Understanding Bidirectional (BIDI) Text in Unicode

By **Cal Henderson**, March 1st 2009.

The Basics

A little-understood corner of Unicode is its handling for bidirectional text (The **spec** is a little dry). While English languages are read left-to-right, plenty of scripts (notably Arabic and Hebrew) are read from right to left. When only a single direction of text is used in a document, it's fairly straight forward, but when texts with different directions are mixed in one document, some difficulty arises in determining direction. This document attempts to explain how bidirectional text in Unicode works and what this means for the web.

In the Unicode standard, characters have a *representational order* in memory (which English speakers tend to think of as left to right, but is really start-to-finish in a file), which the bidirectional algorithm then operates on to determine the display characteristics. If a file contains the codepoint for A (**U+0041**) followed by the codepoint for B (**U+0042**), they will be displayed with the A to the left of the B. If a file contains Arabic Alef Wasla (**U+0671**) followed by Arabic Tteh (**U+0679**) then the Alef Wasla will appear on the right and the Tteh on the left.

Note: Many of the examples in this article will only work on browsers which support bidirectional rendering and have a font which supports Arabic.

Arabic Alef Wasla	U+0671 ا	ٱ
Arabic Tteh	U+0679 ٹ	ٹ
U+0671 followed by U+0679 ا ٹ		ٱٹ

But how does this work? Not magic, but science. Every code point in the Unicode standard has a directionality assigned to it, along with a strength. The Latin characters A and B are Strong Left-to-Right (LTR), while the Arabic characters used in the example are Strong Right-to-Left (RTL). This tells the renderer how to order them for display.

The Weak Characters

The *Strong* characters are easy, but the complexity comes around the *Weak* ones (There are also some *Neutral* characters, but they are things like whitespace and punctuation and they take direction from their surroundings). Weak characters have a direction which is used when they appear alone, but they take on special powers when embedded in Strong characters in the opposite direction - they don't create a directional *boundary*, so the surrounding characters keep their ordering. For an example, we'll introduce a third Strong Right-to-Left character, the Arabic Nyeh (**U+0683**) and use some Weak Left-to-Right characters, 3 (**U+0033**) and 7 (**U+0037**).

Arabic Nyeh	U+0683 ڃ	ڃ
U+0671 'A' 'B' U+0679 U+0683 ا ب ڃ		اٌAB ٹڃ
U+0671 '3' '7' U+0679 U+0683 ا ب ڃ		اٌ37 ٹڃ

Where the Strong characters meet in the first example, the text is split into three separate blocks and each is then rendered (the last two characters are rendered RTL). When Strong meets Weak, the Weak characters are rendered in order (LTR), but the Strong characters remain a single block, rendered RTL.

But hey, something here is missing. What happens if we have an A followed by a Nyeh. They are both Strong, so we render them separately. But once we've decided how these blocks render (easy! they only contain one character), what determines their display order? That's done by the document order, which is LTR by default on this page. We can change that in HTML by using the `dir="rtl"` attribute. Let's see what happens (I've thrown in a B so you can see that groups of Strong characters

always maintain their order locally).

A B U+0683 `dir="ltr"`	AB
A B U+0683 `dir="rtl"`	AB

Implicit Markers

Ok! So far so easy. There are some complications, however, when we embed one direction of text in another. Imagine we want to embed an Arabic quote in English text. Our Arabic quote (for simplicity) is "Alef Wasla, Tteh, Exclamation Mark, Nyeh". Our English text is 'he said "quote" to her'. The code points for this are pretty simple and the result is as you might expect.

he said "U+0671 U+0679 ! U+0683" to her	he said "!" to her

The whole Arabic quote is RTL, with an embedded exclamation mark (It's a Neutral, so doesn't have any ordering). But what happens if we remove the Nyeh and leave the exclamation mark on the end?

he said "U+0671 U+0679 !" to her	he said "!" to her

The exclamation mark isn't RTL and isn't embedded in the Strong RTL characters, so it inherits direction from the document. This is where the implicit marker characters come in - Left-to-Right Mark (**U+200E**, LRM) and Right-to-Left Mark (**U+200F**, RLM). They act as zero width characters

with Strong direction, so that Neutral and Weak characters between them and the Strong characters are included in the Strong character block. Let's put one after the exclamation mark in the above example and see what happens.

he said "U+0671 U+0679 ! U+200F" to her	he said "ٱٹ!" to her

Excellent! Implicit markers are the recommended way to hint about the direction of Neutral characters in bidirectional documents.

Explicit Markers

There are two more kinds of direction marker in Unicode - the Embeds and the Overrides. The embeds are the easiest and we'll deal with those first.

Left-to-Right Embedding (**U+202A**, LRE) and Right-to-Left Embedding (**U+202B**, RLE) are used to mark a block of text as having a direction, much like the overall document direction. This then affects Weak characters at the boundaries which would have otherwise taken direction from the document. An embedded block is terminated with a Pop-Directional-Formatting (**U+202C**, PDF). As the name implies, this character 'pops' the state - each LRE and RLE pushes a state change onto a stack and each PDF pops one state from the stack (always the most recently pushed state). This allows nested blocks. For example:

start {LRE} foo {RLE} bar {PDF} baz {PDF} end

In this example, `start` and `end` are outside any embedding, `foo` and `baz` are in a Left-to-Right embedding and `bar` is in a Right-To-Left embedding. Embedding blocks can be used in place of LRM and RLM markers to ensure that punctuation and other Neutrals are displayed in the correct order if you know that an entire span of text needs to have a certain direction. Use of these embeddings, however, is discouraged. Because every push must be matched by a pop, this can lead to confusing situations when they are not fully balanced.

The second form of explicit markers are the overrides - Left-to-Right Override (**U+202D**, LRO) and Right-to-Left Override (**U+202E**, RLO). These work in much the same way as the Embeddings,

pushing state onto the stack which must be popped with a PDF character. However, the way they affect text within them is to override the directionality to be Strong in the indicated direction. The purpose is to allow things like mixed English and Arabic numbers and maintain a single direction, but they can also be used to redirect Strong characters:

U+202E A B C U+202C	CBA
U+202D U+0671 U+0679 U+0683 U+202C	ٱ ٹ ڃ

There isn't often any call to use these overrides, and their use is discouraged for the same reason as the embeddings - they must be balanced correctly with PDFs.

Bidirectional Text in a Nutshell

So there you have it. Unicode mostly takes care of the direction of text itself, by assigning directions and strengths to every character. If you know that your text is mostly RTL, then set `dir="rtl"` on your `<body>` element to ensure that Neutrals at the borders of Strong character blocks are treated correctly.

In some odd situations with embedded text with a different direction, we can use the implicit markers (U+200E and U+200F) at the start and end of the embedded text to make sure any Neutral characters at the borders are set in the correct direction. We could also use embeddings, but these are harder to use.

If we want even greater control, changing the direction of non-Neutral characters, we can use overrides. This is rarely necessary, however.

Filtering User Input

When we write a web-based application which accepts Unicode text, we need to be careful about the sequences of codepoints we allow. This is even more so the case for bidirectional text.

The implicit markers are not generally a big worry, because they only affect Weak and Neutral characters that sit adjacent to the user-entered text. Let's imagine an example, on a LTR site, where we let users choose a username and then we embed it in an English sentence: "Hello {username},

how are you?". There is a Neutral comma after the username, but any RLM in the name will not affect it since it's adjacent to the Strong LTR 'h' and the document order is LTR. Let's take a look:

Username:	ا ب U+0671 U+0679 U+200F
Hello آٹ, how are you?	

The marker here has no effect. We can safely ignore markers in input text - they can be used to control Neutrals and blocks of Weak characters within the input text, but won't affect text surrounding it.

However, the embedding markers will affect Neutrals that follow them. Let's use the above example, but instead use a RLE at the end of the username.

Username:	ا ب U+0671 U+0679 U+202B
Hello how are you? ,آٹ	

This treats the rest of the sentence as an embedded RTL block. It doesn't go past the block (pushed states only last until the end of the paragraph), but it still messes up our display. The explicit override markers cause even more trouble.

Username:	ا ب U+0671 U+0679 U+202E
Hello ?uoy era woh ,آٹ	

Here, not only does it apply RTL to the rest of the sentence, but it overrides the direction of our Strong English characters. Slightly trippy to see this on your website, but ultimately bad. So what can

we do? Well, we could disallow these explicit characters (U+202A-U+202E) which is pretty easy. This does mean that anybody who wants to use them to include Neutrals at the edges of their Arabic usernames will be out of luck - and that sucks more when it's a comment they're posting, where the period jumps to the 'beginning' of the text.

If we want to allow use of these characters, the solution is fairly simple (if hard to implement): we need to make sure that every opening marker has a paired closing marker (PDF) so that the state stack coming out of the string is at the same state as when we went in. We also need to be careful that we don't allow any PDFs to be used without accompanying push markers, else we can't use any ourselves outside of the block.

PHP Filtering Code

Here's a PHP function which does just that when given a UTF-8 encoded string. It's probably not the best way to do it, but it works:

```
function unicode_cleanup_rtl($data){

        #
        # LRE - U+202A - 0xE2 0x80 0xAA
        # RLE - U+202B - 0xE2 0x80 0xAB
        # LRO - U+202D - 0xE2 0x80 0xAD
        # RLO - U+202E - 0xE2 0x80 0xAE
        #
        # PDF - U+202C - 0xE2 0x80 0xAC
        #

        $explicits      = '\xE2\x80\xAA|\xE2\x80\xAB|\xE2\x80\xAD|\xE2\x80\xAE';
        $pdf            = '\xE2\x80\xAC';

        preg_match_all("!$explicits!",  $data, $m1, PREG_OFFSET_CAPTURE | PREG_SET_ORDER);
        preg_match_all("!$pdf!",        $data, $m2, PREG_OFFSET_CAPTURE | PREG_SET_ORDER);

        if (count($m1) || count($m2)){

                $p = array();
                foreach ($m1 as $m){ $p[$m[0][1]] = 'push'; }
                foreach ($m2 as $m){ $p[$m[0][1]] = 'pop'; }
                ksort($p);

                $offset = 0;
                $stack = 0;
                foreach ($p as $pos => $type){

                        if ($type == 'push'){
                                $stack++;
                        }else{
                                if ($stack){
                                        $stack--;
                                }else{
                                        # we have a pop without a push - remove it
                                        $data = substr($data, 0, $pos-$offset)
                                                .substr($data, $pos+3-$offset);
                                        $offset += 3;
                                }
                        }
                }
```

```
                # now add some pops if your stack is bigger than 0
                for ($i=0; $i<$stack; $i++){
                        $data .= "\xE2\x80\xAC";
                }

                return $data;
        }

        return $data;
}
```

The function works by finding the positions of every push and pop marker in a string, sorting them into an ordered list then going through them one by one, keeping track of the stack size. If we see a PDF when the stack is empty, we remove it. If we reach the end of the string and the stack isn't empty, we append PDFs until it is.

With this function, we can accept RTL user data and safely display it in a LTR site (or vice-versa!) and know that it won't change direction of any of our own text. Of course, there's plenty of other bad stuff we still need to filter out :D

Post-Script

While I'm a bit of a Unicode nerd, I'm no expert. The above article may contain mistakes or fail to mention very important parts of the bidi algorithm. If you spot any glaring mistakes or omissions, drop me an email and teach me: cal [at] iamcal.com