ELECTRONIC PUBLISHING, VOL . 5(4), 163–208 (DECEMBER 1992)

Arabic formatting with ditroff/ffortid

JOHNY SROUJI (جوني سروجي, גוזני סרוג'י) AND DANIEL BERRY (دانيال بيري, דניאל ברי)

Computer Science Department
Technion
Haifa 32000
Israel

SUMMARY

This paper describes an Arabic formatting system that is able to format multilingual scientific documents, containing text in Arabic or Persian, as well as other languages, plus pictures, graphs, formulae, tables, bibliographical citations, and bibliographies. The system is an extension of ditroff/ffortid **that is already capable of handling Hebrew in the context of multilingual scientific documents.** ditroff/ffortid **itself is a collection of pre- and postprocessors for the** UNIX ditroff **(Device Independent Typesetter RunOFF) formatter. The new system is built without changing** ditroff **itself. The extension consists of a new preprocessor, fonts, and a modified existing postprocessor.**

The preprocessor transliterates from a phonetic rendition of Arabic using only the two cases of the Latin alphabet. The preprocessor assigns a position, stand-alone, connected-previous, connected-after, or connected-both, to each letter. It recognizes ligatures and assigns vertical positions to the optional diacritical marks. The preprocessor also permits input from a standard Arabic keyboard using the standard ASMO **encoding. In any case, the output has each positioned letter or ligature and each diacritical mark encoded according to the font's encoding scheme.**

The fonts are assumed to be designed to connect letters that should be connected when they are printed adjacent to each other.

The postprocessor is an enhancement of the ffortid **program that arranges for right-to-left printing of identified right-to-left fonts. The major enhancement is stretching final letters of lines or words instead of inserting extra inter-word spaces, in order to justify the text.**

As a self-test, this paper was formatted using the described system, and it contains many examples of text written in Arabic, Hebrew, and English.

مقدم

هذا المقال يصف برنامج لتوضيب اللغ العربيه والذي يمكن من توضيب نصوص علميه متعدد اللغات، محتويه ع نص بالعربيه والفارسيه بالاضاف للغات اخرى، رسومات، رسومات بيانيه، جداول، مصادر بيبليوغرافيه، وبيبليوغرافيا. لبرنامج هو تحسين لـ-ditroff/ffortid القادر ا ن عـ معالجـ العبريـ في وثا ـق متعـدد اللغات. ditroff/ffortid عبار عن قبل معالج (preprocessor) وبعد معالج (postprocessor) لبرنامج الصف في UNIX ، ditroff (Device Independent Typesetter

0894–3982/92/040163–46$28.00
Received 10 July 1992
Revised 1 February 1993

RunOFF). البرنامج الجديد مبني من دون دخال اي تغيير ء ditroff القا م. ا ضاف مكون ء الاغلب من قبل معالج جديد، طقم كامل من الحروف المطبعي، ونسخ محسن من بعد معالج قا م.

قبل المعالِج يترجم الماد التهجي لوثيق عربي، مكتوب باستعمال مجموعتي ا حرف (cases) في ا لف-باء اللاتيني. قبل المعالِج يعين لكل حرف موقعه بالكلم، غير متصل، نهاي (متصل مع قبله)، بداي (متصل مع بعده)، ووسط (متصل مع قبله وبعده). هو يضا يعين التركيبات (ligatures) ويقرر المكان العامودي للحركات (diacritical marks). قبل المعالِج يستطيع يضا ن يستقبل ماد من لوح مفاتيح قياسي ل حرف العربي (Arabic Keyboard)، والتي تستعمل شيفر ASMO القياسي. ء اي حال، بالنتاج (output)، لكل حرف او تركيب في الكلم ولكل حرك هنالك شيفر حسب النظام الشفري ل حرف.

شكليات الخط (font) مصمم بحيث ان الاحرف تتصل ببعضها عند اللزوم حين تكتب الواحد تلو ا خرى.

بعد المعالِج هو تحسين لبرنامج ffortid، والذي يقرر طباع من اليمين اليسار للاحرف المعرف ء انها تكتب من اليمين اليسار. التحسين الريسي هو مد (stretching) ا حرف ا خير با سطر او الكلمات بدلا من دخال الفراغات الزا د بين الكلمات بهدف تهميش النص (justification).

كفحص نفسي، هذا المقال صُف بواسط الجهاز الموصوف، وهو يحتوي ء مثل عديد لنص مكتوب بالعربي، العبري، والانگليزي.

תקציר

במאמר זה מתוארת מערכת סדר דפוס בערבית המאפשרת סדר דפוס במסמכים מדעיים רב-לשוניים, המכילים טקסט בערבית ופרסית בנוסף לשפות אחרות, ציורים, גרפים, משוואות, טבלאות, מקורות ביבליוגרפיים, וביבליוגרפיות. המערכת היא הרחבה של ditroff/ffortid שכבר מסוגלת לטפל בעברית בתוך מסמכים מדעיים רב-לשוניים. ditroff/ffortid הנה אוסף של קדם-

מעבדים ואחר-מעבדים עבור מסדר הדפוס של UNIX, ditroff (Device Independent Typesetter). המערכת החדשה נבנתה מבלי לשנות את ditroff הקיימת. ההרחבה מורכבת בעיקר מקדם-מעבד חדש, פונטים, וגירסה מעודכנת של אחר-מעבד קיים.
קדם המעבד מתרגם קלט פונטי של טקסט ערבי, הנכתב בהשתמש רק בשני המקרים (cases) של האלף-בית הלטיני. קדם המעבד קובע לכל אות את מיקומה במילה, עומד לבד, סוף (מקושר קודם), התחלה (מקושר אחר), אמצע (מקושר שניהם). הוא מזהה גם הרכבות (liga-tures) וקובע את המיקום האנכי של סימני הניקוד (diacritical marks). קדם המעבד יכול לקבל גם קלט ממקלדת סטנדרטית של טקסט ערבי (Arabic Keyboard), המשתמשת בקידוד הסטנדרטי ASMO בכל מקרה, בפלט, לכל אות או הרכבה הממוקמות במילה ולכל נקודה יש קוד לפי קידוד של הפונט (font).
הפונטים מתוכננים כך שהאותיות תתחברנה ביחד במקרה הצורך כאשר הן מודפסות האחת ליד השנייה.
אחר המעבד הוא הרחבה של התכנית ffortid הקובעת הדפסה מימין לשמאל של אותיות הפונטים המזוהים כנכתבים מימין לשמאל. ההרחבה העיקרית היא מתיחת (stretching) האותיות האחרונות בשורות או במילים במקום הכנסת רווחים נוספים בין המילים כדי ליצור את שולי הטקסט (justification).
כבדיקה עצמית, המאמר הזה מסודר דפוס בעזרת המערכת המתוארת, והוא מכיל דוגמאות רבות של טקסט הכתוב בערבית, עברית, ואנגלית.

KEY WORDS Arabic Bidirectional Formatting Multilingual Troff

1 INTRODUCTION

With computers spreading all over the world, there is a clear need for word-processing software to be made available in languages other than English.

Basically the history is as follows. The first computers were developed in English-speaking countries, and the first mass-marketing of computers was in these countries. Computers spread next to countries whose languages are written with the Latin alphabet, but some minor fudging is needed for accents, such as ´, `, ˇ, ¨, and ¯, and for unusual letters, such as ß, æ, Æ, ø, and Ø, which do not appear in English. Finally computers have spread to countries with totally different alphabets, such as the Arabic-Persian family, the Chinese-Japanese-Korean family, the Cyrillic family, Greek, the Hebrew family, and the Hindi family. In some cases the alphabets are very large, so large that one byte is not enough to encode all the characters. These include, of course, the Chinese-Japanese-Korean family. In some cases, the languages are written in other directions. These include the right-to-left languages from the Arabic-Persian family and from the Hebrew family. These also include languages written from top to bottom from the Chinese-Japanese-Korean family.

For word-processing software, there is a need for formatters and editors on the batch side and WYSIWYG processors on the interactive side.

The goal of the research that yielded the software described in this paper is to produce a complete environment for preparation, proofing, and printing of technical and non-technical multilingual documents. We need to be able to edit, preview, and typeset documents with all the hallmarks of technical papers including bibliographies and citations, formulae, tables, indexes, program code, and pictures. The pictures can be either filled, line-drawn figures or half-tones. Among the line-drawn figures are plots, flow diagrams, flow charts, graphs, trees, and data structures.

The software should be able to handle text in a wide variety of alphabets in all the known writing directions. These include the left-to-right languages written with the Cyrillic, Greek, Hindi, and Latin alphabets, the right-to-left languages written with the

Arabic, Persian, and Hebrew alphabets, and the top-to-bottom languages written with the Chinese, Japanese, and Korean alphabets. Any alphabet not specifically listed should *not* be construed as excluded.

The software should work in the increasingly popular UNIX environment. The main reasons for this requirement are that

1. the authors' various organizations are all UNIX shops, and
2. there is a variety of existing software in source form that solves most of the problem and that can be *reused* to provide significant leverage towards a full solution.

Software exists on UNIX environments that is capable of previewing and typesetting technical and non-technical documents with bibliographies and citations, formulae, tables, indexes, program code, and pictures, both line and half-tone, in all the known writing directions, left-to-right, right-to-left, and top-to-bottom, with a wide variety of fonts for Latin-based, Chinese-based, and Hebrew-based languages; some of this software is used to typeset articles for this journal. The specific goal of the work described in this paper is to extend this formatting software to be able to format such documents containing text in the Arabic-Persian family of languages. Because the first author is a native Arabic speaker, the focus of this work is on Arabic. Attention is paid to handling Persian and Urdu when possible, and if not, then at least to not excluding later extensions to handle them by native experts.

Given that a self-test has become *de rigueur* for formatting papers, this paper was typeset using the software described herein, using the command lines

```
refer -e -n -p ~/.refsidx paper > paper.ref
psfig paper.ref | sed -f cross.reference | \
chem | pic | atrn -b -l0 | tbl | eqn | \
troffort -r13 17 22 42 -a42 -sa -IhD -IAN -t > paper.ps
```

What is new in this command line are the invocations of atrn, for Arabic transliteration preserving the original line breaks (**–b**) with ligature level 0 (**–l0**), and of troffort, for bidirectional formatting with the fonts in positions 12, 17, 22, and 42 specified as right-to-left and the font in position 42 specified as Arabic with stretching of all words (**–sa**) and inclusion of the POSTSCRIPT definitions of fonts hD and AN.

Most of the figures were done with pic, tbl, eqn, chem, or pure formatted text. Figure 4 was scanned in and converted to encapsulated POSTSCRIPT with the help of Adobe Illustrator. Figures 5 and 6, and the example involving writing a complete word on top of a stretched letter, are manually programmed encapsulated POSTSCRIPT documents. All of these encapsulated POSTSCRIPT documents are included into the paper with the help of psfig. The document included in Figure 5 uses an experimental dynamic font. The reason Figure 6 was included via psfig rather than typeset as normal text is that providing the five Arabic fonts to typeset it normally makes the POSTSCRIPT document sent to the printer too big. By preparing it as a separate encapsulated POSTSCRIPT document, it was possible to whittle the fonts down to just what is necessary to make the figure. The second and third of the stretching examples of Section 8.2.4 are typeset as separate documents with the described software, because only one kind of stretching can be in effect

for all Arabic fonts in a single document. Their POSTSCRIPT outputs are interpolated into that of the main document by use of an editor.

Details omitted from this paper owing to space limitations are in a complete technical report of the same title available from the second author.

2 ARABIC LANGUAGE AND ITS FORMATTING PROBLEMS

2.1 Arabic language and computerization

The Arabic language is the main language in the Middle East, the mother tongue of about 200 million people in 21 countries, one of the five official languages of the United Nations, and one of the two official languages in the state of Israel, where the authors live. Moreover, the same alphabet is used, with minor changes, both additions and subtractions, in several other languages including Persian, Kazak, Kirghiz, Malay, old Turkish, Uighur, and Urdu. The geographic influence of the language is widespread.

There has been a large effort in recent years to bring the benefits of computerization to the Arabic world. This Arabization effort, described in a variety of papers in recent conferences in the Middle East and elsewhere [1–5], has yielded hardware that can store, read, print, and enter Arabic, Persian, Kazak, Kirghiz, and Uighur text [6, 7]. It has also yielded databases, spreadsheets, and word-processing applications, that can work with the same [7, 8].

2.2 Arabic alphabet and implications for processing

Arabic is an ancient language that originated from the Aramaic language that was used by the Nabateans and, like the other languages of Semitic origins, such as Hebrew, it is written from right to left. However, numerals[1] are written with the most significant digit to the left (i.e., what is commonly called from left to right).

2.2.1 Letters that connect and change form

In Arabic and related languages, the shape of letters depends on their positions within words. In Hebrew, in which characters are disconnected, only five letters change form according to their position within a word, and they change form only when they are last in a word. In Arabic, letters are written mostly connected and, as a consequence, nearly all letters change form according to position within a word. There are up to four different forms for each letter, namely, stand-alone, connecting-before, connecting-after, and connecting-both. The forms adopted by the letters are quite natural for the hand to produce when the hand is writing in a continuous flow. Therefore, for a fluent writer, the positions just happen as the letters are being written, much the same way as the lead stems of Latin letters change to accommodate a preceding "*o*" without the writer really having to think about it.

The Arabic alphabet consists of 31 letters and five vowels (see the table of Appendix II). Because the letters alef, taa_marbouta, and alef_maksura are really other versions of

[1] The designation of the standard numerals written in Latin alphabetic text as Arabic is misleading, as these numerals bear no resemblance to those actually written in the Arabic language.

other letters, grammatically there are only 28 letters. Thus, the table of Appendix II shows the Arabic alphabet as it must appear to any formatting software that is obliged to treat different versions and forms of the same grammatical letter as different characters and that is obliged to treat a diacritical mark as a character. It also has the additional characters that are needed to print Persian and international text.

The table of Appendix II also shows the different forms for each of the letters. Most letters have four forms. The four forms of the letter ب, which are ب, بـ, ـب, and ـبـ, appear in the words كتاب, كَتَبَ, بحر, and جبان. There are letters, e.g., د, that naturally do not connect to the following letter because of where they end. These letters have only two forms, stand-alone and connecting-before. Operationally, it will prove convenient to treat all letters as having four forms. In the case of a two-form letter, the connecting-after form is made a duplicate of the stand-alone form and the connecting-both form is made a duplicate of the connecting-before form.

2.2.2 Diacritical marks

The use of diacritical marks or vowels in Arabic, as in Hebrew, is optional. In normal, everyday text, a diacritical mark would be used only in the rare case in which it would be hard to identify the intended word from the letters and the context. For example, consider the word كتب written without diacritical marks. It could be either كَتَبَ ((he) wrote) or كُتبٌ (book). Normally it is quite easy to distinguish which of these is intended by the position in a sentence. The former is a verb and the latter is a noun. There is other text, either poetry or books for children, in which a full complement of diacritical marks is used. In the former case, word order is often inverted, and in the latter case, the young readers do not know the contexts.

A diacritical mark is written either above or below the letter after which its vowel is pronounced and also affects the accent or stress of that letter. Figure 1 shows the Arabic letter د with all the possible vowels. Below each is its pronunciation expressed in Latin letters. The input of diacritical marks should be allowed, and each should be printed

دَ	دُ	دِ	دً	دٌ	دٍ	دْ
da	du	di	dan	don	den	d
د	د	د	د	د	د	دّ
dda	ddu	ddi	ddan	ddon	dden	dd

Figure 1. Vocalizations of one letter

at the proper height above or depth below the letter that it follows in pronunciation. Here the proper height or depth is determined by the bounding box of the letter itself.

A glance at the table of Appendix II shows that some letters differ graphically from others only by the addition of some dots. These dots are part of the letters and should not

be considered diacritical marks. In fact, to the formatting software, these dots are irrelevant. Only to the font software might these be relevant, as glyphs might be built by calling subroutines that draw the different parts.

2.2.3 Ligatures

Arabic has ligatures, i.e., characters created by merging at least two others. The most common ligature in Arabic and its sibling languages is the ﻻ (lam-alif) created by merging the ل (lam) and the ا (alif). Grammatically, the lam-alif is not a letter; it is two letters, and words containing it are treated grammatically as containing a lam followed by an alif. A ligature is created and used solely to improve the calligraphic appearance of the text. Therefore, strictly speaking, forming ligatures is optional. The lam-alif is the most common ligature and is used in place of the individual letters in sequence virtually every time. It has become, for all practical purposes, obligatory. One section of the table of Appendix II lists the ligatures supported by this software. The optional ligatures, e.g., لم (lam-mim), formed from the ل (lam), and the م (mim) are used less frequently. Figure 2 shows, in the last steps of lines (a) and (c), words involving the ligatures lam-mim and lam-alif. In the figure, the لم is recognized as a ligature in (a), and is not recognized as a ligature in (b). Because the lam-mim is only an optional ligature, either is acceptable. The لا is recognized as a ligature in (c) as is required. Were it not to be recognized as a ligature, as in (d), one would obtain an impossible word containing the unacceptable construction لا.

Obviously, the formatter must treat a ligature as a separate character to be printed as any other letter. Typeset Latin text has ligatures, the most famous being “fi” which is used to avoid the two ugly, beady eyeballs staring at the reader when the ligature is not used, viz. “fi”. While the reader reads the “fi” as two letters, “f” and “i”, the formatting software considers the “fi” as another character. Of course, most formatting applications that provide ligatures do so automatically; the user enters “f” followed by “i” and the software replaces them, if they are still together after hyphenation, by the ligature character “fi”. Any Arabic wordprocessing software worth its salt should provide a similar service. Accordingly, Figure 2 also shows the steps in arriving at the final form of two words involving the ligatures lam-mim and lam-alif.

Arabic has an interesting property in connection with the optional ligatures. Assume that it has been decided for a document to form a ligature for a particular ordered pair of letters. Then sometimes, whether that ligature is formed in a particular place depends on the positions of the two original letters in the word. Take, for example, the optional lam-mim ligature formed from lam and mim. The lam and mim are joined into a unit only when the lam stands in a connected-after position and the mim is in a connected-before or a connected-both position. This is because the lam-mim is available only in connecting-after and stand-alone forms. See Figure 3 for the four cases of the ordered pair, lam and mim. لم is recognized as a ligature only in the first two cases.

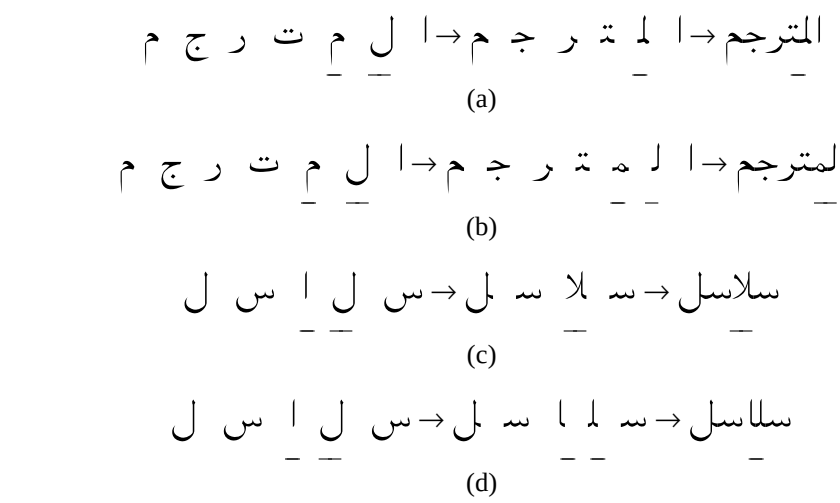


Figure 2. Steps of ligature identification

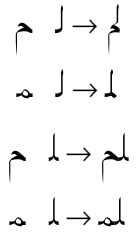


Figure 3. Four cases of the lam-mim combination

2.2.4 Justification, hyphenation, and stretching

In Arabic, typeset text is usually right and left justified. However, there is no hyphenation that can be used to make the job easier. The reason that there is no hyphenation is that hyphenation would mess up the whole positioning system causing two internal letters to behave as ending and beginning letters of words, causing a very strange appearance of the letters in what might be a very familiar word.

For languages with non-connecting letters, the usual method to achieve justification on both sides of a line is to insert extra white space in between words so that the list of words that will fit on a line are spread out to be flush at both ends. Usually the spacing between words is constant for the line, but sometimes extra white space is put after the end of a sentence. In addition, some algorithms, e.g., for TEX [9], try to make the spacing uniform over larger units of text than just the line. In Arabic, in contrast, the more usual treatment is to stretch the last letter or the approach to the last letter if that letter or

approach can be stretched. This stretching is called كشيد (*keshide*). "Keshide" is actually a Persian word derived from the verb كشيدن (keshidan), which means "to stretch".

There appear to be no formal laws specifying when, how, and how much to stretch letters. Instead, calligraphers decide to stretch according to aesthetic considerations. Basically stretching of last letters happens because a calligrapher writing with ink cannot predict the spacing to use between words until he or she reaches the end of the line; at that point there is nothing left to do but stretch the last letter. Someone who is writing with ink does not have the lookahead that a computer does! Lack of lookahead notwithstanding, examples in Section 8.2.4 later in the paper, show that stretching in Arabic is a natural thing and yields a nicer appearance than does spreading the words.

There are two main ways to stretch. One way is to stretch the connection to the letters. As an example, the word الجنـــــ is obtained by stretching the connection to the letter in the word الجنـ by 12 points. The other way to stretch is in fact to stretch the letters themselves. Generally, only those letters with large, mostly horizontal, strokes, such as ك, گ, ف, ڤ, ق, ب, ت, ث, ك, پ, ل, س, ش, ص, ض, ن, ي, and ى, are stretched. Figure 4, taken from an Egyptian text on Arabic calligraphy [10, 11], shows the letters ب, ف, and a variation of گ, with and without stretching. The unstretched versions of the letters are said to be 5 points wide (the point is the width of the dot that appears in two of the letters), and the stretched versions are 11 points wide. This figure also shows the importance of aesthetics in stretching, an importance that precludes clear laws. Because the three letters that are stretched are structurally similar, for appearance's sake, they had to be stretched the same amount rather than individual amounts according to the needed justification. For this kind of situation, the human calligrapher must exercise lookahead.

In manual calligraphy, the preference is for stretching letters themselves, but both methods of keshide are used. Some letters, i.e., those with no horizontal part, e.g., ا, are just not stretchable. It is sometimes not aesthetic to stretch a particular letter. On the other hand, sometimes the last letter is not connecting-before, so there is no connection to stretch. If both happen in the last word of a particular line, then the next-to-last letter or its connection might be stretched.

In electronic publishing with the standard kind of fonts that are available, the current preference is for stretching the connection to a letter, because if the letters connect at the same baseline, it is easy to provide a filler situated at the baseline, touching both vertical boundaries of the bounding box and as wide as the standard stem of the letters. Figure 5 shows the connecting-after and the connecting-before forms of the letter ب connected without and with one such filler between them in lines (b) and (c) respectively. line (a) shows how a connecting form of a letter meets its bounding box on the connecting side and how there is white space between a letter and the bounding box on the non-connecting side. Stretching a letter itself requires a dynamic font in which the width of a character may vary from showing to showing, even though its point size and stem

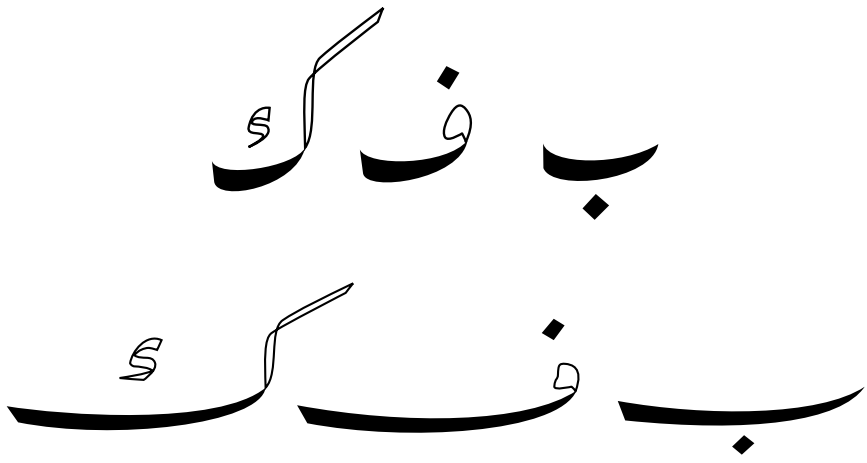


Figure 4. Non-stretched and stretched letters

thickness may not change. André [12] shows how to make such fonts, and we are in the process of making a dynamic version of the font used herein. lines (d), (e), and (f) of Figure 5 were printed using a dynamic, parameterized version of the connecting-before form of ب, for which the parameter of the glyph is the additional width. The three lines were obtained with the parameter being zero, the width of one dot (the diamond under the letter), and twice the width of one dot, respectively. Unfortunately, however, a document filled with such characters takes forever to print, because character cacheing has to be turned off to allow the bitmap of a character to be computed each time it is printed.

2.2.5 Calligraphic styles

Arabic is famous for its various beautiful calligraphic styles. The differences between the styles is in the way of writing the letters and in the amount of overlap between neighboring characters. Some of the styles even permit the writing of complete words on top of the last letter of the previous word. A shining example of this is the assembly of the two words صدق followed by الله. It is customary to write the الله on top of a stretched ق to yield:

صدقﷲ

In electronic publishing, this sort of thing can be done if the font being used has the construction available as a single, special character or if the characters making up the construction can be algorithmically distorted to the right shape to be used as pieces to build the construction.

Over the past thousand years, a number of calligraphic styles have grown in popularity and are quite standard these days. These include the fonts listed in Figure 6. The main Arabic font used in this paper is Naskh. In the figure, if we have a font available for the calligraphic style, we use it to write its own name; otherwise we use Naskh. Such a

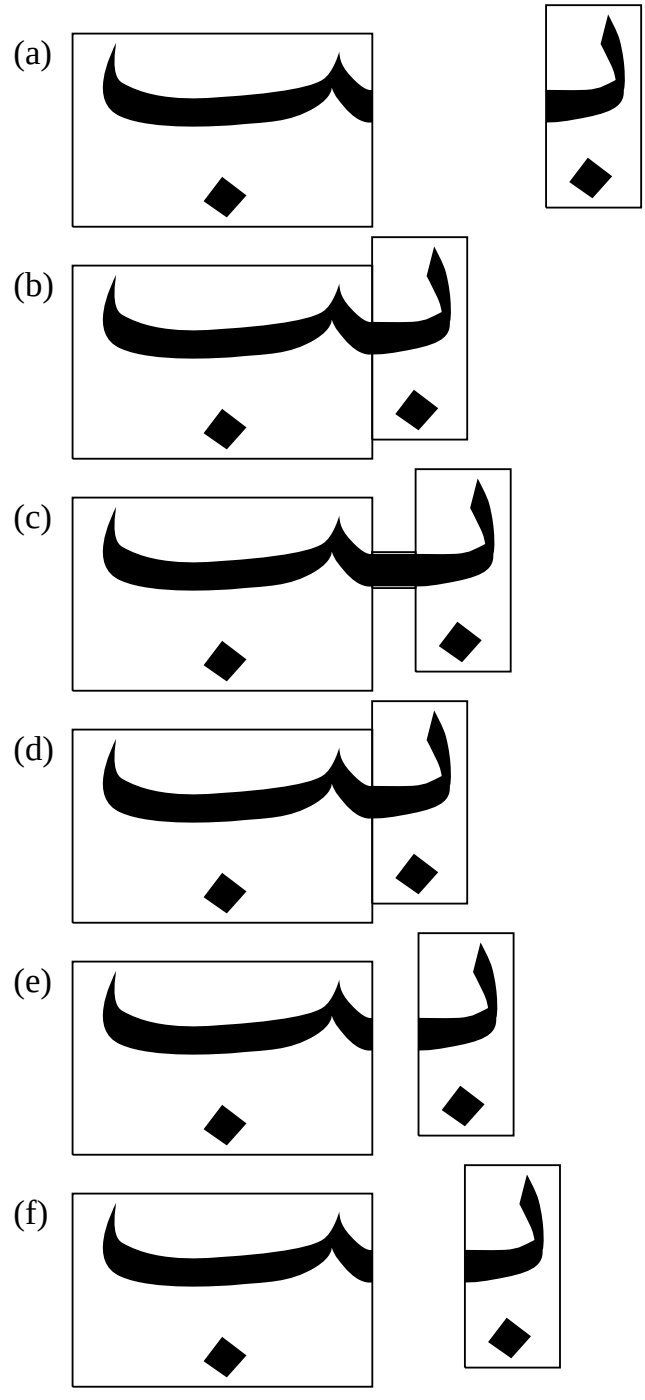


Figure 5. Connecting letters, fillers, and dynamic letters

default use of Naskh is marked in the figure with an asterisk. The available fonts can be used with ditroff/ffortid; it was required to reorganize these fonts to have the same encoding for the glyphs as does the Naskh font.

Baghdadi — بغدادي

Farsi * — فارسي

Diwany * — ديواني

Geezah — جيزة

Kufi — كوفي

Nadeem — نديم

Naskh — نسخ

Requ'ah * — رقعه

Tholoth * — ثلث

Figure 6. Calligraphic styles

2.2.6 Character codes

In the computerization efforts for Arabic and Persian, standards have emerged for codes for letters. Therefore, it is possible now to insist that the code for each letter in each language accepted by any processor should be according to one of the standards for that language. For Arabic, this code should be ASMO, for Persian, this code should be ISCII, for Hebrew, this code should be ESCII, and for English, this code should be ASCII.

3 GENERAL REQUIREMENTS FOR ARABIC FORMATTING

On the basis of the above discussion, it is possible to state the requirements for a multilingual formatting system that formats Arabic.

1. Solve all of the Arabic processing problems mentioned in Section 2.
2. Be user-friendly.
3. Produce book-quality output.
4. Whatever processing, e.g., identifying positions of letters, can be automated should be automated. Whatever should be left to the user, e.g., deciding on the level of ligaturing, is left to the user.
5. Permit formatting of scientific and technical text and all of the document entities that go with them, including formulae, tables, graphs, etc.

4 SOFTWARE ENGINEERING ASPECTS

Our motto is, "A good software engineer is a lazy one!" An existing formatting system should be used as much as possible. It is good if the new software is user-level compatible with the old; it is better if existing code is modified to obtain the new software; it is best if existing code is externally extended to obtain the new software.

Given that the authors' preference is a UNIX environment, then the question to ask is what complete formatting environments exist on UNIX platforms? These can be divided into two classes: WYSIWYG (what you see is what you get) and batch. Examples of UNIX-based WYSIWYG formatters with the most functionality are Interleaf [13] and FrameMaker [14]. The problem is that WYSIWYG formatters are of necessity monolithic programs. All their processing must be in the main program. They are interactive and must compute a new image of the document after each editing change. As a consequence they cannot make use of pre- and postprocessors to do some of their work. Adding new features requires opening up the main program and adding the new features in the midst of all existing functionality, and we do not have access to their source code.

The serious candidate batch systems were ditroff [15] and TEX [16, 17]. They have sufficient basic functionality for doing scientific documents, although, for reasons to be explained later, there is a serious deficiency in the latter. Both ditroff and TEX have been extended, albeit in different manners, to handle bidirectional text.

Figure 7 shows the flow of the current ditroff System with all pre- and postprocessors known to these authors. See References 18 through 36 for more details on each. This system offers hope of implementing new functionality simply by inserting new pre- and postprocessors. This hope arises from the UNIX philosophy of having separate language processors for each language, each understanding part of the job, and leaving all the rest to the others. Here, "language" means not only natural language, but also a notation for expressing some unit of the document, such as a formula. Each processor is easily modified independently of the others. Best of all, existing pre- and postprocessors and macro packages continue to work as each new processor or macro package is added! There is also an economic issue involved. By adding new features via new, separate processors, no source license is needed for ditroff. All that is needed to write a pre- or postprocessor is the specification of the input or output of ditroff.

The bidirectional version of ditroff, ditroff/ffortid, was built in this modular manner by adding a postprocessor, ffortid, to an unchanged ditroff. ffortid is responsible for printing right-to-left text from right to left, while ditroff treats all text as if it were written from left to right. Because ditroff was not modified at all, all ditroff preprocessors and macro packages work for ditroff/ffortid. Moreover, since ffortid output looks like ditroff output, all ditroff postprocessors work for ditroff/ffortid.

The question is, "Why not TEX?" Why do we insist on using old-fashioned, braindamaged troff technology? TEX also suffers from the same sort monolithism as are suffered by WYSIWYG systems; all of the table and formula processing are part of the main program. TEX is not really pipeable, so use of pre- and postprocessors is inconvenient. There are more serious problems, problems of inadequate information, that are discussed fully in Section 10. As a result of these problems, TEX's bidirectional version, TEX/XETT [37] has to be built as a modification of TEX and not simply by adding a postprocessor to an unmodified TEX, as was done to obtain ditroff/ffortid from ditroff.

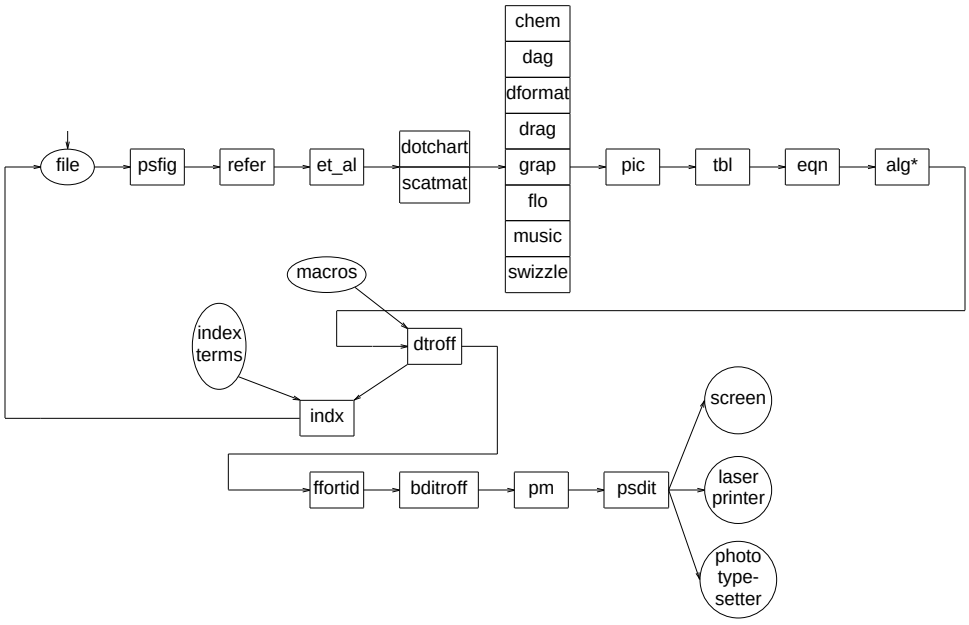


Figure 7. Flow of ditroff *system*

5 EXISTING SOLUTIONS

A number of word-processing and computing systems have been built for Arabic and related languages. The work that has influenced ours is described here.

1. An experimental bilingual, Arabic and English, system called IAS (Integrated Arabic System) was built on the IBM PC by a group led by Murat Tayli at King Saud University [38, 39, 6]. The IAS system was built around the kernel of the IAW (Intelligent Arabic Workstation) and its operating system with the addition of software tools that assist in writing new applications.
2. A WYSIWYG system for processing Arabic, Hebrew, English, and a host of other languages was built by Becker [40, 8] to run on the Xerox desktop publishing system. The system identifies the type of each character as it is being printed and chooses its printing direction on that basis. In other words, the system knows from the beginning that Arabic and Hebrew are printed from right to left and English is printed from left to right. There is the choice of two document directions, from left to right and from right to left. The document direction is that of the language in which most of the document is written or the language designated as the main language of the document. The screen appearance is calculated on the basis of the directions of the characters displayed and the current document direction.
3. The TARIF system [41] was developed at the University of Montpellier, running on an MC68000 microprocessor with a high-resolution graphic screen and a laser

printer, and was written in Pascal on an MS-DOS system. The system has two main parts. The first part is the editor which has the job of accepting Arabic textual input and showing it on the screen with each letter's form changed to match its position in its word. The second part is a formatter whose job is to arrange the text with keshide. The stretching is not in the form of longer connections between letters, but rather by use of long-form letters. This is accomplished by breaking each letter into three parts, the right, the left, and the middle. The short form of the letter is made by concatenating the right and the left part. The long form of the letter is made by inserting one or more middle parts in between the right and the left parts. This requires careful design of the pieces of the letters and works well only with fonts whose letters have perfectly flat horizontal parts. This would not work well with fonts whose characters have curved horizontal parts, such as that illustrated in Figures 4 and 5 above. It is not known from the available documentation if the system is multilingual with bidirectional processing and whether it can handle scientific text.

4. The IBM Scientific Center in Kuwait has developed a bilingual, Arabic and English, word-processing system [42. The problems are handled in two parts. The first part was the generation of an Arabic font in the Naskh style in three different formats, an outline font and two bitmaps for different resolutions. The second part is a bilingual text processor that includes an Arabic language editor that sets the lines with keshide and takes care of placing vowels correctly. The system is not for preparing scientific text.
5. Y. Haralambous [43] has developed an Arabic TEX system composed of the standard unidirectional TEX and a preprocessor called yarbtex. yarbtex transliterates phonetic Arabic text into the input format required by TEX augmented by J. Goldberg's bidirectional style [44] that reverses the printing of text surrounded by pairs of special symbols. Haralambous uses a Naskh font which is in a format acceptable to TEX and its dvi postprocessors. This system is multilingual and bidirectional to the extent that TEX and Goldberg's system are. However, there is no provision for keshide.
6. The most ambitious TEX-based project aimed at formatting Arabic is the TEX/XET program developed at the initiation of P. MacKay by modifying TEX itself to be bidirectional [37]. The program does the same reversing of text in designated right-to-left fonts that ffortid does, but inside the modified TEX using the internal data structures of the program rather than the dvi output. It assumes separate letters as TEX does and does nothing about stretching. TEX/XET was designed to be language-independent, and is in fact being heavily used for Hebrew processing in universities in Israel. It has become a strong competitor for ditroff/ffortid in this respect. Clearly, TEX/XET can handle all the scientific text that TEX can. It was always intended by MacKay that TEX/XET be used for Arabic and it in fact appears to be the version of TEX upon which Haralambous's latest system is based.
7. Haralambous upgraded yarbtex into a full-scale multilingual formatter called SCHOLAR TEX that comes with a very complete set of fonts. Besides being able to format Arabic, it can format Persian, Ottoman Turkish, Pashto, Urdu, Malay, classical Hebrew, modern Hebrew, Yiddish, Syriac, and others, both left-to-right and right-to-left [45]. It appears to be based on TEX/XET and is thus fully bidirectional.

TEX/XET and the current ditroff/ffortid are only partial solutions to the requirements for Arabic word-processing laid down earlier. Neither system is able to handle all of the requirements that stem from Arabic's connecting, changing, and stretchable letters.

6 DITROFF/FFORTID

The current version of ffortid, which has been used for years for formatting Hebrew, is quite a simple program. ffortid is stuck into the ditroff pipe between ditroff and whatever device driver is being used. ditroff has already formatted all the input on the assumption that all input is in left-to-right languages. The input has been broken into lines and pages according to the commands embedded in the input. Generally the lines are both left- and right-justified with words spread farther apart on a line-by-line basis to achieve the double justification. Whether or not the output is justified is controlled by user-issued commands.

The main job of ffortid is to take as input the output of ditroff and rebuild an output in the same format. The output is to be a formatted document in which all text in designated right-to-left fonts is printed in what appears to be from right to left. The right-to-left font positions are indicated by command-line options on the command invoking ffortid. At any time, there are two independent state variables that govern the rebuilding process of ffortid. One is the *current document direction*, which is either LR (left-to-right) or RL (right-to-left). It is settable at any time via the "x X PL" and "x X PR" commands, respectively, in the ffortid input. There are ditroff macros, ".PL" and ".PR", respectively, that cause the right commands to be left in the ditroff output, which is the ffortid input. Initially the current document direction is LR. The other variable is the *current font direction*, i.e., the writing direction of the current font. It is RL if the current font of the text is one of the fonts that has been designated right-to-left to the current run of ffortid and is LR otherwise.

The heart of the ffortid algorithm is a *layout algorithm* that operates on a line-by-line basis.

for each line in the file **do**
    if the current document direction is LR **then**
        reverse each contiguous sequence of RL characters in the line
    else (the current document direction is RL)
        reverse the whole line;
        reverse each contiguous sequence of LR characters in the line
    fi
od

An *RL (LR) character* is a character in any RL (LR) font. This algorithm is also the basis of processing right-to-left text in Becker's multilingual Xerox desktop publishing system, in Knuth and MacKay's TEX/XET, in Habusha's vi.iv 46], and in Allon's MINIX.XINIM [47]. The algorithm is now accepted as *the* way to handle horizontal bidirectional text in software originally designed for strictly unidirectional processing and in new software designed for horizontal bidirectional processing on the assumption that all text is stored in *time order*, i.e., the letters are stored in the order they are heard when the text is read aloud.

Note that this algorithm preserves line breaks and the nature of indentation and justification on each line, relative to the current document direction. That is, if the current document direction is LR, then indentation and justification is exactly as in the original, and if the current document direction is RL, then the indentation is on the opposite side and justification is flipped, e.g., if the original is right-justified, then the result is left-justified. For the purposes of this algorithm, a space is regarded as a character and its font, and thus direction, must be identifiable. In addition, formulae, tables, and pictures are considered LR subdocuments that may contain RL text internally. That is, even if a table contains Hebrew text, the table skeleton itself is an LR unit.

7 STILL TO BE SOLVED

By using ditroff/ffortid as the basis for the solution, many aspects of the requirements are already satisfied. Specifically,

1. There is horizontal bidirectional formatting and proper treatment of paragraphs, pages, and documents. It is possible to turn off hyphenation over any portion of the formatted text.
2. The system is user-friendly, at least insofar as ditroff and its pre- and postprocessors, and macro packages are considered user-friendly.
3. The system does produce book-quality output when used with a printer of sufficient resolution.
4. The system permits formatting of scientific and technical texts and all of the usual document entities that are found in them, including formulae, tables, diagrams, graphs, bibliographical citations, etc.

Yet to be solved are those requirements related specifically to Arabic and Persian formatting, including

1. connecting letters,
2. different forms for each letter,
3. position identification,
4. ligature identification,
5. vertical placement of diacritical marks, and
6. keshide.

It was decided to handle these as follows.

1. An Arabic font would provide the different forms of each letter as independent characters and each character that is to be connected on any side would be designed to be flush to the bounding box on that side at precisely the same place relative to the baseline. lines (a) and (b) of Figure 5 show how letters in such fonts connect.
2. A preprocessor, called atrn, would do letter form and ligature identification on letter-only input to yield output with each glyph to be printed, be it a form of a letter or a form of a ligature. The letter-only input would be according to a standard encoding for the language being processed, and the output would be according to the font's encoding for the glyphs. Thus, ditroff would format input consisting of the glyphs to be printed. If the input to the preprocessor has diacritical marks, then

they will be translated into their glyph codes surrounded by instructions to place them in the proper vertical position with respect to the character with which it is associated.

3. The ffortid postprocessor would be modified to stretch connections to last letters of words and/or lines in order to achieve one kind of keshide.

8 SOLUTION

As mentioned, the solution consists of creating a new program, atrn, and modifying an existing program, ffortid.

8.1 The atrn transliterator

The new program, atrn, is a ditroff preprocessor. Its main function is a mapping from pure spelling into a string of properly vertically and horizontally placed glyph codes, each one representing a letter or ligature, positioned within its word, or a diacritical mark. The pure spelling input is either in the standard encoding of the language or in some Latin, possibly phonetic, rendition of the same.

For Arabic, the input would be a string of letters in the ASMO code minus the lam-alif, plus codes for the vowels that are distinguishable from the codes for the letters. Since the ASMO code has only one code for each letter as opposed to up to four for each, it is clear that ASMO is intended to support automatic position identification and assignment. Because it does have a lam-alif, it does allow a user to force the use of a lam-alif. However, we insist on fully automated ligature identification based on user-selected options and on giving the user a way to prevent the ligature from being formed in any particular case. ASMO does have codes for some vowels but not for all, so we have to add codes, in the form of ditroff two-character special characters, for the other vowels. For uniformity, such codes are introduced for all the vowels, even the ones that happen to be represented in the ASMO code. Thus, for Arabic input in the extended ASMO code, atrn does position identification, ligature identification, and diacritical placement.

For each language supported by atrn, the mapping translates its standard encoding to glyph codes according to the fonts being used. Of course, this means that all the fonts for each language should use the same glyph encoding. Sometimes assuring this uniformity requires changing the `Encoding` vector of POSTSCRIPT fonts from different font foundries. As specified, atrn should accept input from standard input devices for the language. However, such devices are not always available, and no Arabic, Persian, and Urdu keyboards were available to the authors at any place that they worked. Therefore, it is convenient for atrn to also provide for translation from Latin keyboard input based on some phonetic or other mapping from Latin letters to the standard code for the language. This feature permits input of the pure spelling and vowels phonetically using the universally available Latin keyboard.

The flow of atrn is shown in Figure 8. The section below explains the order of the translations, in particular why ligature identification must come first. Each translation in the atrn flow is table-driven to allow the actual codes used to be changed easily.

Each language and each translation step is considered in more detail.

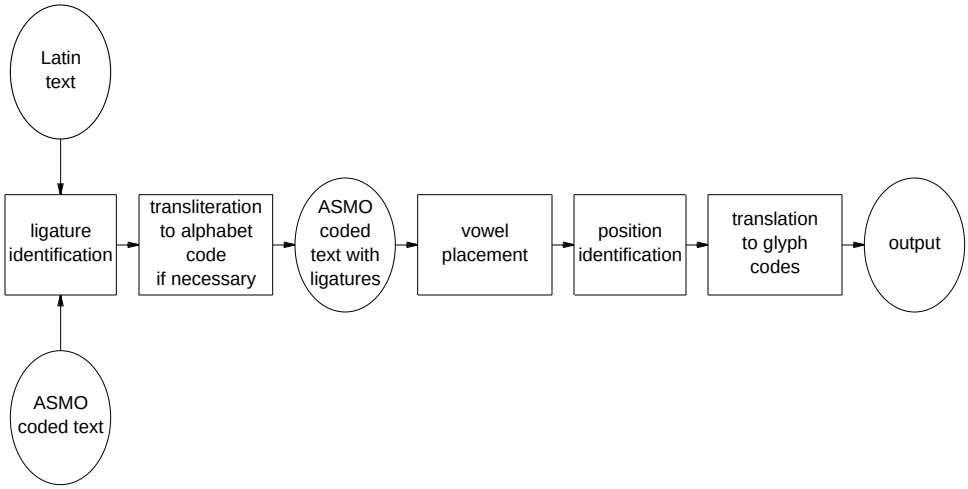


Figure 8. Flow of atrn

8.1.1 Input to the transliterator

The transliterator is structured to be a general transliterator for all kinds of phonetic input for languages in the Arabic-Persian family. If a table defining the transliteration can be built, then it can be integrated into the transliterator, which selects which table it uses as a function of the argument to the `%S` command described below. At present, however, only Arabic translation is supported.

The purpose of the transliteration phase is to allow someone who does not have an ASMO code generating Arabic terminal to prepare input to be formatted. Therefore, the code used for the alphabet to which the phonetic input is transliterated is ASMO. Therefore, a user with an ASMO-generating keyboard needs to skip only this phase; the other phases that determine positions, ligatures, and vowel placements cannot be skipped. Thus, one of the options to atrn is not to translate its input at all. This scheme can be used to provide any pre-formatting processing to any input language regardless of the input keyboard.

The word كتب for example, is represented by the phonetic input `ktb`. Below is the phonetic input of the first paragraph of the Arabic abstract of this paper. `%Sar` marks the beginning of phonetic Arabic text, and `%Ear` marks its end. The text is shown with all of the embedded ditroff commands: `.OA` means “other abstract header”, `.lp` means “left adjusted paragraph”, `.PP` means “indented paragraph”, `\*(AN` means “switch to AN font and change size as necessary”, `\*P` means “switch to previous font and change size as necessary”, `\*H` means “switch to Helvetica font and change size as necessary”, and `\*R` means “switch to Times Roman font and change size as necessary”.

```
.OA %Sar\*(AN\s+2mqdmt‘\s-2%Ear
.lp
%SarHZa almqal ySf brnamj ltwDyb allR‘t‘ alerbyt‘ walZy ymkn mn
twDyb nSwS elmyt‘ mteddt‘ allR‘at, mhtwyt‘ elA‘ nS balerbyt‘
walfarsyt‘ balaDaft‘ llR‘at axrA‘, rswmat, rswmat byanyt‘,
jdawl, mSadr byblywR‘rafyt‘, wbyblywR‘rafya. Albrnamj Hw thsyn
lI-%Ear\*Hditroff/ffortid\*(AN%Sar alqadr ala~n elA‘ mealjt‘
alebryt‘ fy wcaY‘q mteddt‘ allR‘at.
%Ear\*Hditroff/ffortid\*(AN%Sar ebart‘ en qbl mealj
%Ear\*R(preprocessor)\*(AN%Sar wbed mealj
%Ear\*R(postprocessor)\*(AN%Sar lbrnamj alSf fy
%Ear\*HUNIX\*(AN%Sar, %Ear\*Hditroff \*(AN%Sar(%Ear\*RDevice
Independent Typesetter RunOFF\*(AN%Sar). albrnamj aljdyd mbny
mn dwn idxal ay tR‘yyr elA‘ %Ear\*Hditroff\*(AN%Sar alqaY‘m.
aliDaft‘ mkwnt‘ elA‘ alaR‘lb mn qbl mealj jdyd, Tqm kaml mn
alhrwf almTbeyt‘, wnsxt‘ mhsnt‘ mn bed mealj qaY‘m.
.PP
```

The Latin letter chosen to represent any Arabic letter is one whose pronunciation reminds the user of a pronunciation of the Arabic letter. It would be best to have a unique one-for-one mapping. However, this is impossible. From the point of view of accurately representing what the user must choose, there are 42 letters and 17 diacritical marks (vowels) to be represented by 52 characters (upper and lower case). Moreover, there are, in some instances, more than two Arabic letters that can be feasibly represented by the two cases of one Latin letter. As a consequence, it is sometimes necessary to map two ASCII characters to one Arabic letter. When more than one Arabic letter is feasibly represented by one Latin letter, the lower-case letter goes to the most frequent Arabic letter, the upper-case letter goes to the next most frequent, and the two-letter codes go to the least frequent, etc. The idea is to minimize typing time. When a two-letter code is used, it is critical to make sure that the second letter be chosen so that it is not a valid representation of any letter in its own right, so as to insure unambiguous recognition. In other words, since `h` is the code for a letter, it cannot be used as the second letter of another letter's code, e.g., `kh` for khaf, as is commonly used for phonetic renditions of Arabic for human consumption. The table of Appendix II shows the phonetic mapping implemented by `atrn` for Arabic and Persian letters.

8.1.2 Output of the transliterator

Our system prints the Arabic text on a laser printer with high-quality POSTSCRIPT outline fonts. The first font that we had available for use was the Naskh font produced by Draper and Parkins. It was necessary to make a few modifications and additions. The main modifications were to give new codes to the glyphs and to make the internal names of the characters more mnemonic than the standard Adobe names given to the codes. For example, `taa_SA` is more meaningful than `Adieresis`. The names given to the glyphs were the same used in the second table of the transliterator. While these names are in the last analysis merely internal to the programs, making this agreement helped the first author keep his sanity when debugging the software. It was also necessary to add seven new

characters to the font to give it the capability of printing the standard international punctuation that appears in nearly every standard coding of an alphabet, the ASMO code as well! The characters &, @, ^, {, }, |, and ~ were added by lifting outlines for them from a public domain Hebrew font whose other international characters looked most like the international characters that the Arabic font did have. The new codes were assigned so that the international standard characters kept the code that they have in nearly all standard code sequences. Then the glyphs for Arabic, in all forms of all letters, were assigned to the rest of the table so that within the section of glyphs for one position, all the glyphs are in alphabetical order. Thus, the stand-alone glyphs got codes 102–160 (octal), connect-after glyphs got codes 161–224, connect-both glyphs got codes 225–273, and connect-previous glyphs got codes 274–331. The ligatures got the codes 322–355, and the diacritical marks got the codes 356–376. In order that the output be acceptable as input to ditroff, the transliterator used the absolute index escape to name each glyph by its code. That is each glyph is addressed by the escape `\N'xxx'` where *xxx* is the decimal code of the glyph.

8.1.3 Steps of the transliteration

The transliteration accepts a mixture of phonetic or ASMO text together with other languages, English or Hebrew for example. The phonetic text can be in any language for which a translation table is defined. If the transliterator reads a phonetic letter, it transliterates the letter into one of the alphabets' letters, determines the position within the word, and on the basis of this position, translates the alphabetic letter into the ditroff escape sequence that causes printing of the correct form of the letter. Figure 9 shows the steps to transliterate the phonetic letter `t` in the phonetic word `ktb` (كتب). Figure 10 shows the steps to translate the ASMO code equal to the ASCII code for `g` in the ASMO (not phonetic) word `gPG` (هذا). The word written above an arrow gives the name of the procedure that implements the translation represented by the arrow, and the word written under an arrows gives the name of the main table that is involved in the translation.

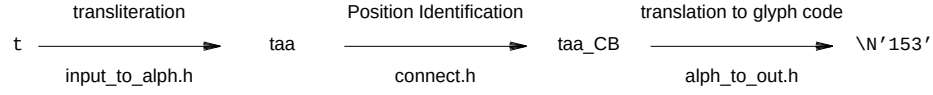


Figure 9. Steps of transliteration of phonetic `t`

If the transliterator reads text written in its phonetic language, it translates it according to the tables before outputting to the standard output. Otherwise, it just copies the input to the standard output. For this reason a method is needed to announce when to start and when to end the transliteration. The mechanism is defined according to the following laws:

1. In the beginning, the transliterator finds itself in the Latin environment and the transliterator does not work on any of the input.

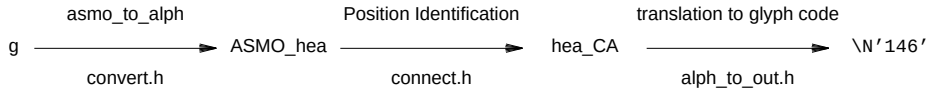


Figure 10. Steps of transliteration of ASMO g

2. The % character is defined as an escape character to announce to the transliterator that a command to start or end a transliteration is coming up. A different character can be selected as the escape character by using it with the **–e** command-line flag. The string %S*ll* indicates the start of transliteration according to the table for the language *ll* and %E*ll* ends that transliteration. In the present implementation of atrn, the possible values and the designated language for *ll* are as in the table below:

ar	Arabic
fr	Persian (Farsi)
ur	Urdu

While the program has hooks for all of the designated languages listed above, the only transliteration currently supported is for Arabic. Neither of the authors knows any of the other languages. The use of *ll* is compulsory only after the %S in order for the transliterator to know which language table to use. The *ll* is optional after %E. If it is not explicitly specified which language environment is ended by a %E, atrn assumes that it ends the most recently started but not ended environment. The scheme below indicates which transliteration is in effect for each region

```
%Sar ... Arabic ... %Sfr ... Persian ...
%E ... Arabic ... %E
```

3. It is thus possible to nest language environments. The transliterator enforces strict nesting and complains if environments are being ended in an order which is not the reverse of that in which they were started. Thus,

`%Sar ... %Ear ... %Sfr ... %Efr` is legal, while

`%Sar ... %Efr` is illegal.

4. Closure of the transliteration of one language's environment, *E*, by the use of an explicit *ll* argument for %E closes also the transliteration of all language environments nested inside *E*. Note the different environments in effect at the ends of the two examples.

```
English ... %Sar ... Arabic ... %Sfr ... Persian ...
%Sur ... Urdu ... %Ear ... English
```

```
English ... %Sar ... Arabic ... %Sfr ... Persian ...
%Sur ... Urdu ... %E ... Persian
```

5. It is forbidden to nest the same language. In the case

```
%Sar ... %Sar ... ,
```

   the transliterator ignores the second beginning.
6. When the transliterator enters the environment of a ditroff command, in a line that begins with a `.` or `'`, or into an escape sequence, which begins with `\` anywhere in the text, then the current global translation environment is interrupted while the translator moves to a local non-translating environment, which ends automatically when the command or escape sequence ends. This implies that the transliterator knows the syntax of ditroff commands and escape sequences. Thus, transliterations can be applied to arguments of commands and escape sequences that happen to be text.
7. Leaving the local environment causes the transliterator to revert to the global environment state that was in effect upon entry to the local environment.

```
This is an English global environment. %SarThis should
be Arabic Phonetic text. %SfrNow this should be Persian.
This is a global Persian environment nested within an
Arabic environment.
.tl 'local-English'%Sarlocal-Arabic'%Surlocal-Urdu'
Now we're back in a Persian environment. A labeled exit
from a command or escape closes all of the language
translation environments inside it.  %EarNow move back
to the English global environment. Note that closing the
Arabic global environment also closes all its internally
nested language environments.
```

8.1.4 Determining position of letters in words

As mentioned before, the form of a letter in the Arabic and related languages depends on its position within the containing word. The atrn preprocessor has the job of determining the position of each letter, because it is required that the form of the letter be known before submitting the text to the formatter. The formatter needs to know the width of each letter. The width of a letter, in turn, depends on the form of the letter, because the different forms of a letter are of different widths.

The beginning and end of *any* environment, even nested, are the beginning and end of words. The letter before the beginning of a nested *global* environment is the end of a word, and the letter after the end of a nested global environment is the beginning of a word. Presumably, no one will switch languages in the middle of a word. The hard question is what to do about text found in escape sequences and command arguments, both of which are considered *local* environments. Oftentimes, but not always, escape sequences and command arguments are applied to interior portions of a word. For example, to get the French word "élève", the input "`\o'e\(aa'l\o'e\(ga've`" can be given to ditroff. In addition, if one has a macro `.BB` for emboldening its argument and connecting directly to the next word, one way to get the letters "por" in "sub**por**tion" emboldened is to say

```
sub\c
.BB por
tion
```

Therefore, it was decided that position determination in a global environment is not interrupted by an embedded local environment. For example, if an escape sequence is in the middle of a global environment word, then the escape sequence does not end the word, and the first character after the escape sequence is also in the middle of the word. Note however, that the beginning and end of the text of an escape sequence or command argument are the beginning and end of a word embedded inside another word. This is, admittedly, a strange effect, but it can be avoided if so desired by use of the `I` character described below.

In general, the user of atrn must be careful about introducing excess blanks into the text that ends up delimiting words. However, the care needed for atrn is no more than that which must be exercised in using ditroff itself, in which extra spaces in the input will break words and lines and will cause the printing of ugly extra spaces on output.

Because of the possibility that words may contain arbitrarily long embedded escape sequences, position determination requires lookahead with a range large enough to get through any escape sequence. Before the form-finder determines the position of a letter *l* that precedes a \, it looks ahead to the next textual character and only then can it determine with certainty the position of *l*. The consecutive escape sequences that appear after the *l* remain in a buffer that is written to the output only after *l*-cum-position is written.

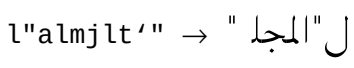
In spite of the fact that the transliterator determines letter forms automatically from the position of the letter in its word, the user has the possibility to intervene and force the algorithm to determine whatever position he or she desires. The capability is needed, for example in this document or in a grammar book, of exhibiting all the forms of the letters in a table of forms, in which each form actually stands alone. This capability is achieved by defining two dummy characters whose appearance in the text causes no output, but instead influences the position of its neighbors. The two dummy characters are `I` and `M`.

An appearance of the character `I` causes the previous letter to be connected-after to the following letter. This character is used when it is desired to print a solitary letter as one which is connected after. For example the phonetic input `b` causes printing of ب, the stand-alone baa, because there is nothing before it or after it. The phonetic input `bI` causes printing of بـ, the connecting-after baa. Figure 11 shows an additional use of the `I` to force the printing of a connecting-after lam instead of a stand-alone lam.

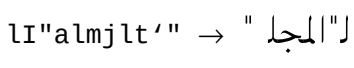
An appearance of the character `M` in the middle of a word causes splitting of the connection between the preceding and the following letter. This character is used when it is desired to force a letter to be in its stand-alone form even when it appears in the middle of a word. For example, the word بلبل is obtained by the phonetic input `blbl`, while the sequence of its letters, ب ل ب ل is obtained by the phonetic input `bMlMbMl`.

8.1.5 Ligatures

Ligature identification requires lookahead. Because the input may, and most likely will, be a pipe, the re-reading of a character after a lookahead cannot be implemented by



(a)



(b)

Figure 11. Use of `I` *to force correct output*

backing up in the input and re-reading the previous character. The consequence of this limitation is that there is a ligature buffer to hold characters read in for ligature determination. If the buffered characters turn out not to be a ligature, then it is arranged that the next characters are read from the buffer rather than from the normal input.

The transliterator checks each letter *l* it reads to see if it could be the first letter of a ligature pair, and if so, it looks at the next letter to see if it is the second character of a ligature that begins with the first character. If so, it looks at the next letter to see if is the next character of a ligature that has started already. It continues in this manner until it finds some character that cannot extend the ligature built so far. At this time, there must of necessity be only one choice left. If a ligature was, in fact, built up, then that ligature is taken as the next letter, in place of *l*. This ligature is subjected to form determination as is an ordinary letter. On the other hand, if the buffered characters do not form a ligature, then a flag is set to tell the reader that the next *n* characters are to be found in the ligature buffer, where *n* is the number of characters read while finding no ligature.

For example, if there are ligatures lam-alif and lam-alif-dal, then phonetically, they are `la` and `lad`. If the input so far is `la`, then it has not yet been recognized as a ligature. Assuming that these are the only ligatures beginning with lam-alif, then ligature recognition comes only with the third letter. If the third letter is a `d`, then the lam-alif-dal ligature is recognized, and the transliterator moves on to the input after the `d`. If the third letter is something else, then lam-alif has been recognized and the third letter is considered as a separate letter.

As stated before, ligatures in Arabic are optional except for one, the lam-alif, ﻻ, and the other variations of it, based on the different variations of alif, ﻷ, ﻹ, and ﻵ. The other ligatures can be ranked into levels such that those of Level *i* include those of Level $i+1$. Figure 12 shows the three levels of ligatures, in which Level 2 denotes the minimal set of lam-alif and its variations. Level 2 is the default and the user signifies the level of ligaturing in effect for a run of `atrn` in a command line option of the form **–l***level_no*. Level 2 ligatures are mandatory and Level 1 and Level 0 ligatures are optional.

The set of ligatures available at any level is captured in a table that is compiled into `ffortid`. Clearly, all fonts should have the mandatory ligatures. The elements of the optional levels depend on the ligatures that are supplied in the available fonts. As a new font with new ligatures is made available, the table must be modified and `ffortid` recompiled. This is not a serious difficulty if the sources are available. Of course, there is the

Figure 12. Levels of ligaturing

annoying problem of a ligature that is in the table but is not available in the current font; so far this must be avoided by the user turning off the problematic level of ligaturing or using the `I` input character between the letters that might otherwise be formed into a ligature. In retrospect, a better design would be to specify the ligatures available with each font in the font's ditroff width table, as is done for the standard "f" ligatures for Latin fonts. This solution was avoided because the strict format of the binary ditroff width tables does not permit the desired specifications. The new version of ditroff, which uses only ASCII width tables, has provisions for fields to be ignored by ditroff, specifically to allow placement of font-relevant information for use by pre- and postprocessors.

The dependence of the ligatures on the position of the component letters obliges atrn to check, after the step of contextual analysis, if the identified ligature has a form for its position within the word. If not, atrn must undo the ligature identification and output the individual letters. Lam-alif has a form for all positions, but lam-mim does not. In particular it does not have stand-alone and connected-after forms. Thus, if the lam followed by mim occur in a position in which the mim is not to connect after, while the lam is not to connect before, thus making the combination stand-alone, or in which the lam is not to connect before but the mim is required to connect after, thus making the combination connecting-after, then the lam-mim sequence must be left as two separate letters.

8.1.6 Vertical placement of vowels

As mentioned, in Arabic and related languages, vowels are optional diacritical marks. This is also the case in Hebrew. A diacritical mark is a sign that appears above or below a letter and specifies only the vowel sound following the sound of the letter, which is thus a consonant. The following issues are relevant to the treatment of diacritical marks.

1. The presence of diacritical marks does not affect the determination of the positions of the letters in words, and therefore, atrn ignores diacritical marks during position identification. This is indicative of the probable reason that the diacritical marks have grown to be optional. The form of the letters comes from the natural continuous flow of the hand. Diacritical marks either interrupt that flow or have to be added after the fact, making them a nuisance. Since the meaning of the word is carried almost entirely by the root consonants and prefix and suffix letters, vowels are not necessary to understand the text. Once a word is understood, its pronunciation is known to all native speakers. Thus, the nuisance becomes an option with avoidance favored.

2. Thus, the transliterator must provide the option of not using diacritical marks at all.
3. The vertical placement of each diacritical mark depends on the height or depth of the letter that it is placed above or below. For example, an above-placed diacritical should be place higher over the letter ا than over the letter ب, and a below-placed diacritical should be placed lower below the letter ي than below the letter ب. For this purpose there is a table compiled into ffortid, mapping each letter in the Arabic-Persian alphabet to a vertical distance above and a vertical distance below the letter for placement of above-the-letter and below-the-letter diacriticals. As with the ligature table, a better design would be for this table to be part of the font width tables, for the heights and depths of letters do vary with fonts. Figure 13 shows three lines with identical letters, the first with no diacritical marks, the second with diacritical marks as placed by the font in which the above-letter marks clear the tallest letter and the below-letter marks clear the deepest letter, and the third with diacritical marks adjusted according to the table. The third clearly looks better than

الجنۀ تحت قدام ا مهات
(a)
الْجَنَّۀُ تَحْتَ قَدَامِ ا ُمهَاتِ
(b)
الْجَنَّۀُ تَحْتَ قَدَامِ ا ُمهَاتِ
(c)

Figure 13. Different forms of voweling

the second. The inputs for the first and third of these lines are

```
aljnt‘ tht Aqdam alAmHat
```

and

```
a’lOj’n~~’t‘u t’hOt’ A’qOd’amE alAum~~’H’atE
```

respectively. The second line could not be forced out of atrn, because it tries to do what it did with the third line. Therefore, it was obtained by inputting the glyph codes directly with no added vertical movements.

There is a command-line option, **-nv**, to turn off vertical adjustment of diacritical marks, but then it is off for the whole of a run of atrn. If this paper had been run with this option, then the input for the third line would cause printing of the second line.

In addition to a phonetic representation given to each vowel, there is also a ditroff two-ASCII-character code given to each vowel. This representation allows the vowels to be accessed directly from ditroff; this way, the diacriticals can be used independently of phonetic translation with exactly the same difficulty that accent marks are used in Latin

text in ditroff. This degree of difficulty is acceptable for something that is optional. In order not to overload an already full table of two-character codes, for each diacritical, we were able to find an existing code in use on our installation of ditroff that is mnemonic of the name of the diacritical. Thus, the fatha, "ˊ" is known both by its visual equivalent "**'**" and by the quite acceptably mnemonic "`\(ft`".

8.1.7 Other features of atrn

The end-of-word indicators recognized by atrn are:

space newline tab **, - _ . ; ? ! :) ( ] [} { > < + & | "**
‘ (hamza) **0 1 2 3 4 5 6 7 8 9** (any digit) **M** (the "M" character)

The translation capabilities of atrn are generally used for the non-English portions in a document. In most cases, then, when starting a portion of text to be translated by atrn, it is necessary to turn ditroff's ligature and hyphenation mechanisms off. Similarly, it may be desired to turn them back on at the end of these portions of text. Therefore, as a convenience for the user, if the –**g** or –**h** arguments are specified, atrn automatically turns the appropriate ditroff mechanisms off and on when it encounters the beginning and end, respectively, of a translation region. Specifically, if present, the optional argument –**g**[*`ligature-on-argument`*] causes atrn to issue

```
.lg 0
```

at the beginning of the output for each translation region in order to turn off Latin ligaturing (e.g., `ffi` → ffi) and

```
.lg x
```

at the end of each such region to turn Latin ligaturing back on. If the optional *`ligature-on-argument`* is present, it is used as *x*; otherwise *x* is `1`. In addition, if present, the optional argument –**h**[*`hyphenation-on-argument`*] causes atrn to issue

```
.hy 0
```

to turn on English hyphenation at the beginning of any translated output and

```
.hy x
```

to turn English hyphenation back on at the end of any translated output. If the optional *`hyphenation-on-argument`* is present, it is used as *x*; otherwise *x* is `1`.

8.2 The extended ffortid

When ffortid is placed in the pipe between ditroff and a device driver, the result is a bidirectional version of ditroff in which all text in fonts designated as right-to-left is

printed from right to left. By use of two macros .PR and .PL, the document direction can be specified as predominantly right-to-left or predominantly left-to-right. The effect of these is to define on which side of the page is a line considered to begin, and thus, from where indentation and other line-dependent transformations take place. All other ditroff commands continue to work, relative to the newly defined line beginning.

ffortid accepts input from ditroff and reorders the contents of each line so that all the text on the line is printed in its correct direction. Its output is identical in form to that of ditroff so that any ditroff postprocessor can receive the ffortid output and be none the wiser about the true source of its input. See Figure 14 for a flow schematic.

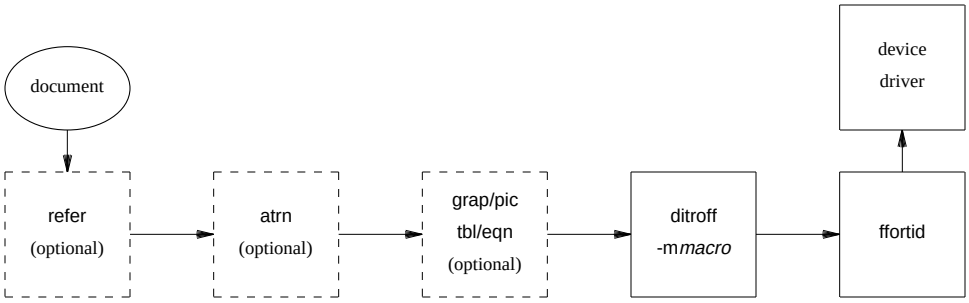


Figure 14. ffortid flow diagram

It is important to remember that the job of dividing the text into lines and pages is done by ditroff and therefore, ffortid does not have to know at all about ditroff's preprocessors. The reader should recall the basic algorithm used by ffortid which is described in Section 6.

The original ditroff/ffortid is not powerful enough to handle Arabic, Persian, and Urdu text for two main reason.

1. It does not take care of changing the form of letters based on their positions within their words.
2. It is not capable of stretching letters to justify the lines to the end, on the right.

The first problem is solved by the atrn preprocessor even before ditroff sees the input. The second problem was a tough nut to crack. After all, ditroff itself does not stretch any letters. It can be told either to adjust lines by inserting more white space between the words, or to leave the lines unadjusted to create a torn flag effect!

We thought of the solution when we examined the code for ffortid. In the ditroff output, an output line is represented generally by a list of (character, movement) pairs, e.g.,

$$c_1\ m_1\ c_2\ m_2\ c_3\ m_3\ \cdots\ m_{n-2}\ c_{n-1}\ m_{n-1}\ c_n$$

in which each movement is the distance to the beginning of the next character. Recall that ffortid's job is to reverse the order of characters that are in right-to-left fonts. Assuming that in the output line above, all characters are in right-to-left fonts, one might think

that it would suffice to simply flip the line to get

$$c_n\ m_{n-1}\ c_{n-1}\ m_{n-2}\ \cdots\ m_3\ c_3\ m_2\ c_2\ m_1\ c_1,$$

but then the movements would be applied to the wrong characters. The simplest way to generate the correct movements is for ffortid to reformat the line itself using code duplicated from ditroff. It reads the c_is and notes the end-of-word markers to figure out what text is in the line, then it fills the line in, with a guarantee that the length of the permuted text of the line can be no longer than the original line length. Then if the original line was adjusted, the excess space after the last word is divided by the number of inter-word gaps, with a bit more for sentence boundaries, and then each inter-word gap gets its share of the extra white space. Granted that this is repeated computation, but it is better to do it in a postprocessor, which is so fast compared to ditroff that ditroff remains the bottleneck, and to leave ditroff unchanged.

Once it was clear what ffortid is really doing, the solution to the stretching problem jumped at us. Let ditroff format the Arabic text with hyphenation turned off and filling and adjusting turned on, in order to determine what can fit on each line. Then let ffortid do what it has been doing, *except* that it now takes all of the excess at the end of the line and uses that as the length of the filler inserted into the connection to the last connecting-before letter in the line or as the total length of all the fillers inserted if more than one is to be inserted.

Yes, this solution in essence lets ditroff do some more work than is needed, throws the result out and does the work again in a different way. The solution does make use of important information generated by the work of ditroff, namely the words that can fit on the line and whether the original line was adjusted. This last piece is important, because it is not desired to stretch out the last word of a line, such as the last line of a paragraph, that was ended before filling up the line and therefore, was not adjusted.

8.2.1 Solution to the stretching problem

The solution consists in a number of simple extensions to ffortid. The problem is divided into three parts, implementation of the stretching itself, calculation of the total amount of stretching needed to adjust a line, and distribution of the stretching among the words in the line.

8.2.2 Implementation of stretching itself

The method of stretching in the current new version of ffortid, as mentioned before, is by lengthening the connection to a character. Thus, only connecting-before characters are considered stretchable. For example, the character ـ is stretchable, but ا is not. A word is said to be *stretchable* if has a stretchable character; if it is stretched, then its last stretchable character is stretched. Thus, a word containing no stretchable character is considered not stretchable.

Once the amount of stretch that is needed is known, then the connection is lengthened by putting in enough fillers to cover that length. The filler is given the two-character code of the hyphen, because there is no real hyphen in Arabic, and the function of the stretch

is to avoid hyphens. Thus, the ditroff escape sequence for making a filler of length *l* is `\l'`*l*`\(hy'`. As mentioned before, the filler is at the same baseline as the connection to and from the letters, is the same thickness as these connections, and is flush to the left and right boundaries of its bounding box. Thus, a sequence of fillers looks like a solid line at the baseline of Arabic letters.

8.2.3 Calculation of the amount of stretch

The amount of stretch for a line is equal to the sum of the lengths of the spaces that ditroff inserted between the words in order to justify the line. This value must include only the space between the words beyond the minimum obtained if the line were not adjusted. This sum has to be extracted from the ditroff output. There are at least two ways of doing this calculation.

1. Compute anew the width of the line, ignoring the extra space that ditroff inserted between the words. The difference between the new line length and the original line length is the amount of stretching needed. This solution requires knowledge of the length of the line, which needs to be computed by summing up the widths of the characters and adding the sum of the movements.
2. Take the sum of the lengths of the inter-word gaps and subtract from it the sum of the length of the same number of spaces. This difference is the total amount of stretching needed! The minimum spacing between the words is the space, and if a line has not been adjusted, its inter-word gap is precisely the size of the space.

The simplest solution is the second, and it is used.

8.2.4 Styles of stretching

In the enhanced ffortid, four styles of stretching are supported when an Arabic-Persian family language is used; that is, there is no stretching for languages outside this family.

1. The default option is no stretching at all. The original ffortid behavior is adopted. This option can be selected explicitly by the **–sn** command-line flag for ffortid.
2. The last stretchable word in the line is stretched by the excess amount calculated. If no word in the line is stretchable, then leave the words spread. This option is selected by the **–sf** flag.
3. The last stretchable word in the line is stretched by the excess amount calculated, *up to a maximum* length equal to the current point size times the length of the connection filler. The left-over excess is given to the previous stretchable word in the line, up to the same maximum, etc. If no word in the line is stretchable, then leave the words spread. This option is selected by the **–sl** flag.
4. Stretch all stretchable words by their share of the excess calculated. If no word in the line is stretchable, then leave the words spread. This option is selected by the **–sa** flag.

Of course, any other style of stretching can be programmed by the user by inserting the `\l'`*l*`\(hy'` construction wherever needed. In this manner, stretching, a capability that ditroff does not offer, is achieved without changing ditroff itself!

The paragraphs below show the results of four different stretching options on the same input. The stretched outputs look significantly better than the unstretched, spread output, even to the non-Arabic eyes of the second author. A narrow column width is used to accentuate the spreading and stretching effects and their differences. The presence of English and Hebrew text is to show the effect of non-Arabic text on the stretching.

Extra spaces are distributed between words.

هذا مثال لطباء وتصفيف اللغ
العربيه سويه مع لغات خرى
كا نگليزيه (English) والعبريه (עברית) .
مثا المعطا توضح الفرق بين كل
واحد من ساليب التصفيف ومد
الكلمات .

Connections to last connecting-before letters in lines are stretched.

هذا مثال لطباء وتصفيف اللغـــــــ
العربيه سويه مع لغات خـــــــــــرى
كا نگليزيه (English) والعبريـــ (עברית) .
مثا المعطا توضح الفرق بين كـــل
واحد من ساليب التصفيف ومـــــــد
الكلمات .

Connections to last connecting-before letters in lines are stretched to maximum amount, with remainder going to preceding words.

هذا مثال لطباء وتصفيف اللغـــــــ
العربيه سويه مع لغـــــات خـــــــرى
كا نگليزيه (English) والعبريـــ (עברית) .
مثا المعطا توضح الفرق بين كـــل
واحد من ساليب التصفيف ومـــــــد
الكلمات .

Connections to last connecting-before letters in all words in lines are stretched.

هـذا مثـال لطباعـ وتصفيـف اللغـ
العربيـ سويـ مـع لغـات خـرى
كا نگليزيـ (English) والعبريـ (עברית) .
مثل المعطا توضح الفرق بين كل
واحـد مـن ساليـب التصفيـف ومـد
الكلمات .

For the future, after we have developed dynamic Arabic fonts with actual stretchable letters, it will be necessary to introduce more options to ffortid, among which are

1. using only stretchable letters
2. using only stretchable connections, and
3. using both

in all of the variations as to where in the line to stretch.

8.2.5 ffortid *command-line options*

Those features of ffortid that have not had a natural mentioning in the above discussion are summarized here by describing their command-line options.

In the command line, the –**r***font-position-list* argument is used to specify which font positions are to be considered right-to-left. ffortid, like ditroff, recognizes up to 256 possible font positions (0–255). The actual number of available font positions depends only on the typesetting device and its associated ditroff device driver. The default font direction for all possible font positions is left-to-right. Once a font's direction is set, it remains in effect throughout the entire document. Observe then that ffortid 's processing is independent of what glyphs actually get printed for the mounted fonts. It processes the designated fonts as right-to-left fonts even if, in fact, the alphabet is that of a left-to-right language. In fact, it is possible that the same font be mounted in two different positions, only one of which is designated as a right-to-left font position. This is how a single font can be printed left-to-right and right-to-left in the same document. This is also how it is recommended to obtain left-to-right (in order of decreasing digit significance) printing of Arabic numerals without having to input the digits backwards.

The –**a***font-position-list* argument is used to indicate which font positions, generally a subset of those designated as right-to-left, contain fonts for Arabic, Persian, or related languages. For these fonts, left and right justification of a line is achieved by stretching instead of inserting extra white space between the words in the line. Stretching is done on a line only if the line contains at least one word in a –**a** designated font. If so, stretching is used in place of extra white space insertion for the entire line. There are several kinds of stretching, and which is in effect for all –**a** designated fonts is specified

with the –**s** option, described in Section 8.2.4. If it is desired not to stretch a particular Arabic, Persian, or other font, while still stretching others, then the particular font should not be listed in the –**a***font-position-list*. Words in such fonts will not be stretched and will be spread with extra white space if the containing line is spread with extra white space.

The –**r** and the –**a** specifications are independent. If a font is in the –**a***font-position-list* but not in the –**r***font-position-list*, then its text will be stretched but not reversed. This independence can be used to advantage when it is necessary to designate a particular Arabic, Persian, or other font as left-to-right, for examples, or to get around the abovementioned limitations in the use of *eqn*, *ideal*, *pic*, or *tbl*.

Owing to the difficulties mentioned in Section 1 in typesetting the stretching examples of Section 8.2.4, it is now clear that it should be possible to specify a different stretching style for each mounted Arabic font. Then, all of the examples could have been done as part of the single document simply by mounting the one Arabic font in four different positions, each with a different stretching style specified.

9 RESULTS

Appendix I shows some examples of the use of ditroff preprocessors together with the new software. The first of these uses eqn to give a more scientific interpretation of what was said when light was created in the midst of an Arabic translation of the relevant sentences of Genesis. The second of these uses chem to show the structure of compounds found in petroleum as they might be illustrated in a chemistry class in an Arabic-speaking petroleum-exporting country.

The appendix also shows the famous story, “The Rabbit and the Elephant” typeset by the system described herein. This output should be compared to that from yarbtex [43]. The two outputs use similar fonts, but the latter does not exhibit any keshide.

10 CONCLUSIONS

It appears that software described herein has met its goals. In particular, when we show the output of the software to native Arabic-speaking scholars of Arabic here, they seem genuinely appreciative of the output.

Recall that the two main jobs of the new ffortid are to reverse right-to-left text line-by-line, according to the algorithm described in Section 6 and to stretch one or more words in these lines. The reversing algorithm requires the ability to determine the ends of formatted lines, and stretching requires the ability to determine the ends of words in formatted lines. Therefore it must be possible to find ends-of-words and ends-of-lines in the input ffortid, which is the output of ditroff.

ditroff output consists of a preamble describing the device, followed by a sequence of page descriptions. The description of a page consists logically of a sequence of (position, character) pairs, each describing exactly where on the page to print a character. The actual form of the position information is as occasional absolute coordinates with intervening horizontal and vertical movements. Thus a program, usually a device driver, reading this output must keep a position state and follow the relative movements in order to calculate the exact position of each character. Embedded among these (position, character) pairs, and actually independent of them, are end-of-line markers, of the form n*b a*

(the important thing here is the n; the *b* and the *a* give the amount of space before and after the line in the device's units) and end-of-word markers of the form w. The ditroff output of the line

```
This is an example of a line.
```

is

```
H576
V96
cT
49h40i22sw51i22sw51a36nw60e36x40a36m62p40L22ew56o40fw47aw
56L22i22n40e36.n96 0
```

Note the bold-faced end-of-word and end-of-line markers. Note that no w commands are issued before hyphens generated by the formatter; they come only at the ends of input words. Device drivers generally ignore the semantic markers, but the semantic markers permit other analyses, such as that necessary to do reversing and stretching.

These markers are necessary and cannot be deduced from the movements. Not all large movements to the left with small movements downward are ends of lines. One finds such movements in tables, pictures, graphs, etc. Not all movements the size of a space or a bit more are ends of words. They may be movements within equations, tables, pictures, etc.

The lack of end-of-line and end-of-word markers in TEX's output, in dvi [48] format, prevents production of a bidirectional, stretching version of TEX using the simple scheme of reorganizing the dvi output on a line-by-line basis. The only way to add reversing and stretching is to modify TEX itself either to do the reversing and stretching internally or to put more information in the dvi form output. The latter is probably worse, because then none of the existing independently developed device drivers would accept the new output. Therefore, MacKay and Knuth opted for the former in making the bidirectional TEX/XETEX. For either approach, one cannot use the standard distributed TEX, and one faces the problem of maintaining more than one version of TEX. This maintenance problem is immediate, because TEX/XETEX does not do stretching and would have to be modified in order to do it.

The modularity of the ditroff system and the end-of-word and end-of-line markers in the standard ditroff output made developing this software quite straightforward, in that we could focus directly on the problems of Arabic-Persian formatting without having to concern ourselves with other parts of the general formatting problem. However, once the new software was available it was possible to use it in conjunction with all of the rest of the ditroff system with very little bother.

The solutions developed for this software are now available to be incorporated into other, less modular, systems.

The next step for the future is to complete the dynamic fonts with stretchable letters and to develop a ditroff-to-device-driver interface for letters whose widths vary from that given in the standard width tables. This will be done, as always, without modifying ditroff or its output language.

ACKNOWLEDGEMENTS

The authors thank Gregory Abowd, Farhad Arbab, and Lorinda Cherry for their detailed reading-cum-comments of an earlier draft, Yannis Haralambous for answering questions about his work and providing good friendly competition, Anoosh Hosseini for answering questions about Persian formatting, Brian Kernighan and Nils-Peter Nelson for answering questions about ditroff, Pierre MacKay for answering questions about his work and about Arabic formatting in general, and Murat Tayli for sending to them copies of proceedings of computer Arabization conferences and teaching us modern Arabic words for some computer science terminology.

This paper has used trademarked names strictly for the purpose of identifying the trademarked products; there is no attempt herein to usurp the rights of their owners.

REFERENCES

1. *Proceedings of the First KSU Symposium on Computer Arabization*, Riyadh, Saudi Arabia, 1987.
2. *Proceedings of the Ninth National Computer Conference*, Riyadh, Saudi Arabia, 1986.
3. *Proceedings of the Tenth National Computer Conference*, Jeddah, Saudi Arabia, 1988.
4. *Proceedings of the Eleventh National Computer Conference*, Dharan, Saudi Arabia, 1989.
5. P.A. MacKay, *Computers and the Arabic Language, Proceedings of the Arab School of Science and Technology*, The Hemisphere Press, New York, Washington, Philadelphia, London, 1990.
6. M. Tayli and A.I. Al-Salamah, 'Building Bilingual Microcomputer Systems', *Communications of the ACM*, **33** (5), 495–504 (1990).
7. Z. Wu, W. Islam, J. Jin, S. Janbolatov, and J. Song, 'A Multi-Language Characters Operating System on IBM PC/XT Microcomputer', in *Proceedings of Second International Conference on Computers and Applications*, Beijing, PRC, 1987, pp. 579–585.
8. J.D. Becker, 'Arabic Word Processing', *Communications of the ACM*, **30** (7), 600–611 (1987).
9. D.E. Knuth and M.F. Plass, 'Breaking Paragraphs into Lines', *Software—Practice and Experience*, **11**, 1119–1184 (1981).
10. Mahdi ElSayed Mahmud, *Learning Arabic Calligraphy: Naskh, Requah, Tholoth, Farsi*, Ibn Sina, Publisher, Cairo, Egypt, 1987.
11. مهدي السيد محمود ، كيف تتعلم الخط العربي: نسخ ، رقع ، ثلث ، فارسي ، مكتب بن سينا ، للنشر والتوزيع والتصدير ، القاهر ،مصر ، ١٩٨٧.
12. J. André and B. Borghi, 'Dynamic Fonts', *PostScript Language Journal*, **2** (3), 4–6 (1990).
13. *Interleaf Workstation Publishing Software User's Guide*, Interleaf, Inc., 1986.
14. *FrameMaker Reference*, Frame Technology Corporation, San Jose, CA, 1990.
15. B.W. Kernighan, 'A Typesetter-independent TROFF', *Computing Science Technical Report No. 97*, Bell Laboratories (1982).
16. D.E. Knuth, *The TEXbook*, Addison-Wesley, Reading, MA, 1984.
17. D.E. Knuth, *TEX: The Program*, Addison-Wesley, Reading, MA, 1986.
18. B.W. Kernighan and L.L. Cherry, 'Typesetting Mathematics — User's Guide (Second Edition)', *Technical Report*, Bell Laboratories (1978).
19. M.E. Lesk, 'TBL — A Program to Format Tables', *Technical Report*, Bell Laboratories (1978).

20. B.W. Kernighan, 'PIC — A Graphics Language for Typesetting, Revised User Manual', *Computing Science Technical Report No. 116*, Bell Laboratories (1984).
21. C. J. Van Wyk, 'IDEAL User's Manual', *Computing Science Technical Report No. 103*, Bell Laboratories (1981).
22. N. Batchelder and T. Darrell, 'Psfig — A DITROFF Preprocessor for POSTSCRIPT Figures', *Technical Report*, Computer and Information Science Department, University of Pennsylvania, Philadelphia, PA.
23. M.E. Lesk, 'Some Applications of Inverted Indexes on the UNIX System', *Computing Science Technical Report No. 69*, Bell Laboratories (1978).
24. K.K. Abe and D.M. Berry, 'indx and findphrases, A System for Generating Indexes for Ditroff Documents', *Software—Practice and Experience*, **19** (1), 1–34 (1989).
25. 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 (1985).
26. Z. Becker and D.M. Berry, 'triroff, an Adaptation of the Device-Independent troff for Formatting Tri-Directional Text', *Electronic Publishing*, **2** (3), 119–142 (1990).
27. B.W. Kernighan and C.J. van Wyk, 'Page Makeup by Postprocessing Text Formatter Output', *Computing Systems*, **2** (2), 103–132 (1989).
28. *TRANSCRIPT Software Package*, Adobe Systems Incorporated, Menlo Park, CA, 1986.
29. *GHOSTSCRIPT 2.4.1 POSTSCRIPT Previewer*, Aladdin Enterprises, Menlo Park, CA, 1992.
30. J.L. Bentley and B.W. Kernighan, 'GRAP — A Language for Typesetting Graphs, Tutorial and User Manual', *Computing Science Technical Report No. 114*, AT&T Bell Laboratories, Murray Hill, NJ (1984).
31. J.L. Bentley, 'Little Languages for Pictures in AWK', *AT&T Technical Journal*, **68** (4), 21–32 (1989).
32. J.L. Bentley, L.W. Jelinski, and B.W. Kernighan, 'CHEM—A Program for Phototypesetting Chemical Structure Diagrams', *Computers and Chemistry*, **11** (4), 281–297 (1987).
33. E. Foxley, 'Music—A Language for Typesetting Music Scores', *Software—Practice and Experience*, **17** (8), 485–502 (1987).
34. E.R. Ganser, S.C. North, and K.P. Vo, 'DAG—A Program that Draws Directed Graphs', *Software—Practice and Experience*, **18** (11), 1047–1062 (1988).
35. H. Trickey, 'DRAG — A Graph Drawing System', in *Electronic Publishing '88*, ed. J. André and H. van Vliet, Cambridge University Press, Cambridge, UK, 1988, pp. 171–182.
36. T. Wolfman and D.M. Berry, 'flo — A Language for Typesetting Flowcharts', in *Electronic Publishing '90*, ed. R. Furuta, Cambridge University Press, Cambridge, UK, 1990, pp. 93–108.
37. D.E. Knuth and P. MacKay, 'Mixing Right-to-left Texts with Left-to-right Texts', *TUGboat*, **8** (1), 14–25 (1987).
38. M. Tayli, 'Integrated Arabic System, Technical Information and Programming Manual', *Technical Report*, King Saud University, College of Computer and Information Sciences, Riyadh, Saudi Arabia (1988).
39. M. Tayli, 'Integrated Arabic System', in *Proceedings of the First KSU Symposium on Computer Arabization*, Riyadh, Saudi Arabia, 1987, pp. 135–143.
40. J.D. Becker, 'Multilingual Word Processing', *Scientific American*, **251** (1), 96–107 (1984).
41. A. Khettar, M. Nanard, and J. Nanard, 'High Quality Page Make Up For Arabic Documents', in *Protext II, Proceedings of the Second International Conference on Text Processing Systems*, ed. J.J.H. Miller, Boole Press, Dublin, Ireland, 1985, pp. 162–167.
42. S. Sami and O. Alameddine, 'Generation Of High Quality Arabic Computer Output', in *Computers and The Arabic Language, Proceedings of the Arab School of Science and Technology*, ed. P.A. MacKay, The Hemisphere Press, New York, Washington, Philadelphia, London, 1990, pp. 171–182.
43. Y. Haralambous, 'Arabic, Persian and Ottoman TEX for Mac and PC', *TUGboat*, **11** (4), 520–524 (1990).
44. J.J. Goldberg, 'Approximate TEX for Semitic Languages', in *Conference Proceedings Ninth Annual Meeting of the TEX Users Group, TEXNiques*, ed. C. Thiele, Montréal, 1988, pp. 171–178.
45. Y. Haralambous, 'TEX and Those Other Languages', *TUGboat*, **12** (4), 539–548 (1991).

46. 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).
47. G. Allon and D.M. Berry, ‘MINIX.XINIM, Towards a Bi-Directional, Bi-Lingual UNIX Operating System’, in *Proceedings of the Soviet UNIX User’s Group Conference*, Moscow, USSR, 1991, pp. 8–21.
48. D.E. Knuth, ‘Device-Independent File Format’, *TUGboat*, **3** (2), 14–19 (1982).

APPENDIX I

eqn Examples

وقال ٱللّٰه :

$$\varepsilon_o \oint \mathbf{E} \cdot d\mathbf{S} = q$$

$$\oint \mathbf{B} \cdot d\mathbf{S} = 0$$

$$\oint \mathbf{B} \cdot d\mathbf{l} = \mu_o \varepsilon_o \frac{d\Phi_{\mathbf{E}}}{dt} + \mu_o i$$

$$\oint \mathbf{E} \cdot d\mathbf{l} = \frac{-d\Phi_{\mathbf{B}}}{dt}$$

∴

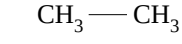
$$c = \frac{1}{\sqrt{\mu_o \varepsilon_o}}$$

وكان نور

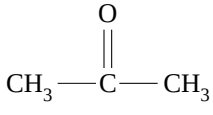
chem Examples

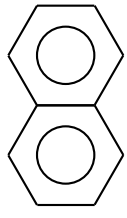
١ . معادل بسيط :

٢ . سلسل لكيلي :



٣ . حلقات خرى :





The Rabbit and the Elephant, from Kalila and Dimna

رنب والفيل

زعموا ن رض من رض الفيل ، تتابعت عليها السنون وجدبت، فقل الماء في تلك البلاد وغارت العيون، وصاب الفيل عطش شديد. فشكت ذلك ملكها. فرسل الملك رسله ورواده في التماس الماء في كل ناحي. فرجع ليه بعض رسله فخبروه بنهم وجدو في بعض ا مكن عين تدعى القمري ، كثير الماء. فتوجه ملك الفيل بفيلته تلك العين ليشربن منها. وكانت تلك ا رض رض رانب. فوطئت الفيل ا رانب برجلها في جحرتها فهلكن كثرها. فاجتمع البقي منها ملكها فقلن له: قد علمت ما صابنا من الفيل ، فاحتل لنا قبل رجوعهن علينا؛ فنهن راجعات لوردهن ومفنيات عن خرنا. فقال ملكهن: ليحضرني كل ذي ري بريه. فتقدم خزر منها يقال له فيروز، وقد كان الملك عرفه با دن والري، فقال: ن رى الملك ن يبعثني الفيل ويبعث معي مين يرى ويسمع ما قول وما صنع ويخبره به، فليفعل. فقال له ملك ا رانب: نت ميني، ونا رضى ريك، وصدق قولك؛ فانطلق الفيل وبلغ عني ما حببت، واعمل بريك، واعلم ن الرسول، به وبريه ودبه يعتبر عقل المرسل وكثير من شنه. وعليك باللين والمواتا ؛ فن الرسول هو يلين القلب ذا رفق، ويخشن الصدر ذا خرق. فانطلق ا رنب في ليل ، القمر فيها طالع، حتى انتهى موضع الفيل . فكره ن يدنو منهن فيطنه برجلهن، ون لم يردن ذلك، فشرف عـ تل فنادى ملك الفيل باسمه وقال له: ن القمر رسلني ليك، والرسول مبلغ غير ملوم، ون غلط في القول. فقال له ملك الفيل : وما الرسال؟ قال: يقول لك القمر نه من عرف فضل قوته عـ الضعفاء

فاعتبر بذلك من ا قوياء ، كانت قوته حينا وبالاً عليه؛ و نـــك قد عرفت فضل قوتك ع الدواب فغرك ذلك مني فعمدت عيني التي تسمى باسمي فشربت ماءها وكدرته نت و صحابك؛ و ني تقدم ليك و نذرك لا تتيها ف عشي بصرك و تلف نفسك. و ن كنت في شك من رسالتي، فهلم العين من ساعتك، ف ني موافيك بها. فعجب ملك الفيل من قول فيروز، وانطلق معه العين. فلما نظر ليها رى ضوء القمر في الماء. فقال له فيروز: خذ بخرطومك من الماء واغسل وجهك واسجد القمر. ففعل. ولما دخل خرطومه الماء فحركه، خيل ليه ن الماء يرتعد، فقال ملك الفيل : وما ش ن القمر يرتعد؟ تراه غضب من دخال جحفلتي في الماء؟ قال: نعم، فاسجد له. فسجد الفيل للقمر وتاب ليه مما صنع، وشرط له لا يعود هو ولا حد من فيلته العين.

APPENDIX II

Arabic-Persian Character Set

Number	Phonetic	Name	Stand-alone	Connected- both	after	before
Principal Letters (Arabic & Persian)						
1	‘	hamza	ء	—	—	—
2	a	alef	ا	ـا	ا	ـا
3	b	baa	ب	ـبـ	بـ	ـب
4	t	taa	ت	ـتـ	تـ	ـت
5	c	thaa	ث	ـثـ	ثـ	ـث
6	j	jeem	ج	ـجـ	جـ	ـج
7	h	haa	ح	ـحـ	حـ	ـح
8	x	chaa	خ	ـخـ	خـ	ـخ
9	d	dal	د	ـد	د	ـد
10	Z	thal	ذ	ـذ	ذ	ـذ
11	r	raa	ر	ـر	ر	ـر
12	z	zein	ز	ـز	ز	ـز
13	s	seen	س	ـسـ	سـ	ـس
14	C	sheen	ش	ـشـ	شـ	ـش
15	S	Sad	ص	ـصـ	صـ	ـص
16	D	Dad	ض	ـضـ	ضـ	ـض
17	T	Tah	ط	ـطـ	طـ	ـط
18	Z‘	dhah	ظ	ـظـ	ظـ	ـظ
19	e	ain	ع	ـعـ	عـ	ـع
20	R‘	rain	غ	ـغـ	غـ	ـغ
21	f	faa	ف	ـفـ	فـ	ـف
22	q	qaf	ق	ـقـ	قـ	ـق

23	**k**	caf	ك	ك	ك	ك
24	**l**	lam	ل	ل	ل	ل
25	**m**	meem	م	م	م	م
26	**n**	noon	ن	ن	ن	ن
27	**H**	hea	ه	ه	ه	ه
28	**w**	waw	و	و	و	و
29	**y**	yaa	ي	ي	ي	ي
30	**t‘**	taa_marbouta				
31	**A‘**	alef_maksura	ى	ى	ى	ى

Hamza Letters

32	**A**	hamza_on_alef				
33	**i**	hamza_under_alef				
34	**Y‘**	hamza_on_yaa				
35	**H‘**	hamza_on_hea				
36	**w‘**	hamza_on_waw				

Persian Letters

37	**p**	paa	پ	پ	پ	پ
38	**G**	geem	چ	چ	چ	چ
39	**g**	jeh	ژ	ژ	ژ	ژ
40	**v**	vaa	ڤ	ڤ	ڤ	ڤ
41	**Q**	Gaf	گ	گ	گ	گ

Other Letters

42	**a~**	madda_on_alef	ٓ			
43	**U**	hamzat_wasel		_	_	_

44	**\(ak**	alef_kasira	ٍ	—	—	—
45	**\(md**	madda	ٓ	—	—	—

Ligatures do not have their own phonetic spellings.
They are recognized automatically as the combination of other letters.

Number	Name	Stand-alone	Connected- both	after	before
Ligatures					
46	lamalef	لا	ﻟﺎ	لا	ﻻ
47	hamza_on_lamalef				
48	hamza_under_lamalef				
49	lamalef_maksura		—	—	
50	madda_on_lamalef				
51	madda_on_alef				
52	tanweenfateh_on_alef				
53	lam_meem	لم	—	ـلـ	—
54	lam_yaa	لي	—	—	ﻟﻲ
55	faa_yaa	في	—	—	—

The rest do not have any form but stand-alone.

Number	Phonetic	Name	Stand-alone
Vowels/Diacriticals			
56	**’ \(ft**	fatha	◌َ
57	**u \(dm**	dammah	◌ُ
58	**E \(ks**	kasra	◌ِ

69	**O \(sn**	sukun	◌ْ
60	**~~ \(sh**	shaddah	◌ّ
61	**~~’ ’~~ \(sf**	fatha_on_shaddah	
62	**~~u u~~ \(sd**	dammah_on_shaddah	
63	**~~E E~~ \(sk**	kasra_under_shaddah	
64	**~~’’ \(st**	tanweenfateh_on_shaddah	
65	**~~uu \(su**	tanweendamm_on_shaddah	
66	**~~EE \(sv**	tanweenkaser_under_shaddah	

Special

67	**L**	allah	ﷲ

International Characters

68	**!**	exclamation_mark	!
69	**$**	currency_sign	$
70	**#**	number_sign	#
71	**%**	percent	٪
72	**&**	ampersand	&
73	**\(lq**	left_quote	«
74	**\(rq**	right_quote	»
75	**)**	left_parenthesis	(
76	**(**	right_parenthesis	)
77	*****	asterisk	*
78	**+**	plus_sign	+
79	**,**	arabic_comma	،
80	**-**	minus_sign	-
81	**/**	slash	/
82	**@**	at	@

83	]	left_bracket	[
84	\	back_slash	\
85	[	right_bracket	]
86	^	hat	^
87	_	under_score	_
88	}	left_brace	{
89	|	bar	|
90	{	right_brace	}
91	>	less_sign	<
92	<	greater_sign	>
93	=	equal_sign	=
94	?	question_mark	؟
95	;	semicolon	؛
96	:	colon	:

Digits

96	0	zero	٠
97	1	one	١
98	2	two	٢
99	3	three	٣
100	4	four	٤
101	5	five	٥
102	6	six	٦
103	7	seven	٧
104	8	eight	٨
105	9	nine	٩