Scheme for Client-Side Scripting in Mobile Web Browsing

or AJAX-Like Behavior Without Javascript

Ray Rischpater
Rocket Mobile, Inc.
ray@rocketmobile.com

Abstract

I present an implementation of Scheme embedded within a Web browser for wireless terminals. Based on a port of TinyScheme integrated with RocketBrowser, an XHTML-MP browser running on Qualcomm BREW-enabled handsets. In addition to a comparison of the resulting script capabilities, I present the changes required to bring TinyScheme to Qualcomm BREW, including adding support for BREW components as TinyScheme data types. The resulting application supports the same kinds of dynamic client-side scripted behavior as a traditional Javascript-enabled Web browser in environments too memory constrained for a Javascript implementation.

Keywords Scheme, Web, JavaScript, AJAX, mobile computing, Qualcomm BREW

1. Introduction

In the last twenty-four months, many commercial sites have deployed highly interactive Web applications leveraging the flexibility of XHTML[1], JavaScript[2], and XML[3]-based Web services. Popular sites including Google Maps[4] have both inspired Web developers and heightened consumer expectations of Web sites. This approach has significantly decreased the apparent latency of many Web applications, enhancing the user experience for all. At the same time, wide-area wireless carriers have deployed third-generation wireless networks with performance roughly equivalent to commercial broadband to the home solutions, resulting in a marked increase by consumers in the use of wireless terminals for Web access. Moreover, in an attempt to recoup network costs and increase the average revenue per subscriber (ARPU), many wireless carriers have deployed network-based programming environments such as Qualcomm BREW[5] that enables third-party developers to create and market applications that run on subscriber terminals. These applications run the gamut from entertainment (news, sports, and games) to personalization applications that permit consumers to customize the look and feel of their terminal as they purchase screen backgrounds, ring tones, and even whole handset themes that re-skin the handset's user interface.

While custom applications for these handsets can be built entirely using the native platform in C or C++, many developers have opted for hybrid native/Web-based solutions, in which interactive content is presented in a variety of ways using both native user interface controls and XHTML rendered using either the device's native Web browser or a custom part of the application capable of displaying XHTML. Rocket Mobile's RocketBrowser, an XHTML-MP[6] capable Web browser written in C for mobile handsets running Qualcomm BREW, is an example of this approach. This application provides a framework for building Web-enabled applications that run on wireless terminals, permitting application developers to introduce new tags and Web protocols that trigger native operations (compiled into the application as C functions) as well as mix the viewing of Web pages with native interfaces built using Qualcomm BREW interfaces in C. Complex applications with client-server interactions can be built quickly using RocketBrowser as a starting point, mixing traditional browser activities (say, permitting you to browse a catalog of personalization assets such as ring tones or screen backgrounds for your terminal) with activities that require support from the platform APIs (such as purchasing and installing desired media).

While flexible—developers can freely mix the browser and traditional application metaphors—this approach has some serious limitations. Notably, software behaviors must be implemented in C or C++ and compiled into the application; there is no facility for downloading additional application components. The impact of this limitation is amplified by the typical deployment scenario, in which a developer signs the application package, a third party certifies the application for quality and network interoperability, adds their signature cryptographically, and only then can the application be released to wireless operators for distribution. Once released, the application cannot be changed without passing through this entire cycle again, making offering new program features to consumers is a costly and time-consuming process. This approach also suffers the usual limitations of solutions written in C or other low-level languages, including the need to compile code for execution, the need for explicit memory management, and the lack of support for higher-order functions.

One way to ameliorate these problems is to introduce a client-side scripting solution such as JavaScript or its kin ECMAScript[7]. A scripting language such as JavaScript has the obvious advantages of rapid-high level development on the client, permitting developers to focus on the problems at hand in the content environment rather than the mechanics of extending the Web browser's implementation. Scripting also permits downloading patches or additional application components; portions of the application written as scripts interpreted on the client side can be updated in a networked application. Oddly, although this presents obvious weaknesses in the certification requirements set by most wireless operators, most carriers today permit scripted behavior in certified applications, provided that the documentation accompanying the application provided at the time of certification provides ample explanation of what behavior may be changed through script execution dur-

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

ing application operation.[1] As a result, the addition of a client-side script engine can help reduce the time-to-market for new features such as user interface enhancements.

However, while JavaScript is the de facto standard for client-side scripting in Web applications on the desktop it's not without drawbacks in mobile computing. Notably, the footprint of a JavaScript implementation is actually quite large; as I discuss on page 152 in section 5, the SpiderMonkey JavaScript runtime from the Mozilla foundation is nearly twice the size of the solution I describe here on the debugging target and significantly more complex to port and maintain. While writing a simpler JavaScript implementation is possible, doing so was not desirable in the scope of our work; the time and scope of the project preferred porting an existing code base to re-implementing in whole or part an existing solution. This was true even at the expense of compatibility, because the resulting application is used by only a handful of resources internal to our firm to create end-user prototypes and applications, not the general public.

Because of these considerations—code complexity, static memory use, and time alloted for the project—when faced the need for client-side scripting in RocketBrowser to support asynchronous transactions made popular by sites using JavaScript, I decided that embedding an implementation of Scheme[8] within the application would be an appropriate choice.

This paper describes the application of Scheme to client-side scripting within a a mobile Web browser application and shows current and future applications of the technology in commercial applications. In the following section, "Preliminaries", I first present some previous work in this area that strongly influenced my approach. I describe the work required to integrate Scheme in our application, RocketBrowser, in the subsequent section "Implementation". Because there's more to shipping an application than just providing a platform, the section "Examples" shows two ways we're currently using Scheme within the resulting application at Rocket Mobile. I summarize the memory footprint of the resulting application in the section "Results", and then close the paper by describing what we've learned in the process of using Scheme for our client-side scripts in the section titled "Future Work". Finally, an appendix provides some additional detail regarding the work involved in bringing a Scheme interpreter to the Qualcomm BREW platform.

2. Preliminaries

Both client-side scripting in Web applications and Scheme applications to Web programming have a distinguished history for so young an application of computing as the Web. Understanding this history provides a crucial understanding of my motivation in selecting Scheme as a client-side language for our browser products.

2.1 Client-Side Scripts in Web Browsing

Client-side scripting in Web browser applications is not new; the initial JavaScript implementation by Brendan Eich was provided as part of Netscape 2.0 in 1995. Scripts in Web content are introduced using the `<script>` tag, which treats its content as XML CDATA, permitting scripts to consist of un-escaped character data, like so:

```
<html>
  <body>
    <script language="javascript">
     document.write('<p>Hello world.</p>')
    </script>
  </body>
</html>
```

Using a a client-side scripting language such as JavaScript or ECMAScript, content developers can write scripts that:

- Access the content of the document. JavaScript provide access to a document's contents via the model constructed by the Web browser of the document, called the Document Object Model (DOM)[9]. The DOM provides mechanisms for accessing document objects by an optional name or unique ID as well as by an object's position within the document (e.g., "the third paragraph").
- Define functions. Scripts can define functions that can be invoked by other scripts on the same page either as parts of computation or in response to user action.
- Interact with the user interface. XHTML provides attributes to many tags, including the `<body>` tag, that permit content developers to specify a script the browser should execute when a particular user action occurs. For example, developers can use the `onmouseover` attribute to trigger a script when you move the mouse cursor over the contents of a specific tag.
- Obtain content over the network. On most newer browsers, scripts can use the browser's network stack via an object such as `XMLHTTPRequest`[11] or the Microsoft ActiveX object `XMLHTTP`[12]. Using one of these interfaces a script can create a Web query, make the request over the network, and have the browser invoke a callback when the query is completed.[2]

As these points show, the flexibility of today's Web browser applications is not simply the outcome of JavaScript the language, but rather the relationship between the scripting run-time, the ability of scripts to access the DOM and user events, and the ability of scripts to obtain data from servers on the network. The union of these characteristics enables the development of asynchronous networked applications residing entirely within a set of pages being viewed by a Web browser, a strategy popularly known as Asynchronous JavaScript and XML[10] (AJAX).

2.2 Scheme and the Web

Scheme plays an important part of many Web applications, from providing scripting for servers[14] such as Apache[14] to providing servers written in Scheme providing entire Web applications, such as the PLT Web server[15] within PLT Scheme. Much has been made in both the Lisp and Scheme communities about the relationship between XML and S-expressions; recent work on SXML[16] demonstrates the advantages of working with XML within a functional paradigm. At the level of this work, those resources did not significantly affect how I went about integrating a Scheme interpreter with the RocketBrowser, but rather helped build Scheme's credibility as the choice for this problem. As I suggest later on page 153 in section 6, SXML holds great promise in using a Scheme-enabled Web browser as the starting point for building applications that use either XML Remote Procedure Call[17] (XML-RPC) or Simple Object Access Protocol[18] (SOAP) to interact with Web services.

3. Implementation

I selected the TinyScheme[19] for its size, portability, and unencumbered license agreement. This choice turned out to be a good

[1] Most operator's guidelines are vague in this regard. The position of the operator is that under no circumstances may an application interfere with a wireless terminal's use as a telephone; consequently, assurances that scripted behavior cannot interfere with telephony operations often appears ample for certification.

[2] These objects do far more than broker HTTP[13] requests for scripts. As their name suggests, they also provide XML handling capabilities, including parsing XML in the resulting response.

one; initial porting took only a few days of part-time work, and packaging both the interpreter and foreign function interfaces to the interpreter for access to handset and browser capabilities was straightforward.

TinyScheme is a mostly R^{5}RS[8] compliant interpreter that has support for basic types (integers, reals, strings, symbols, pairs, and so on) as well as vectors and property lists. It also provides a simple yet elegant foreign function interface (FFI) that lets C code create and access objects within the C runtime as well as interface with native C code. Consisting of a little over 4500 lines of C using only a handful of standard library functions, TinyScheme was an ideal choice.

Once TinyScheme was running on the wireless terminal, I integrated the TinyScheme implementation with our RocketBrowser application. This work involved adding support for the `<script>` tag as well as attributes to several other tags (such as `<body>` and `<input>`) to permit connecting user events with Scheme functions. This involved changes to the browser's event handler and rendering engine, as well as the implementation of several foreign functions that permit scripts in Scheme to access the contents of the document and request content from the browser's network and cache layers.

3.1 Bringing Scheme to the Wireless Terminal

From a programmer's standpoint, today's wireless terminals running platforms such as Qualcomm BREW are quite similar to traditional desktop and server operating systems, despite the constrained memory and processing power. There were, however, some changes to make to TinyScheme before it could run on the wireless terminal:

- Elimination of all mutable global variables to enable the application to execute without a read-write segment when built using the ARM ADS compiler for Qualcomm BREW.
- Implementation of all references to standard C library functions, typically re-implemented as wrappers around existing Qualcomm BREW functions that play the role of standard C library functions.
- Initialization of the Scheme opcode table of function pointers at run time, rather than compile time to support the relocatable code model required by Qualcomm BREW.
- Introduction of a BREW-compatible floating point library to replace the standard C floating point library provided by ARM Ltd.
- Modification of the TinyScheme FFI mechanism to pass an arbitrary pointer to an address in the C heap to permit implementation of foreign functions that required context data without further changes to the TinyScheme interpreter itself.
- Addition of the TinyScheme type `foreign_data` to permit passing references to pointers on the C heap from the FFI layer into Scheme and back again.
- Encapsulation as a Qualcomm BREW extension. TinyScheme and its foreign function interface are packaged as Qualcomm BREW extensions, stand-alone components that are referenced by other applications through the wireless terminal's module loader and application manager.
- Capping the amount of time the interpreter spends running a script to avoid a rogue script from locking up the handset.

Readers interested in understanding these changes in more detail may consult the appendix on page 154.

3.2 Integrating Scheme with the Web Browser

The RocketBrowser application is built as a monolithic C application that uses several BREW *interfaces*—structures similar to Windows Component Object Model[24] (COM) components—as well as lightweight C structures with fixed function pointers to implement a specific interface we refer to as *glyphs*. Each RocketBrowser glyph carries data about a particular part of the Web document such as its position on the drawing canvas and what to draw as well as an interface to common operations including allocation, event handling, drawing, and destruction. This is in sharp contrast to many desktop browsers, which use a DOM to represent a document's structure. These two differences: the use of C-based component interfaces and the lack of a true DOM affected the overall approach for both porting TinyScheme and integrating the interpreter with the browser.

To permit Scheme scripts to interface with the browser through the FFI, I wanted to expose a BREW interface to the browser that would allow scripts to invoke browser functions. To do this, I chose to extend TinyScheme's types to add an additional Scheme type that could encapsulate a C structure such as a BREW interface on the C heap. I added a new union type to the structure that represents a cell on the heap, and provided a means by which foreign functions could create instances of this type. Instances of the new `foreign_data` type contain not just a pointer to an address in the application's C heap, but an integer application developers can use to denote the object's type and a function pointer to a finalizer invoked by the TinyScheme runtime when the garbage collector reclaims the cell containing the reference. This new type lets developers building foreign functions pass C pointers directly to and from the Scheme world, making both the Scheme and C code that interface with the browser clearer than referring to glyphs via other mechanisms such as unique identifiers. One such object that can be passed is a BREW interface; its corresponding BREW class id (assigned by Qualcomm) provides its type identifier, and an optional destructor handles reclaiming the object when the Scheme runtime collects the cell containing the instance. Moreover the combination of a user-definable type marker and finalizer function makes the mechanism suitable for sharing a wide variety of objects with a modicum of type safety for foreign function implementers.[3]

One challenge (which I was aware of from the beginning of the project) was the lack of a true DOM within the browser; this was in fact one of the reasons why JavaScript was a less-suitable candidate for a client-side scripting engine, as it requires a fully-featured DOM for best results. As the notion of a glyph encapsulates visible content such as a string or image, there is only a vague one-to-one correspondence between glyphs and tags, let alone glyphs and objects in the DOM. As such, the glyph list maintained by the browser is a flat representation of the document suited for low memory consumption and fast drawing sorted by relative position on a document canvas, and does not provide the hierarchical view of the document content required by a true DOM. Rather than implement the entire DOM atop the browser's list of glyphs, the resulting interface supports only access to named glyphs that correspond to specific XHTML tags such as `<input>`, `<img>`, and `<div>`. To obtain a reference to a named glyph in the current document, I introduce the foreign function `document-at`, which scans the list of glyphs and returns the first glyph found with the indicated name.

In addition to being able to access a specific piece of the DOM, developers must also be able to get and mutate key properties of any named glyph: the text contents of text glyphs such as those corresponding to the `<div>` tag, and the `src` attribute of glyphs

[3] Unfortunately, as seen in the Appendix, the resulting type checking system relies on developers writing and using functions that provide type-safe casts, a mechanism scarcely better than no type checking at all in some settings.

such as image glyphs corresponding to the `<img>` tag. (As I discuss in section 6 on page 153, later extension of this work should provide access to other glyph properties as well.) I defined foreign functions to obtain and set each of these values:

- The `glyph-src` function takes a glyph and returns the URL specified in the indicated glyph's `src` attribute.
- The `glyph-value` function takes a glyph and returns the value of the glyph, either its contents for glyphs such as those corresponding to `<div>` or the user-entered value for glyphs corresponding to `<input>` or `<textarea>`.
- The `set!glyph-src` function takes a glyph and new URL and replaces the `src` attribute with the provided URL. After the interpreter finishes executing the current script, the browser will obtain the content at the new URL and re-render the page.
- The `set!glyph-value` function takes a glyph and string, and replaces the text of the indicated glyph with the new string. After the interpreter finishes executing the current script, the browser will re-render the page with the glyph's new contents.

These functions, along with `document-at`, play the role provided by the JavaScript DOM interface within our Scheme environment.

Finally, I defined the foreign function `browser-get` to support asynchronous access to Web resources from scripts on a browser page. This function takes two arguments: a string containing a URL and a function. `browser-get` asynchronously obtains the resource at the given URL and invokes the given function with the obtained resource. This provides the same functionality as JavaScript's `XMLHTTPRequest` object to perform HTTP transactions.

4. Scheme in Client Content

Implementing a client-side scripting language for RocketBrowser was more than an exercise; I intended it to provide greater flexibility for application and content developers leveraging the platform when building commercial applications. As the examples in this section demonstrates, the results are not only practical but often more concise expressions of client behavior as well.

4.1 Responding to User Events

An immediate use we had for client-side scripting was to create a page where an image would change depending on which link on the page had focus. In JavaScript, this script changes the image displayed when the mouse is over a specific link:

```
<html>
<head>
<script language="javascript">
  function change(img_name,img_src) {
    document[img_name].src=img_src;
  }
</script>
</head>
<body>
  <center>
    <img id="watcher" src="0.jpg"
     height="60" width="80"/>
  </center>
  <br/>
  <table>
    <tr>
      <td>
        <A HREF="1.html">
         onmouseover="change('watcher','1.jpg')"
         onmouseout="change('watcher','0.jpg')">
           1
        </a>
      </td>
      <td>
        <A HREF="2.html">
         onmouseover="change('watcher','2.jpg')"
         onmouseout="change('watcher','0.jpg')">
           2
        </a>
      </td>
      <td>
        <A HREF="3.html">
         onmouseover="change('watcher','3.jpg')"
         onmouseout="change('watcher','0.jpg')">
           3
        </a>
      </td>
      <td>
        <A HREF="4.html">
         onmouseover="change('watcher','4.jpg')"
         onmouseout="change('watcher','0.jpg')">
           4
        </a>
      </td>
    </tr>
  </table>
</body>
</html>
```

This code is straightforward. A simple function `change`, taking the name of an `<img>` tag and a new URL simply sets the URL of the named image to the new URL; this causes the browser to reload and redisplay the image. Then, for each of the selectable links, the XHTML invokes this function with the appropriate image when the mouse is over the link via the `onmouseover` and `onmouseout` attributes.

In the Scheme-enabled browser, I can write:

```
<html>
<head>
<script language="scheme">
  (define resourceid-offsets '(0 1 2 3 4))
  (define focus-urls
    (list->vector
      (map
        (lambda(x)
          (string-append (number->string x) ".jpg"))
        resourceid-offsets)))

  (define on-focus-change
    (glyph-set!-src (document-at "watcher")
      (vector-ref focus-urls browser-get-focus)))
</script>
</head>
<body onfocuschange="
  (on-focus-change browser-get-focusindex)">
<center>
  <img id="watcher" src="0.jpg"
    height="60" width="80"/>
</center>
<table>
<tr>
  <td><a href="1.html">1</a></td>
  <td><a href="2.html">2</a></td>
  <td><a href="3.html">3</a></td>
  <td><a href="4.html">4</a></td>
</table>
```

```
</body>
</html>
```

There are substantial differences here, although the implementation is conceptually the same. The key difference imposing a different approach is a hardware constraint that drives the browser's capabilities: most wireless terminals lack pointing devices, substituting instead a four-way navigational pad. Thus, there's no notion of mouse events; instead, the browser provides `onfocuschange` to indicate when focus has changed from one glyph to the next. A side effect of not having a pointing device is that for any page with at least one selectable item, one item will always have focus; the navigation pad moves focus between selectable items, and a separate key actually performs the selection action (analogous to the mouse button on a PC).

The XHTML opens with a brief set of definitions to establish a vector of URLs, each corresponding to a specific image to be shown when you navigate to a link on the page. This provides a way for content developers to quickly change the images shown on the page by simply updating the list at the start of the script.

The script handling the focus change, `on-focus-change`, performs the same action as its JavaScript counterpart `change`, replacing the target `<img> src` attribute with a new URL. Instead of being supplied with the new URL, however, this function takes an index to the nth selectable glyph on the page. In this case there are four, one for each link (and one corresponding to each URL in the `focus-urls` vector). The browser invokes `on-focus-change` each time the you press a directional arrow moving the focus from one link to another, as directed by the `<body>` tag's `onfocuschange` attribute.

This coupling of XHTML and Scheme replaces several kilobytes of full-screen graphic images, a custom menu control invoked by a client-specific browser tag, and the tag itself that previously provided menus in which the center region of the screen changes depending on the focused menu item. Not only is this a clear reduction in the complexity of the application's resources, but it reduces development time as our staff artist need only provide individual components of an application's main menu, not a multi-frame image consisting of screen images of each possible selected state of the application's main menu,

4.2 Asynchronous Server Interactions

As touched upon in the introduction, a key motivation for this work is the incorporation of asynchronous client-side server interaction with remote Web services. Asynchronous transactions provide users with a more interactive experience and reduce the number of key presses required when performing an action.

Consider an application providing reverse directory lookup (in which you're seeking a name associated with a known phone number). The following XHTML provides a user interface for this application:

```
<html>
<head>
<script language="javascript">
var req;

function callback() {
  div = document.getElementById("result");
  if (req.readyState == 4 &&
      req.status == 200) {
    div.innerHTML = req.responseText;
  }
  else
  {
    div.innerHTML = "network error";
  }
}

function lookup() {
   var field = document.getElementById("number");
   var url = "http://server.com/lookup.php?phone="
    + escape(field.value);

   if ( field.value.length == 10 ) {
     req=new XMLHttpRequest();
     req.open("GET", url, true);
     req.onreadystatechange = callback;
     req.send(null);
  }
}
</script>
</head>
<body>
  <form>
    <b>Phone</b>
    <input type="text"
           value="408"
   id="number"
           onkeyup="lookup();"/>
    <div id="result"/>
</body>
</html>
```

The page has two functions and handles one user event. The function `lookup` creates a query URL with the number you've entered, and if it looks like you've entered a full telephone number, it creates an `XMLHttpRequest` object to use in requesting a server-side lookup of the name for this number. This operation is asynchronous; the `XMLHttpRequest` object will invoke the function `callback` when the request is complete. The `callback` function simply replaces the contents of the named `<div>` tag with either the results of a successful transaction or an error message in the event of a network error. This process—testing the value you enter, issuing a request if it might be a valid phone number by testing its length, and updating the contents a region of the page—is all triggered any time you change the `<input>` field on the page.

In the Scheme-enabled browser, the algorithm is exactly the same but shorter:

```
<html>
<head>
<script language="scheme">
  (define service-url
    "http://server.com/lookup.php?phone=")
  (define input-glyph (document-at "entry"))
  (define result-glyph (document-at "result"))

  (define (lookup)
   (if (eqv? (string-length
     (glyph-value input-glyph)) 10)
     (browser-get
       (string-append service-url
         (glyph-value input-glyph))
       (lambda (succeeded result)
         (if succeeded
          (set!glyph-value result-glyph result)
          (set!glyph-value result-glyph "error"))))))
</script>
</head>
<body>
```

```
  <form>
    <b>Phone</b>
    <input type="text"
      value="408"
      id="entry"
      onkeyup=
        "(lookup)"/>
    <div id="result"/>
</body>
</html>
```

For clarity, this code pre-fetches references to the glyphs corresponding to the two named XHTML tags and saves those references in the `input-glyph` and `result-glyph` variables. The function `lookup` does the same work as its JavaScript counterpart, although is somewhat simpler because the underlying interface to the browser for HTTP transactions is already created and only needs an invocation via the browser foreign function `browser-get`. Like its JavaScript cousin `XMLHTTPRequest`, it operates asynchronously, applying the provided function to the result of the Web request. This browser provides this result as a list with two elements: whether the request succeeded as a boolean in the list's first element, and the text returned by the remote server as the list's second element. In the example, I pass an anonymous function that simply updates the value of the `<div>` tag on the page with the results, just as the JavaScript `callback` function does.

This example not only shows functionality previously impossible to obtain using the browser without scripting support (a developer would likely have implemented a pair of custom tags, one processing the user input and one displaying the results, and written a fair amount of code in C for this specific functionality), but demonstrates the brevity Scheme provides over JavaScript. This brevity is important not just for developers but to limit the amount of time it takes a page to download and render, as well as space used within the browser cache.

5. Results

As the examples show, the resulting port of TinyScheme meets both the objectives of the project and provides a reasonable alternative to JavaScript. Not surprisingly, its limitations are not the language itself but the degree of integration between the script runtime and the browser.

However, TinyScheme provides two key advantages: the time elapsed from the beginning of the port, and overall memory consumption within the application. As previously noted, the small size of TinyScheme made porting a trivial task (less than twenty hours of effort).

While no JavaScript port to the wireless terminal was available for comparison, one basis of comparison is the SpiderMonkey[20] JavaScript implementation within Firefox on Microsoft Windows. Compared against the TinyScheme DLL on Microsoft Windows for the Qualcomm BREW simulator, the results are nearly 2 to 1 in favor of TinyScheme for static footprint.

Implementation	Source Files	Symbols[1]	Size[2]
SpiderMonkey	36	8537	321 KB
TinyScheme	3	605	188 KB

Where:

1. The symbol count was calculated using Source Insight's (www.sourceinsight.com) project report; this count includes all C symbols (functions and variables) in the project.
2. This size indicates the size as compiled as a Windows DLL for the appropriate application target (Firefox or Qualcomm BREW Simulator) in a release configuration.

On the wireless terminal, the TinyScheme interpreter and code required to integrate it with the browser compile to about fifty kilobytes of ARM Thumb assembly; this results in an increase of approximately 50% to the browser's static footprint. This is a sizable penalty, although in practice most application binaries for Qualcomm BREW-enabled handsets are significantly larger these days; at Rocket Mobile engineers are typically concerned more with the footprint of application resources such as images and sounds rather than the code footprint.

The static footprint is a key metric for success because the Qualcomm BREW architecture does not support any memory paging mechanism. As a result, applications must be loaded from the handset's flash file system in their entirety prior to application execution. This means that the static size directly affects the run-time RAM footprint of the browser application as well. On newer handsets this is not an issue—most mid-range handsets sold in the last eighteen months have a half-megabyte or more of RAM for application use, so a fifty kilobyte increase in memory footprint when loading an application is incidental.

In addition to the memory consumed by simply loading the Scheme runtime implementation and related support code into RAM, memory is consumed by the interpreter itself. In practice, once loaded, simply starting the interpreter consumes approximately eight kilobytes of memory; our initial Scheme definitions, derived from TinyScheme's init.scm file and containing the usual definitions for things such as `when`, `unless`, and a handful of type and arithmetic operators, consumes another ten kilobytes. Thus, the startup overhead from first loading the code into memory and then initializing the runtime before doing any useful work is about seventy kilobytes of RAM. While not insignificant on older handsets, this is not a serious impediment on handsets commercially available today; in production, we can tune this file to use only the definitions likely to be used by dynamic scripts.

Run-time performance is well within acceptable bounds for user interface tasks. The user interface example shown in section 4.1 on page 150 takes on the order of sixty milliseconds of time within the Scheme interpreter on a high-end handset (the LG-9800, based on the MSM6550 chipset from Qualcomm) to execute, resulting in no apparent latency when navigating from one link to the next. The asynchronous request example shown in section 4.2 on page 151 is somewhat slower, although the time spent executing the Scheme code is dwarfed by the latency of the cellular network in completing the request.

A final measurement of success, albeit subjective, is developer satisfaction. The general consensus of those developing content for browser-based applications at Rocket Mobile is that at the outset developing applications using scripts in Scheme is no more difficult than doing the same work in JavaScript would be. Because our Web applications are built by engineers proficient in both wireless terminal and server-side work, their strong background in object-oriented and procedural methodologies tend to slow initial adoption of Scheme, and many scripts begin looking rather procedural in nature. Over time, however, contributors have been quick to move to a more functional approach to the problems they face. A key advantage helping this transition is in providing a command-line REPL built with TinyScheme and a small file of stubs that emulate the browser's foreign function interfaces. This lets new developers prototype scripts for pages without the overhead of creating XHTML files on a server, wireless terminal, or wireless terminal emulator, and encourages experimentation with the language.

6. Future Work

Commercialization of this work at Rocket Mobile is ongoing but not yet complete for two reasons. First, to streamline the development cycle, products at Rocket Mobile are typically released as a

single binary for all commercially available handsets at the time of product release; thus the binary must meet the lowest common denominator in both static and run-set size. With several million handsets on the market today having RAM capacities under a half-megabyte, the overhead posed by the interpreter and related code prohibits a widespread release. However, this is not expected to be a significant drawback for new products aimed at recently-marketed mid- and high-tier handsets, which have much greater amounts of RAM available. In the mean time, we are using the technology in a variety of prototype and pilot projects for content providers and carriers with great success.

Second, software quality is of paramount importance to wireless carriers and subscribers. While the TinyScheme implementation has had years of use in some configurations, the underlying port to Qualcomm BREW is far from proven. At present, we are testing and reviewing the implementation of the interpreter with an eye to the types of problems that can cause application failures on wireless terminals, such as ill-use of memory (dereferencing null pointers, doubly freeing heap regions or the like). This work is ongoing, and I intend to release any results of this work to the community of TinyScheme users at large.

Another area of active investigation is to provide better interfaces via the FFI to the browser's list of glyphs and individual glyph attributes. In its present form, the `glyph-src` and `glyph-value` functions and their `set!` counterparts are workable, but somewhat clumsy. Worse, as the browser exports an increasing number of glyph attributes (such as color, size, and position), the current approach will suffer from bloating due to the number of foreign functions required to access and mutate individual glyph attributes.

An obvious future direction for this work is for the implementation to include support for SXML. While the present implementation of `browser-get` does nothing with the results returned from an HTTP transaction but pass that content on as an argument to an evaluated S-expression, client applications wishing to interact with Web services via XML-RPC or SOAP would benefit from having a parsed representation of the request results. SXML provides an ideal medium for this, because it provides easy mechanisms for querying the document tree or transforming the tree[21] in various ways to provide human-readable output which can then be set as the contents of an XHTML `<div>` region. Using this approach, a front-end to a Web service can be built entirely using the browser application and client-side scripts in Scheme that collect and process user input, submit queries to the remote Web service and present results without the need for an intermediary server to transform requests and responses from XML to XHTML.

Acknowledgments

I would like to thank Shane Conder and Charles Stearns at Rocket Mobile for their initial vetting of my approach to introducing a Scheme runtime to the browser and their support throughout the project. Notably, Charles performed much of the work on RocketBrowser required to integrate the TinyScheme engine.

References

[1] Steven Pemberton, et al. editors. *XHTML 1.0: The Extensible HyperText Markup Language.* W3C Recommendation 2002/REC-xhtml1-20020801. `http://www.w3.org/TR/2002/REC-xhtml1-20020801`. 1 August 2002.

[2] JavaScript Mozilla Foundation, Mountain View, CA, USA. `http://www.mozilla.org/js/`. 2006

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau, editors. *Extensible Markup Language (XML) 1.0.* W3 Recommendation 2004/REC-xml-20040204. `http://www.w3.org/TR/2004/REC-xml-20040204`. 4 February 2004.

[4] Google. *Google Maps API.* `http://www.google.com/apis/maps/`. 2006.

[5] Qualcomm, Inc. *BREW API Reference*. `http://www.qualcomm.com/brew/`. 2005.

[6] Open Mobile Alliance. *XHTML Mobile Profile.* `http://www.openmobilealliance.org/`. 2004.

[7] ECMA International. *ECMAScript Language Specification.* `www.ecma-international.org/publications/standards/-Ecma-262.htm`. 1999.

[8] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices, 33(9):2676, September 1998.*

[9] Arnaud Le Hors, et al, editors. *Document Object Model (DOM) Level 3 Core Specification.* W3 Recommendation 2004/REC-DOM-Level-3-Core-20040407. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407`. 7 April 2004.

[10] Jesse James Garrett. *Ajax: A New Approach to Web Applications.* `http://adaptivepath.com/publications/essays/archives/000385.php`. 18 February 2005.

[11] Anne van Kesteren and Dean Jackson, editors. *The XMLHttpRequest Object.* W3 Working Draft 2006/WD-XMLHttpRequest-20060405/ `http://www.w3.org/TR/XMLHttpRequest/`. 5 April 2006.

[12] Microsoft, Inc. IXMLHTTPRequest. *MSDN Library* `http://msdn.microsoft.com/library/default.asp?-url=-/library/en-us/xmlsdk/html/63409298--0516-437d-b5af-68368157eae3.asp`. 2006.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616, Hypertext Transfer Protocol—HTTP/1.1* `http://www.w3.org/Protocols/rfc2616/rfc2616.html` The Internet Society, 1999.

[14] Todd.Gillespie and Rahul Nair. mod-scheme. `http://blufox.batcave.net/mod_scheme.html`. 2005.

[15] P. T. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. *European Symposium on Programming, pages 122-136, Apr. 2001.*

[16] Oleg Kiselyov. SXML Specification. *ACM SIGPLAN Notices, v.37(6), pp. 52-58, June 2002.*

[17] Dave Winer. *XML-RPC Specification.* `http://www.xmlrpc.com/spec`. 15 Jul 1999.

[18] Don Box, et al. *Simple Object Access Protocol (SOAP) 1.1* W3C Note 2000 NOTE-SOAP-20000508. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`. 8 May 2005.

[19] Dimitrios Souflis and Jonathan S. Shapiro. TinyScheme. `http://tinyscheme.sourceforge.net/`. 2005.

[20] Mozilla Foundation. SpiderMonkey (JavaScript-C) Engine. `https://www.mozilla.org/js/spidermonkey/`. 2006.

[21] Oleg Kiselyov. XML and Scheme (a micro-talk presentation). *Workshop on Scheme and Functional Programming 2000* `http://okmij.org/ftp/Scheme/SXML-short-paper.html`. September 17, 2000

[22] Ward Willats. How to Compile BREW Applets with WinARM 4.1.0. `http://brew.wardco.com/index.html`. May 11, 2006

[23] Free Software Foundation, Inc. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/` August 1, 2006.

[24] Microsoft, Inc. COM: Component Object Model Technologies `http://www.microsoft.com/com/default.mspx` 2006.

A. Porting TinyScheme to Qualcomm BREW

While software development for wireless terminals is not nearly the constrained affair it was a scant five years ago, platforms such as Qualcomm BREW still impose many restrictions on the kinds of things that can be done in C and C++. While this does not significantly impair the development of all-new code for a mobile platform, it can make porting applications from a more traditional computing environment somewhat challenging.

In the case of the TinyScheme port, the single largest impediment was the lack of a read-write segment on the Qualcomm BREW platform.[4] Without a read-write segment, the target compiler (ARM's ADS 1.2) cannot compile applications or libraries with global variables. This makes using the standard C library impossible. Instead, applications must use a limited set of helper functions provided by Qualcomm to perform operations typically provided by the C standard and math libraries. Thus, references to C functions such as `strcmp` must reconciled with Qualcomm BREW helpers such as `STRCMP`. Rather than making numerous changes to the TinyScheme implementation (which would make merging changes from later releases difficult), I implemented a small porting layer with functions for each of the required C standard library functions. In many cases, this work was as simple as writing functions such as:

```
static __inline int
strcmp(const char *a, const char *b) {
  return STRCMP(a, b);
}
```

where `STRCMP` is the Qualcomm BREW API providing the C standard library `strcmp` facility. In addition, in a few cases such as the interpreter's use of the standard C file I/O library I had to implement the interfaces atop Qualcomm BREW's native file management functions. The resulting porting layer is approximately 300 lines of code (including comments) and can be reused when porting other code requiring the standard library to Qualcomm BREW.

Dealing with floating-point numbers without a traditional math library involved similar work; in addition to providing porting functions for numeric functions such as trigonometric operations, I also needed to deal with places where the interpreter used arithmetic operators on floating-point numbers to keep the tool chain from attempting to include the default floating-point library. When writing new code, Qualcomm suggests that developers use their helper functions for arithmetic; these provide functions for addition, subtraction, multiplication, and division. Unwilling to make such drastic changes to the TinyScheme implementation, I chose a second route. To facilitate porting, Qualcomm has made available a replacement floating-point library for basic arithmetic that does not use a read-write segment; including this library incurs an additional static use of eight kilobytes of memory. If needed, I can back this out and rewrite the functions that use floating-point arithmetic to use the Qualcomm BREW helper functions to reduce the overall footprint of the interpreter.

Along the same vein, the TinyScheme interpreter had a few global variables that I had to move to the interpreter's context `struct scheme`; these were for things like the definition of zero and one as native numbers in the interpreter's representation of integer and floating-point numbers. Similar work was required for the array of type tests and a few other global variables. More problematic, however, was the table of opcodes, defined at compile time using a series of preprocessor macros to initialize a large global array of opcodes. Although this table is constant (and can thus be loaded into the read-only segment), it results in compilation errors for another reason: Qualcomm BREW requires relocatable code, and the tool chain doesn't know what to do with function pointer references in constant variables. By moving the opcode dispatch table into the interpreter's context `struct scheme`, and adding the following snippet to the interpreter's initialization function `scheme_init`:

```
#define _OP_DEF(A,B,C,D,E,OP) \
  sc->dispatch_table[j].func = A; \
  sc->dispatch_table[j].name = B; \
  sc->dispatch_table[j].min_arity = C; \
  sc->dispatch_table[j].max_arity = D; \
  sc->dispatch_table[j].arg_tests_encoding = E; \
  j++;
{
  int j = 0;
  #include "opdefines.h"
  #undef _OP_DEF
};
```

As each Scheme opcode is defined in the opdefines.h header using the `_OP_DEF` macro, this yielded an easy solution with a minimum of changes to the existing implementation.

With this work complete, the interpreter itself was able to compile and execute on Qualcomm BREW-enabled handsets, although its packaging was less than ideal. The Qualcomm BREW platform is built around a component-oriented model similar in many ways to the original Microsoft Windows Component Object Model; packaging the Scheme interpreter as a module wrapped in a BREW interface would provide greater opportunity for reuse throughout my employer's software development efforts. The resulting module, called an extension in Qualcomm parlance, actually offers three interfaces:

- The `ISchemeInterpreter` interface exports the basic interface to the interpreter permitting developers to create an instance of the interpreter, set an output port and have it evaluate S-expressions.
- The `ISchemeFFI` interface exports the foreign function interface provided by TinyScheme in its `struct scheme_interface` structure, permitting developers familiar with Qualcomm BREW an easy way to implement foreign functions for the interpreter without needing to see or access the implementation of the interpreter itself.
- The `ISchemeFFP` interface is an abstract interface that developers implement when creating foreign functions for use with the interpreter. This interface uses the methods provided by the `ISchemeFFI` interface of a running interpreter instance to implement foreign functions. The foreign functions provided by RocketBrowser are implemented as a BREW extension implementing the `ISchemeFFP` interface.

To facilitate the `ISchemeFFI` and `ISchemeFFP` interfaces, I extended TinyScheme to support references to C heap objects as opaque Scheme cell contents through a new TinyScheme type, `foreign_data`. A cell containing a reference to a C heap object looks like this:

```
typedef struct foreign_data {
  void *p;
  uint32 clsid;
} foreign_data;

// and inside of the cell structure union:
```

[4] Tools such as WinARM[22], based on GCC[23], have recently become available that will generate "fix-up" code that creates read-write variables on the heap at runtime, although they generate additional code. In addition, Qualcomm has announced support for a toolchain for use with ARM's ADS 1.2 that does not have this limitation, but this tool was not available as I performed this work.

```
  struct {
    foreign_data *data;
    foreign_func cleanup;
  } _fd;
```

Thus, the `foreign_data` structure contains a pointer to the C heap region it references, and an unsigned double word that can contain type information, such as the native class identifier of the object being referenced. Within the cell of a foreign data object is also a pointer to a foreign function the garbage collector invokes when freeing the cell. This allows foreign functions to create instances of C heap objects for use with other foreign functions without the need for explicit creation and destruction by the application developer. This new type is supported in the same manner as other TinyScheme types, with functions available for creating instances of this type as well as testing a cell to see if it contains an instance of this type. The `display` primitive is also extended to display these objects in a manner similar to the display of foreign functions.

The type data that the `foreign_data` type carries permits developers to provide type-safe cast functions when accessing C data in foreign function interfaces. For example:

```
// Return the class of the foreign_data
static __inline AEECLSID
ISCHEMEFFITYPE_GetClass(foreign_data *p) {
  return p ? p->clsid : 0;
}

// return whether this foreign_data is
// of the desired class
static __inline boolean
ISCHEMEFFITYPE_IsInstanceOf(foreign_data *p,
                            AEECLSID cls) {
  return (boolean)(p && p->clsid == cls);
}

// Return a typeless pointer to the data
// contained by a foreign_data
static __inline void *
ISCHEMEFFITYPE_GetData(foreign_data *p) {
  return p ? p->p : NULL;
}

static __inline IShell *
ISCHEMEFFPTYPE_GetShell(foreign_data *p) {
  return
   ISCHEMEFFITYPE_IsInstanceOf(p, AEECLSID_SHELL) ?
   (IShell *)ISCHEMEFFITYPE_GetData(p) : NULL;
}
```

Using inline functions such as `ISCHEMEFFITYPE_GetShell` to ensure type-safe access to `foreign_data` wrapped data at the C layer is unfortunately a manual approach. Because developers can at any time circumvent this type-safety by accessing the contents of a `foreign_data` item directly, it must be enforced by convention and inspection.

The `foreign_data` type is also used with foreign functions in this implementation. When defining foreign functions with the interpreter, the developer can also register a `foreign_data` object and its associated destructor. This lets foreign functions keep state without needing to directly modify the TinyScheme context `struct scheme`.

Wireless terminals typically provide a watchdog timer, so that a rogue application cannot lock the handset indefinitely and prevent its use as a telephone. If application execution continues until the watchdog timer fires (typically two seconds), the handset reboots, terminating the rogue application. To avoid erroneous script errors from triggering this timer and resetting the handset, I add a similar timing mechanism to `Eval_Cycle`, as well as a function `scheme_set_eval_max` to let TinyScheme users set the watchdog timer's maximum value. If a script runs for longer than the maximum permitted time, execution is aborted and the runtime user notified with an error indicating that the script could not be completed and the runtime's state is indeterminate. The RocketBrowser application sets the maximum script execution time at 500 ms, giving ample time for simple UI operations and a more-than-adequate ceiling for remaining browser operations during page layout and drawing.

With all of this in place, extensions to the TinyScheme interpreter could be written as stand-alone BREW extensions implementing the `ISchemeFFP` interface, making dynamic loading of extensions to the TinyScheme runtime possible.