SUPPORTING SCRIPTING LANGUAGES ON TI ARCHITECTURES

Mark Grosen[1], Timothy J. Harvey[2], and Todd Waterman[2]

[1]Santa Barbara, CA and [2]Sugar Land, TX, USA

[mgrosen|t-harvey|twaterman]@ti.com

***Abstract* – Scripting languages are popular programming tools for prototyping and application development. These languages make programming easier with simplified syntax, constrained type systems, and high-level abstractions that remove many low-level considerations from the user.**

The goal of the "scripting project" is to increase the user base of TI's architectures by making them easier to program. For professionals, we aim to make prototyping easier; for more casual users, we want to make the out-of-box experience simple and intuitive.

We have two groups we need to support: first, the end-users ("scripters"), who will insist on writing programs in the scripting language of their choice, and, second, the owners of the underlying libraries, SDKs, *etc.* ("maintainers"), who will insist on minimal rewriting of existing, tested code (traditionally written in C).

We have developed an alpha implementation of a system that accomplishes both of these goals for Javascript in small-memory environments.

***Keywords*: Javascript, Jerryscript, WebIDL**

INTRODUCTION

Scripting languages such as Ruby, Lua, Javascript, *etc.* make programming easier by hiding or abstracting away low-level details such as memory management, type systems, and management of architectural resources. An increasing number of users are adopting one of these languages, and they would like to take their experience and use it on our architectures. Further, IOT applications are a natural use of scripting languages, as most of these languages seamlessly support IOT's event-based paradigm.

A number of questions arises when considering how to best enter this space. First, the proliferation of scripting languages has made programming easier, but there is no clear "best" language: every language has its own community of devotees, its own strengths and weaknesses, *etc.* – so how do we choose a scripting language to support?

Instead of tying ourselves down to a single choice and letting slip the rest of the market, we need to consider how we can leverage the greatest number of languages with the least amount of effort. From the scripters' view, we need a way to describe behavior that is language independent. From the maintainers' view, we need a write-once-use-everywhere approach.

Along these lines, we are targeting smaller-memory architectures such as the SimpleLink MCUs. Larger processors and memories already have at least some support for scripting: any of these scripting environments that runs in Linux, for example, already works on our larger parts. The more resource-constrained parts, however, suffer from both the ease-of-use problem and from limited system resources – many applications run on "bare metal," without access to the kinds of services a larger operating system can provide, notably dynamic memory allocation, among many other things.

The model used by scripting languages includes a language interpreter that sits on top of – and communicates with -- the native operating system. For constrained-resource devices, another of the questions we have to address is the feasibility of running the interpreter in such an environment. Fortunately, many interpreters for many scripting languages target limited-resource environments.[1] The interpreters that we are interested in not only have small footprints, but also

[1] One of the use cases for scripting languages is including scripts (and the interpreter needed to evaluate the script) in another executable (such a C program); as a result, the implementers tend to write interpreters with extremely small-footprints. Although this paper focuses on running the interpreter as a standalone program, everything we do is amenable to the shared-executable use case, as well.

provide a critical factor: a mechanism to attach our library to the interpreter, so that when a script mentions one of our library functions, the interpreter knows to invoke our code.

Another question related to the problem of supporting scripting is: how do we provide a single interface that will be natural for users of any scripting language? If we were to use one of the scripting languages itself as our model, that view (the syntax, types, *etc.*) might not be generic enough or translate correctly to another language. Instead, we will use an *interface-definition language*, or IDL. IDLs specify APIs, the types used, calling conventions, *etc.*, but do not actually implement real code. If we use an IDL with a generic syntax and relatively simple type system, scripters should be able to use our libraries, regardless of the language they are writing in.

With this list of concerns, what, then, does our implementation need to include? We have designed our initial implementation to be extensible to many scripting languages with little or no extra (*i.e.*, past some initial setup) work on the part of the maintainer. Our model is: 1) use extensible, open-source interpreters to support the scripters; 2) use an intermediate language to define APIs that both the scripters and maintainers utilize, and 3) build a system to automatically generate "glue" code to marry library code to the interpreters while at the same time hiding these details from the maintainers.

In the next section, we will briefly describe our initial support of Javascript using the Jerryscript interpreter. This will lay the foundation for the rest of the discussion of our implementation, where we will show what our code looks like and how it can be used in multiple environments.

CHOOSING THE INITIAL IMPLEMENTATION

The open-source world has a legion of interpreters covering a multitude of different languages. We narrowed the possibilities by focusing on interpreters:

1) for languages with larger user bases;
2) with small memory footprints;
3) with active development communities.

We settled on the Jerryscript project because it supports Javascript (useful for IOT applications), runs in very little memory (<32K of RAM), and is supported by both professionals (Samsung is the driving force behind Jerryscript) and the amateur community.

The Jerryscript interpreter is written entirely in C. It does its own memory management, including garbage collection, which is important in keeping the memory footprint small. Critically, Jerryscript includes a relatively robust API that lets us add language extensions (think: libraries) to the interpreter that can then be accessed from the Javascript side.

For example, the Jerryscript API includes "register_function", which is a procedure that links its two parameters: the name of the Javascript command and the C function that will be linked in to the interpreter. If we write:

```
jerry_register_function("foo", foo_handler)
```

...the scripter can then use "foo" as a Javascript command, and when Jerryscript sees that command, it will call our function, "foo_handler." The maintainer is responsible for writing the C code "foo_handler" and compiling/linking it into Jerryscript.

This leads to a natural question: how do the scripter and maintainer agree on function names, parameters, *etc.*?

WEBIDL

Remember that IDLs are specification languages that define APIs and are designed to provide communication between dissimilar systems (which could include different hardware, different languages, or both).

The WebIDL language (W3C, 2016) has a simple, C-like syntax and was designed to describe communication between webpages, which dovetails nicely into our desire to support IOT applications. Further, it has a simple static type system that lines up neatly with Jerryscript's type system.

In contrast, Javascript is a dynamically typed language. As we show, below, our system hides Jerryscript's handling of runtime values from the maintainer. Jerryscript has a number of query functions in its API that allow C code – which is statically typed – to probe a value's type before assigning it.

The small set of simple types in WebIDL are easily understood by scripters, while having a limited, static set of types simplifies the C implementation.

WebIDL has three main features: interfaces, operations, and dictionaries. Both interfaces and dictionaries map to objects in Javascript and structures in C. Operations are methods in Javascript and functions in C.

Putting it all together

We have targeted the WebIDL from Lenaro's Zephyr project as our first test. As an example of what our code looks like, we can examine part of the WebIDL for the Zephyr GPIO implementation:

```
interface GPIO {

    GPIOPin open(GPIOInit init);   };
```

...this code is intended to describe a GPIO object with a single method, `open`, that takes one parameter and returns another object. (We have elided many details for ease of exposition, but this is actual Zephyr WebIDL.)

The scripter, having consulted the WebIDL, might write some Javascript that looks like this:

```
var gpio = new GPIO;

var gpio_pin = gpio.open({...});
```

...and so on. On the C side, the maintainer would need to write:

```
register_function("GPIO",&create_GPIO);

add_field_to_object("GPIO",

                    "open",

                    &open_handler)
```

...the first call to register_function links a C function called `create_GPIO` to the call to "`new GPIO`" in Javascript; likewise, the second procedure call links a C function named `open_handler` to the Javascript `open` call, while at the same time adding "open" as a method call to the `GPIO` object. Each of the C functions has a signature of four parameters: the function name, the contextual object (in Javascript, the scope of a function depends on when and where it is invoked), and the list of parameters coupled with the count of those parameters. The C implementation of each function starts out by taking this list of parameters, checking their types, and assigning them to local (C) variables. Any return value from the function has to undergo translation in the opposite direction: from C type to Jerryscript type. At the top of the next page is what the body of our example "open_handler" would look like.

```
jerry_val open_handler(jerry_val function,

                           jerry_val this_pointer,

                           jerry_val argv,

                           int argc)

{

   GPIOInit init = jerry_get_object(argv[1]);



     /* do something */



   GPIOPin result = {...};

   return (jerry_val) result;

  }/* open_handler */
```

...many of the details have been simplified, but what the above code shows is that, except for the "do something" part of the function, all of the C code can be automatically generated.

Our system does just this: it parses the WebIDL and creates all of the marshaling and demarshaling code. Most importantly, the system also creates "outlined" (as opposed to "inlined") functions for the "do something" body of each function. The outlined function for our example would look like:

```
GPIOPin GPIO_open(GPIOInit init)

{
    /* do something */
} /* GPIO_open */
```

...and the maintainer would fill in the body naturally. Notice that this code follows directly from the WebIDL and is interpreter agnostic; it contains none of the code necessary to convert to/from Jerryscript types. Indeed, because it doesn't know anything about Javascript types, once the maintainer fills in this body, we would be able to use it for *any* language and *any* interpreter. That is, we could develop a system that would read in the same WebIDL and produce the necessary (de)marshaling for a Python interpreter, for example, plugging in the maintainer's code that he has already written.

Structure of the Generator

In this section, we will briefly describe the structure of the "generator" -- the tool we have built to support Jerryscript.

Remember that one of the design goals of this project was to utilize, as much as possible, open-source software. To that end, we discovered the code (Eltuhamy) based on Mohamed Eltuhamy's masters thesis (Eltuhamy, 2014). In his work, Eltuhamy was automatically generating Javascript and C++ stubs from WebIDL.[2]

This generator is split into three phases. The first utilizes the WebIDL parser (w3c) that seems to be the industry standard.

After parsing the input WebIDL into a simple abstract-syntax tree (AST), the generator makes a pass over the AST, annotating and summarizing subtrees of the AST to make it amenable to the third pass.

The output pass uses the Mustache compiler (mus) to parse the AST and produce the files our generator needs: for each interface, we create a pair of .c and .h files that form the glue code and a .c file to hold the stubs; we also produce a single .h and .c file that represents all of the definitions and assorted types included by the interfaces.

Related work

Most of the code that the generator produces is aimed at fulfilling the *foreign function interface*, or FFI, for Jerryscript. FFI is a general term that describes the method of connecting two languages to allow calls back and forth between them. The very fact that there is a precise term for this concept implies that we are not the first to attempt this.

A natural question is: why don't we use an already existing system that does something similar to what we need?

We certainly tried – but what we learned is a painful lesson in using open-source software: while centralized stores of free software like GitHub have an abundance of tools/projects/*etc.*, unless the code is *very* close to the requirements, it is probably not going to turn out to be very useful.[3] Further, evaluating each of the sheer number of projects that *might* be relevant is extremely time-consuming.

With that in mind, we looked at projects like SWIG (SWI), which purports to do largely what we set out to do; we were immediately stymied by their license, which is the anathema (to TI) GPL. Other similar projects like emscripten (Zakai) are close to what we need, but are designed in the wrong direction: emscripten, for example, enables C code to use Javascript libraries, instead of the other way around.

Other languages such as Lisp (Oliveira), Haskell (Chakravarty), and Java (Sun Developer Network), provide native FFIs, and there are a number of open-source language-to-language FFIs like libffi (lib) and libffcall (Free Software Foundation), but these do not address our problem: we rely on interpreters to handle the scripting languages' FFIs; we are concerned with the FFIs of the interpreters themselves.

Conclusion

We have developed an open-source system that takes a WebIDL specification and automatically generates most of the code necessary to enable a Javascript interpreter called Jerryscript to utilize existing libraries coded in C. We carefully designed the interface so that the human-generated code can be reused by other interpreters and scripting languages.

[2] We completely altered much of the middle pass and all of the final pass of Eltuhamy's implementation to make it produce the output we need – the overall design remains the same between his code and ours, so we present it as he wrote it.

[3] By "useful", we mean that the amount of effort required to reshape the project into something resembling another set of requirements may be equal to or more than the effort of just starting from scratch.

References

Chakravarty, Manuel M. T. The Haskell 98 Foreign Function Interface 1.0: an Addendum to the Haskell 98 Report. [Online] http://www.cse.unsw.edu.au/~chak/haskell/ffi/.

Eltuhamy, Mohamed. 2014. *Native Calls: JavaScript - Native Client Remote Procedure Calls.* Department of Computing, Imperial College London. London : Imperial College, 2014. p. 112, Masters Thesis.

Eltumany, Mohamed. native-calls. *GitHub.* [Online] https://github.com/meltuhamy/native-calls.

Free Software Foundation. GNU libffcall. [Online] https://www.gnu.org/software/libffcall/.

libffi. *GitHub.* [Online] https://github.com/libffi/libffi.

mustache/mustache. *GitHub.* [Online] https://github.com/mustache/mustache.

Oliveira, James Bielman and Luis. CFFI -- The Common Foreign Function Interface. [Online] https://common-lisp.net/project/cffi/.

Sun Developer Network. Java Native Interface: Programmers Guide and Specification. [Online] https://web.archive.org/web/20120728074805/http://java.sun.com/docs/books/jni/.

SWIG. *GitHub.* [Online] https://github.com/swig/swig.

W3C. 2016. WebIDL Level 1, W3c Recommendation 15 December 2016. [Online] December 15, 2016. https://www.w3.org/TR/WebIDL-1/.

w3c/webidl2.js. *GitHub.* [Online] https://www.w3.org/TR/WebIDL-1/.

Zakai, Alon. emscripten. *Github.* [Online] https://github.com/kripken/emscripten.