, Santa Cruz Operation, November/December, 1988.

[SUN1990] *HLE 1.0 Product Description*, Mountain View, CA: Sun Microsystems, 1990.

[Tanenbaum1987] A.S. Tanenbaum, *Operating Systems: Design and Implementation*, Englewood-Cliffs, NJ: Prentice-Hall, 1987.

[Tayli1990] M. Tayli and A.I. Al-Salamah, Building Bilingual Microcomputer Systems, *Communications of the ACM*, 33(5): 495–504, May, 1990.

[Weinstein1986] J. Weinstein, *MacInHebrew*, Cambridge, MA: MIT Hillel House, 1986.

[X/OPEN1987] *X/OPEN Portability Guide*, Amsterdam: Elsevier Science, January, 1987.