Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Study and Development of Software Integration for Embedded Microprocessors

Relatore:
Prof. Luca Sterpone

Relatore aziendale:
Giovanni Di Sirio

Candidato:
Edoardo Mazzella
matricola: 230560

Dicembre 2017

Study and Development of Software Integration for Embedded Microprocessors
Integration of a JavaScript Engine on STM32
EDOARDO MAZZELLA
Computer Engineering
Politecnico di Torino

Abstract

Nowadays more and more powerful microcontrollers are available on the market at extremely low prices and therefore many experts are questioning the future of C language in the embedded field in favor of higher level languages.

JavaScript, with its huge community, is proposed as a candidate to be one of the predominant languages in this field, thanks to its accessibility, fast development times and its syntax similar to C. For these reasons in the last years many embeddable JavaScript engines have been developed in order to execute JS code from C and vice versa.

Taking into account this trend, Giovanni Di Sirio, my tutor during the the internship period at STMicroelectronics in Arzano (NA), gave me the job of integrating Duktape JavaScript engine in ChibiOS, an RTOS developed by Giovanni himself. The assigned task, object of this thesis document, has the goal of exporting ChibiOS device drivers and functions to JavaScript in order to develop hybrid C/JS applications for embedded microprocessors exploiting the advantages of both languages.

Being a free license JavaScript engine with a focus on portability and compact footprint, Duktape suits perfectly to be integrated in embedded devices. By including Duktape library inside a C/C++ program it is possible to use its C API calls in order to create JavaScript threads of execution, also called Duktape contexts. Furthermore the Duktape API allows to export to JavaScript C types, structures and functions, populating the heap of a context with the corresponding JS objects.

ChibiOS is an open-source development enviroment for embedded applications. Its design methodology makes ChibiOS performing, small and easily portable. In fact the Hardware Abstraction Layer, containing the ChibiOS device drivers, provides the same interface for different hardware specific function implementations and furthermore an Operating System Abstraction Layer makes the HAL completely RTOS aware.

Scriptly is the name of the developed ChibiOS layer that integrates Duktape with the purpose of extending embedded C application development through scripting. Using the Duktape API, several ChibiOS device drivers and functions are exported to JavaScript in order to exploit the features of the scripting language, from the object oriented paradigm to the possibility of updating code at run-time. JavaScript threads of execution can be easily initiliazed through a simple C function call, populating their heaps with JS objects which abstract the main microcontroller peripherals.

In particular the ChibiOS Port Abstraction Layer, the events subsystem, the Serial and PWM device drivers, as well as functions to handle the native thread status, are exposed to JavaScript, providing JS objects whose methods are associated to the corresponding C functions. Arguments passed to JS functions are always checked and errors are handled providing an error code and message to communicate outside what is gone wrong during Duktape execution. Furthermore a configuration file is provided in order to minimize the code size, permitting to enable and disable pieces of code, and to avoid useless JS object allocations.

In conclusion Scriptly is a C library completely based on open-source components and optimized for low memory embedded microprocessors. After initializing a context, a JavaScript API is made available permitting developers to access microcontroller peripherals either at run-time, through the Scriptly Shell, or evaluating strings from C code. It's usage is very simple and permits to easily develop hybrid C/JavaScript applications according to the developer's choiches. Furthermore its own and the underlying HAL architecture make Scriptly easily portable on different RTOSs and platforms.

Considering performance, the interpreted nature of JavaScript code increases execution time. Furthermore the Garbage Collector, while eliminating the need of explicitly freeing memory, causes undeterministic pauses in the application impacting performance and reducing predictability. For this reasons C code must always be preferred for tasks with strong Real-Time constraints while JavaScript accessibility and fast development times can be exploited for less critical sections.

Analyzing memory usage, while Scriptly code size is configurable, at least 70kB of RAM are necessary to create and initialize a Duktape heap, which is a reasonable size considering the mordern microcontrollers features.

Looking at the future, Scriptly is also a first step to permit visual programming on microcontrollers. A possible challange could be the development of a Google Blockly Application in order to generate Scriptly compliant JS code, allowing less experienced programmers to develop applications for embedded microprocessors just moving and interlocking custom visual blocks, providing a simplified programmation approach.

Acknowledgements

Ringrazio la mia famiglia ed i miei amici, che mi hanno sempre sostenuto durante questo percorso universitario.

Inoltre un ringraziamento speciale va a Giovanni Di Sirio e Rocco Marco Guglielmi che hanno contribuito con le loro conoscenze allo sviluppo di questo elaborato ed alla mia formazione professionale.

Edoardo Mazzella, Napoli, Dicembre 2017

Contents

List of Figures

1
Introduction

1.1 Background

Nowadays the embedded systems field is rapidly evolving and the availability of microcontrollers offering good performances at low prices is questioning the future of C language in favor of higher level languages. Many languages are proposed as candidates to substitute or juxtapose C for programming microcontrollers but still none of these succedeed in carving out a dominant role.

Peter Hoddie, founder of Kinoma, a division of Marvell Semiconductor, has declared: "Computing power has become so inexpensive, and the JavaScript developer community so large, that the accessibility and fast development times of JavaScript will outweigh the efficiency advantages of C and assembly for all but fairly specialized projects." [1]. Supporting this thesis, many other experts publicize the simplicity of JavaScript and the huge availability of libraries developed by its community.

On the other hand there are those who believe that JavaScript is not suitable for programming microcontrollers and cannot be used for professional solutions because of the impossibility of interpreted code to guarantee deterministic response times and since memory management is quite critical in systems with little memory [2].

However the usage of JavaScript on microcontrollers is an important trend in the last years and many embeddable JavaScript engines have been developed in order to execute JS code from C and vice versa.

1.2 Scope

Taking into account this trend and these schools of thought, Giovanni Di Sirio, my tutor during the the internship period at STMicroelectronics in Arzano (NA), gave me the job of integrating Duktape JavaScript engine in ChibiOS, an RTOS developed by Giovanni himself. The assigned task, object of this thesis document, has the goal of exporting the ChibiOS device drivers and functions to JavaScript in order to develop hybrid C/JavaScript applications for embedded microprocessors exploiting the advantages of both languages.

Today different solutions to program in JavaScript on microcontrollers have already been proposed but the attempt of this research is to provide something that could be suitable with real-time systems and efficiently used in low-memory enviroments.

1.3 Outline

This report is composed of three main chapters:

- **Theory**: all the necessary concepts for integrating Duktape in ChibiOS are presented here. It is divided in three main sections regarding JavaScript, Duktape and ChibiOS.

- **Methods**: it is an explanation of the methods and the strategies applied to develop Scriptly, the new ChibiOS layer which makes possible to export the RTOS device drivers and functions to JavaScript.

- **Results**: the results of my research are shown in this chapter, critically analyzing and discussing them.

2 Theory

This chapter is dedicated to all the thoretical perspectives useful to understand the methods to develop Scriptly, the new ChibiOS layer extending the functionalities of the RTOS through scripting. It is divided in three sections presenting the main concepts of JavaScript, Duktape and ChibiOS.

2.1 JavaScript

JavaScript, often abbreviated as JS, is a high-level, dynamic, weakly typed, object-based and multi-paradigm programming language. Initially only implemented client-side in web browsers, JavaScript engines are now embedded in many other types of host software [3]. Its application in the embedded field is supported by many experts which publicize its accessibility and fast development times.

2.1.1 Features

In this section some JavaScript features, useful to understand the development of Scriptly, are briefly summarized.

2.1.1.1 Dynamic

Javascript dynamicity can be summarized in two concepts:

- **Dynamic Types**
  In JavaScript a type is associated with values instead of variables. For example the same variable could change its type from a number to a string during the execution of a program [3].

- **Run-time evaluation**
  JavaScript provides a function to dynamically evaluate strings [3]. This feature permits to easily execute and modify code at run-time, without complex programming strategies.

2.1.1.2 Object-Oriented and prototype-based

JavaScript can be considered almost completely object based [3]. In JavaScript all that is not a primitive type is an object, including JS functions. Each object have its own properties and methods and a prototype which can be dynamically modified.

Prototypes

All the JavaScript objects have a prototype object as property. Prototypes permits a particular inheritance chain mechanism in which each object inherits methods and properties of its prototype. By changing a prototype during the program execution it is possible to modify object properties at run-time [3].

Functions as object constructors

Function objects are used as constructor in JavaScript. By calling a function with the **new** prefix, it is possible to create an istance of the prototype property of the function object [3].

Functions as methods

In JavaScript functions and methods are defined in the same way. The distinction between the two occurs when they are called: a function called as method have the **this** keyword binded to the object which have that function as property [3].

2.1.1.3 Interpreted

JavaScript code is usually interpreted. Differently from compilers, interpreters translates and executes the program instruction by instruction instead of analyzing the whole code and compiling it.

2.1.1.4 Automatic memory management

Differently from other languages like C and C++, in JavaScript memory allocation and de-allocation is automatically handled by the Garbage Collector. Through a reference counting process or a mark-and-sweep algorithm, de-allocation is performed only when an object is not referenced anymore by other objects. This approach allows developers to unconcern about memory managment.

2.2 Duktape

Duktape is an embeddable JavaScript engine with a focus on portability and compact footprint [4]. Integrating Duktape it is possible to extend the functionalities of C/C++ application through scripting [4]. Its API in fact permits to execute C code from JavaScript and viceversa.

2.2.1 Programming model

This section is dedicated to the Duktape programming model, presenting the fundamental concepts in order to integrate the JavaScript engine inside a C/C++ application [4]. The development of Scriptly is completely based on the theoretical concepts presented in this section.

2.2.1.1 Heap and context

The Duktape API permits to create JavaScript threads of execution, also called Duktape contexts, inside your program. Contexts, represented in the Duktape API by a duk_context *, reside in a certain heap and are associated with a global enviroment [4].

An **heap** is a region where JavaScript variables and objects can be allocated. Inside it memory management is handled by a **garbage collector** which automatically de-allocates memory regions when objects are not referenced anymore. This mechanism is possible thanks to an object header which keeps informations about reference counting, mark-and-sweep garbage collection and object finalization. While objects residing in the same heap can reference each other, when they reside in different heaps serialization is the only way to pass values between heaps [4].

A Duktape context is also associated to a coroutine having a value and call stack. The **value stack** can be accessed using the provided API, inserting values, calling functions etc. The **call stack** keeps track of JavaScript and native function calls within Duktape. Basing on the current function call, the value stack presents an active **stack frame** containing the passed arguments and with which the active function can interact and insert a return value. It is also present a **catch stack**, which is not visible to the caller, to track error catching sites established using for example try-catch-finally blocks [4].

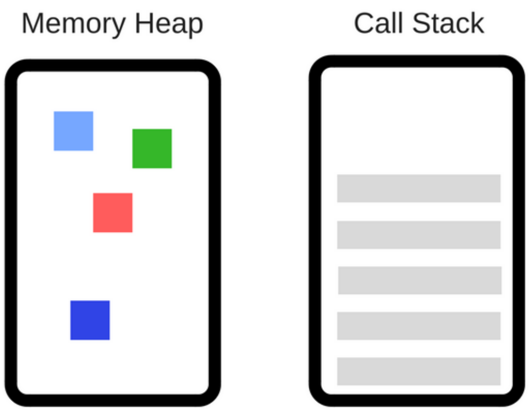


Figure 2.1: Memory heap and call stack representation.

Different contexts can share the same heap, sharing the same garbage collection state, or reside in different ones. Obviously in the last case it is not possible to directly exchange object references and serialization must be used.

In Duktape it is possible to run multiple indipendent heaps and contexts without restriction but only one native thread can execute any code within a single heap at any time [4].

In order to create a new Duktape heap and an initial context residing in it, the following code must be used [4]:

```
duk_context *ctx = duk_create_heap_default();
if (!ctx) { exit(1); }
```

Custom memory allocation functions and a fatal error handler can also be provided using the following API call [4]:

```
duk_context *ctx = duk_create_heap(my_alloc, my_realloc,
   my_free, my_udata, my_fatal);
if (!ctx) { exit(1); }
```

To create a new context sharing the same heap and global enviroment, the following instructions must be used [4]:

```
duk_context *new_ctx;

(void) duk_push_thread(ctx);
new_ctx = duk_get_context(ctx, -1 /*index*/);
```

To create a new context inside the same heap with a new global enviroment [4]:

```
duk_context *new_ctx;

(void) duk_push_thread_new_globalenv(ctx);
new_ctx = duk_get_context(ctx, -1 /*index*/);
```

Contexts are automatically garbage collected when they become unreachable. For this reason if C code holds a pointer to a Duktape context, the corresponding Duktape coroutine must be reachable from a garbage collection point of view [4].

A heap must be explicitly destroyed when it is no used anymore [4]:

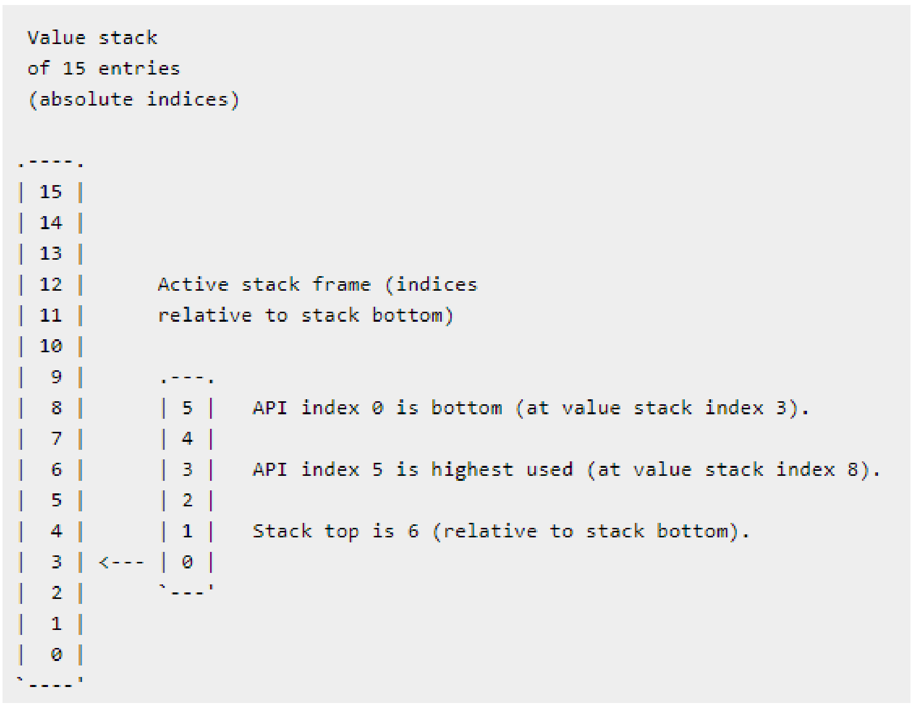
```
duk_destroy_heap(ctx);
```

This call frees all heap objects allocated, and invalidates any pointers to such objects. All the C pointers referencing objects inside the value stack are invalidated after the heap destruction and must not be de-referenced anymore [4].

2.2.1.2 Value stack and stack indeces

The value stack of a context is an array of values of different stack types, related to the current execution state of a coroutine. The stack types used in Duktape are descripted in detail in section 2.2.2.

The value stack contains the values related to all the active function calls on the coroutine's call stack. In each moment there is an active stack frame related to the current function which extabilish the bottom of the stack, which corresponds with the indexing origin. The bottom can be accessed by the Duktape API using the zero index, while the top index corresponds to the index of the highest currently used element plus one and can be obtained using duk_get_top() function [4]. The following diagram, taken by the Duktape guide [4], illustrates this concept:



```
  Value stack
  of 15 entries
  (absolute indices)

.----.
| 15 |
| 14 |
| 13 |
| 12 |      Active stack frame (indices
| 11 |      relative to stack bottom)
| 10 |
|  9 |      .---.
|  8 |      | 5 |   API index 0 is bottom (at value stack index 3).
|  7 |      | 4 |
|  6 |      | 3 |   API index 5 is highest used (at value stack index 8).
|  5 |      | 2 |
|  4 |      | 1 |   Stack top is 6 (relative to stack bottom).
|  3 | <--- | 0 |
|  2 |      `---'
|  1 |
|  0 |
`----'
```

Figure 2.2: Value stack and active stack frame. [4]

It is impossible to access to the internal value stack: Duktape API can only access to the current stack frame and there is no direct way to refer to elements in the internal value stack [4].

A **value stack index** is a signed integer index used in the Duktape API to refer to elements in currently active stack frame, relative to the current frame bottom [4].

Non-negative indices refer to stack entries in the current stack frame, relative to the frame bottom [4]:

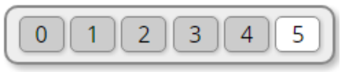


Figure 2.3: Value stack positive indeces. [4]

Negative indices refer to stack entries relative to the top:

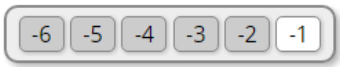


Figure 2.4: Value stack negative indeces. [4]

The value stack top is the non-negative index of an imaginary element just above the highest used index. For instance, above the highest used index is 5, so the stack top is 6. The top indicates the current stack size, and is also the index of the next element pushed to the stack [4].

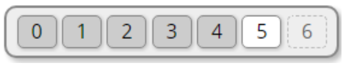


Figure 2.5: Value stack top index. [4]

2.2.1.3 Duktape/C function

Using the Duktape API it is possible to associate a C function to a JavaScript function object in order to call the first when the JavaScript function is called. A Duktape/C API function is defined as follows [4]:

```c
duk_ret_t my_func(duk_context *ctx) {
    duk_push_int(ctx, 123);
    return 1;
}
```

The value stack of the context contains the argument passed to the function and duk_get_top() can be used in the first lines of code to obtain the number of arguments inside the value stack.

The this binding, which in case of a method call corresponds to the object having the function as method, is not automatically inserted into the stack and can be accessed using duk_push_this() API call.

When a Duktape/C function wrapping a C one is created, the number of arguments must be specified so that, when the function is called, extra arguments are dropped and missing arguments are replaced with undefined. DUK_VARARGS can be used instead of the number of arguments to indicate a variable number of arguments [4].

Return value of Duktape/C functions can be:

- 1: to indicate that the value on the top of the stack is the return value [4].
- 0: to indicate that there is no explicit return value on the value stack; an undefined is returned to caller [4].
- negative: to indicate that an error is to be automatically thrown. Error codes named DUK_RET_xxx map to specific kinds of errors (do not confuse these with DUK_ERR_xxx which are positive values) [4].

Negative error return values makes error handling easier, avoiding to construct and explicitly throw errors using Duktape API calls. However, adopting this approach, the error message cannot be customized and it is automatically generated by Duktape [4].

Duktape/C functions can be also used as constructor if the related JS functions are preceded by the keyword **new**. Duktape API also provides a function in order to check if the function is called as constructor or not [4]:

```
static duk_ret_t my_func(duk_context *ctx) {
    if (duk_is_constructor_call(ctx)) {
        printf("called as a constructor\n");
    } else {
        printf("called as a function\n");
    }
}
```

2.2.1.4 Error handling

As for the JavaScript language, Duktape permits to throw errors either explicitly or implicitly and subsequently to catch and handle them. The main difference is that, instead of try-catch statements, the protected Duktape API calls are used to indicate the areas where errors can be caught and handled.

If an error is not caught, the fatal error handler is called, putting the system in an unrecoverable state, an undesired situation which should be avoided [4].

To avoid fatal errors, Duktape API protected calls, like duk_peval(), duk_pcompile() and duk_pcall(), must be used to establish an error catch point [4].

Despite protected calls usage, fatal errors are not completely avoided. There are cases in which abnormal conditions are not recoverable causing a fatal error or propagating an error outside the protected call. For this reason it is important to provide a custom fatal error handler when creating a new heap region [4].

2.2.2 Stack types

Duktape provides the following stack types [4]:

Table 2.1: Duktape Stack Types.

Type	Description
(none)	no type (missing value, invalid index, etc)
undefined	undefined
null	null
boolean	true and false
number	IEEE double
string	immutable (plain) string or (plain) Symbol
object	object with properties
buffer	mutable (plain) byte buffer, fixed/dynamic/external; mimics an Uint8Array
pointer	opaque pointer (void *)
lightfunc	plain Duktape/C function pointer (non-object); mimics a Function

The stack types listed in the table above involve additional heap allocations [4]:

- **String**: a single allocation contains a combined heap and string header, followed by the immutable string data [4].

- **Object**: one allocation is used for a combined heap and object header, and another allocation is used for object properties. The property allocation contains both array entries and normal properties, and if the object is large enough, a hash table to speed up lookups [4].

- **Buffer**: for fixed buffers a single allocation contains a combined heap and buffer header, followed by the mutable fixed-size buffer. For dynamic buffers the current buffer is allocated separately. For external buffers a single heap object is allocated and points to a user buffer [4].

Note that while strings and buffers are considered a primitive (pass-by-value) types, they are a heap allocated type from a memory allocation point of view [4].

2.2.2.1 Duktape buffer objects

Duktape provides different types of buffer. Principally it offers both memory efficient primitive plain buffer types, with reduced functionalities, and more complex buffer objects like ArrayBuffer and DataView [4]. Duktape API provides function to create, configure buffer and access buffer data.

2.2.2.2 Duktape error objects

In Duktape it is possible to create and throw error objects. This objects have the following properties: the name of the error, an optional error message, a file name, a line number and a traceback stack [4].

2.2.2.3 Duktape function objects

Function objects are very similar to the JavaScript one and just adds the name and filename properties besides the standard length, prototype, caller and arguments JS properties [4].

2.2.3 Finalization

Duktape garbage colloction uses a combined reference counting and mark-and-sweep approach. When an object is detected as unreachable, the finalizer is called in order to free the native resources associated with the JavaScript object. Duktape provides a method to customize object finalization, permitting to define finalizers either as JavaScript functions or Duktape/C functions [4].

Finalization in Duktape ensures the following behaviors:

- Finalizers are executed when an object is detected as unreachable from garbage collector or mark-and-sweep. However finalizers are not called immediately when this situation arises [4].

- Finalizers also called for all the objects when the heap is destroyed [4].

- Finalizers are called exactly once, except in some situations where the object is rescued during a mark-and-sweep cycle. For this reason finalizer functions must avoid to free the same resource multiple times [4].

However, there are some rare cases in which this rules are not respected causing undeterministic behaviors.

2.2.4 Threading

There are some limitations when using multi-threading in Duktape:

- Inside a particular Duktape heap, created using duk_create_heap() API call, only one native thread can execute code at a time. During program execution more native threads can access to the same heap if they do not do it in the same time [4].

- Using the API calls duk_suspend() Duktape execution can be suspended and then resumed through duk_resume(). In this timeframe another thread may call Duktape/C functions to access the same heap. The application hosting Duktape is charge of provide locking mechanism in order to avoid that multiple native threads execute code at the same time [4].

- Different Duktape heaps are complitely isolated from each other and multiple native threads can execute code at the same time in case the heap is not shared [4].

Furthermore Duktape heap is a single region for garbage collection regardless of how many Duktape threads exist in the heap (don't confuse native threads and Duktape threads). A synchronization mechanism for memory handling should be provided when different threads reside in the same heap. Synchonization could be a portability issue and for this reason it is always recommended to execute a single thread in the heap [4].

2.2.5 Performance

Duktape is an interpreted engine, for this reason performance is similar to other interpreted languages. Furthermore garbage collector, although minimizes memory usage, slightly reduces execution performance [4].

2.2.6 Memory usage

Duktape memory allocation is on demand and a pre-allocated heap is not needed. When a new heap is created on a 32-bit microprocessor, 70kB are required for the built-in JavaScript objects. By using specific low memory options, it is possible to reduce this initial memory occupation to 27kB. Duktape also provides the possibility to move built-in objects and strings into ROM to furhter reduce it to about 3kB. Furthermore custom native bindings can be moved to ROM too [4].

After the heap is created, further memory is allocated when executing application scripts. Reference counting permits to minimize the unused allocated memory, the only exception being objects partecipating in reference loops, which are eventually collected by mark-and-sweep [4].

Duktape memory allocations can be divided into two categories. First, there are a lot of small allocations between 16 to 128 bytes which for strings, buffers, objects, object property tables, etc. Second, there are much fewer larger allocations needed for e.g. Ecmascript function bytecode, large strings and buffers, value stacks, the global string table, and the Duktape heap object [4].

In case Duktape have to be executed on low memory systems, with less than 1MB of RAM, it is recommended to provide custom allocation functions when creating an heap area. A pool-based allocations, for example, is a good solution avoiding fragmentation issues. On the other hand memory pool sizes must be tuned to match the concrete allocation patterns [4].

2.3 ChibiOS

ChibiOS is an OpenSource development environment for embedded applications including RTOS, an Hardware Abstraction Layer, peripheral drivers, support files and tools.

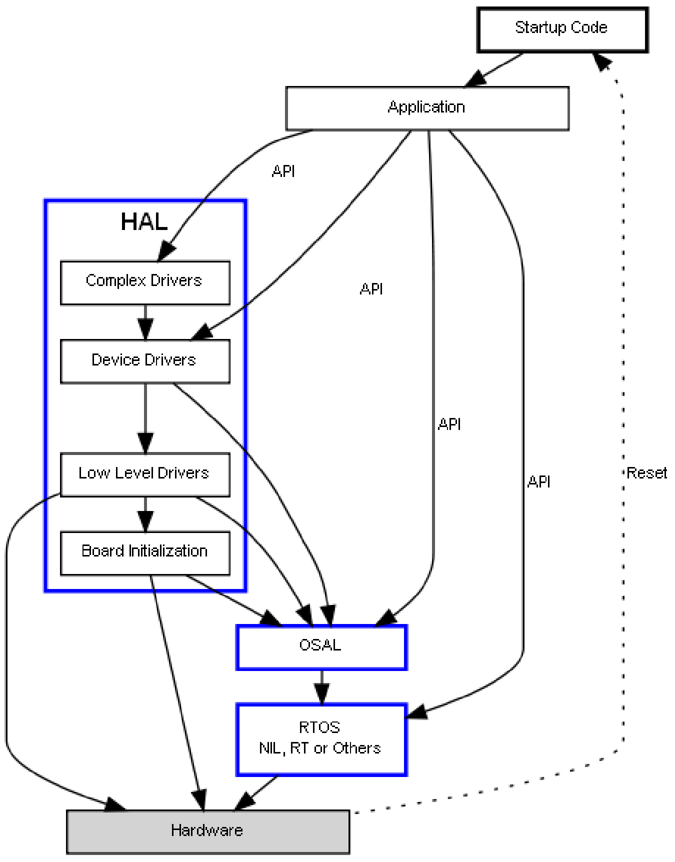


Figure 2.6: ChibiOS Architecture. [7]

Starting from the bottom of the architecture showed above, ChibiOS offers:

- Two RTOS solutions according to the needs of the developer:
  - **RT**: an RTOS solution focused on performance [5].
  - **NIL**: an RTOS solution focused on memory size targeting very small devices [5].
- **HAL**: a package containing the most common device drivers. Its layered structure makes HAL easily portable on different platforms and furthermore an **Operating System Abstraction Layer** makes HAL completely RTOS-aware [5].

2.3.1 HAL

The HAL component provides an Hardware Abstraction Layer between the application and the underlying micro-controller hardware [6]. Its architecture has been inspirational for developing Scriptly, the new layer which extends ChibiOS applications through scripting.

HAL offers an high level API to developers for accessing common microcontroller peripherals offering an object-oriented approach.

Its architecture makes HAL extremely portable on different platforms and RTOS. This ChibiOS subsystem is in fact structured in several internal layers [7]:

- **Device Drivers**: It is the layer containing portable device drivers.
  This layer take care of [7]:
  - Driver API.
  - API parameter checks.
  - Driver state machine handling and checks using assertions.
  - Low level driver invocation for inner functionality.

  This is the layer that exports the API used by the main application. This layer is perfectly portable, there are no dependencies on any specific HW architecture [7].

- **Low Level Drivers**: In this layer the platform-specific device driver implementations are contained. The application does not interact directly with this layer but uses the API provided by the high level device drivers [7]. The match between the high level driver and the low level driver depends on the header inclusions inside the Makefile.

- **Board Initialization**: This layer encapsulate all the dependencies between the HAL and the physical board [7]:
  - Board name.
  - Mounted MCU model.
  - Initial GPIO settings.
  - Clock details, for example oscillators frequencies.
  - Board-related initializations, for example external devices.

  The board initializer is called internally by the HAL, the application does not interact directly with this layer [7].

- **Complex Drivers**: Complex Drivers are higher level drivers that do not interact directly with the hardware but use other drivers in order to perform their I/O functions. For example an LCD driver could use the SPI driver for communication without interacting directly with the SPI hardware [7].

2.3.1.1 The HAL Object-Oriented approach

Even if C is not a Object-Oriented language, the HAL device drivers are structured with an Object-Oriented approach.

Each ChibiOS device driver subsystem, in fact, contains the definition of a structure representig the driver object. Driver structures contain a virtual methods table and the driver data. As example, the serial driver structure follows:

```
struct SerialDriver {
/** @brief Virtual Methods Table.*/
const struct SerialDriverVMT *vmt;
_serial_driver_data
};
```

The virtual methods table, which is a structure of function pointers, is filled with the device drivers methods, when sdInit() is called during the HAL initialization phase. Each method is an high level driver API call, receiving the driver structure as argument and providing the same interface for different low level implementations.

The driver data, declared inside the specific low level driver, is filled during the driver configuration phase when the API call sdStart() is called. Considering the serial objects, the driver data are the following:

```
#define _serial_driver_data                                    \
    _base_asynchronous_channel_data                            \
    /* Driver state.*/                                         \
    sdstate_t                  state;                          \
    /* Input queue.*/                                          \
    input_queue_t              iqueue;                         \
    /* Output queue.*/                                         \
    output_queue_t             oqueue;                         \
    /* End of the mandatory fields.*/                          \
    /* Pointer to the USART registers block.*/                 \
    USART_TypeDef              *usart;                         \
    /* Clock frequency for the associated USART/UART.*/        \
    uint32_t                   clock;                          \
    /* Mask to be applied on received frames.*/                \
    uint8_t                    rxmask;
```

Macro functions are also defined in the device driver header file in order to access its methods. As well as methods even macro functions receives the driver structure as first argument. An example is reported in the code below:

```
#define sdWrite(sdptr, b, n)                                   \
    oqWriteTimeout(&(sdp)->oqueue, b, n, TIME_INFINITE)
```

2.3.2 ChibiOS events

This section provides a brief explanation of the ChibiOS events handling, having partially exported this subsystem to JavaScript.

ChibiOS events primarily permits device drivers to notify the application that something happened at the I/O level. One or more threads can wait for one or more events in a many to many relationship [8].

Events are very useful when a thread must wait for multiple conditions and permit to synchronously check for events asynchronously broadcasted [8].

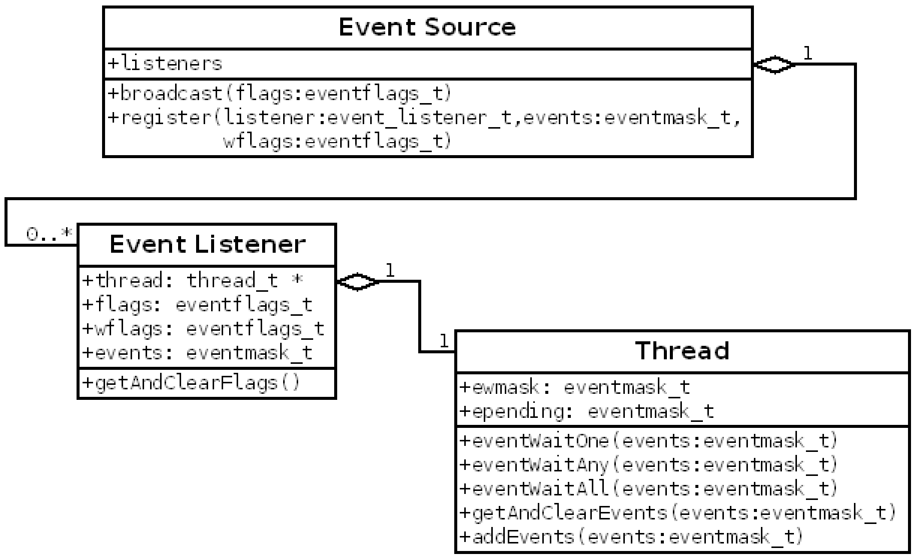

Event Source
+listeners
+broadcast(flags:eventflags_t)
+register(listener:event_listener_t,events:eventmask_t, wflags:eventflags_t)
1
0..*
Event Listener
+thread: thread_t *
+flags: eventflags_t
+wflags: eventflags_t
+events: eventmask_t
+getAndClearFlags()
1
1
Thread
+ewmask: eventmask_t
+epending: eventmask_t
+eventWaitOne(events:eventmask_t)
+eventWaitAny(events:eventmask_t)
+eventWaitAll(events:eventmask_t)
+getAndClearEvents(events:eventmask_t)
+addEvents(events:eventmask_t)

Figure 2.7: ChibiOS Events. [8]

Three class of objects are involed in the ChibiOS event handling:

- **Event listeners** are objects linked with a single thread which can register on an event source also indicating on what kind of events they are insterested in [8].
- **Event sources** are objects which can be broadcasted to notify all threads registered on the source that an event occurred. The informations related to the occurred event are encoded as **event flags** [8].
- **Thread** objects have an event listener for each event source they are listening. Its attribute ewmask is the mask of events the thread is interested in and epending is the mask of the events waiting to be served by the thread [8].

2.3.3 ChibiOS shell

ChibiOS offers a commands line shell which has been modified, following the steps shown in the Methods chapter, to obtain the Scriptly shell. In this section its code is explained in order to understand the changes made.

The shell offers a default command list, defined as an array of shell commands. Shell commands are simply C structures matching a string indentifying the command name to a pointer to the C function they execute.

Custom commands can also be added to an extra command list contained inside a configuration structure passed as argument to the shell thread function.

It is possible to add commands to the default command list or to an extra list matching commands to function having the following prototype:

```
static bool cmd_func(const ShellCommand *scp,
   BaseSequentialStream *chp, char *name, int argc, char
   *argv[]);
```

The C API provided by the ChibiOS shell subsystem is mainly composed of three functions: the shell thread function and two API calls to initialize and terminate the shell.

The shell initialization function just initializes an event source used to notify the termination of the shell:

```
void shellInit(void) {
    chEvtObjectInit(&shell_terminated);
}
```

The shell termination function is used to broadcast the shell_terminated event and terminate the shell thread in kernel lock state:

```
void shellExit(void){
    chSysLock();
    chEvtBroadcastI(&shell_terminated);
    chThdExitS(msg);
}
```

The thread function is the core of the shell. Analyzing and conceptually explaining its body, firstly it receives a configuration structure as argument to provide a list of extra commands and the I/O channel associated to the shell.

```
THD_FUNCTION(shellThread, p){
    ShellConfig *scfg = p;
    BaseSequentialStream *chp = scfg->sc_channel;
    const ShellCommand *scp = scfg->sc_commands;
```

Then it enters in an infinite loop in which firstly the input line inserted into the shell is obtained, then the words composing the string are splitted and stored in separate variables. At last the function cmdexec(), to execute shell commands, is called twice, passing the default command list and the extra command list as argument.

```
if (cmdexec(shell_local_commands, chp, cmd, n, args) &&
   ((scp == NULL) || cmdexec(scp, chp, cmd, n, args))) {
    chprintf(chp, "%s", cmd);
    chprintf(chp, " ?"SHELL_NEWLINE_STR);
}
```

the return value, indicating if the command has been executed or not, is checked and, in case it is 1 for both the calls, an error message is written on the shell I/O channel.

The loop can be broken using the exit command.

2.3.4 ChibiStudio

ChibiStudio is a free ARM development environment based on Eclipse and other Open Source tools and components. It has been created in order to support the development of ChibiOS. The enviroment permits to create, modify and import new ChibiOS projects and provides a C/C++ perspective and a Debug perspective

3
Methods

This chapter is dedicated to the methods and the adopted strategy to integrate Duktape in ChibiOS. Starting from the first approach to the JavaScript engine, the implementation details of Scriptly will be addressed motivating the design choiches.

3.1 Duktape in ChibiOS

3.1.1 Used material

To develop Scriptly, the new ChibiOS layer aimed at extend microcontroller application development through scripting, the following components, tools and target board has been used:

- **ChibiStudio**:
  ChibiStudio is an open-source Eclipse development enviroment encapsulating all the necessary material to work with ChibiOS.
  The adopted version to develop Scriptly is ChibiStudio_Preview19.7z.

- **Duktape**:
  Duktape is an open-source JavaScript engine whose purpose is to extend the functionalities of a C/C++ application through scripting.
  The adopted version to develop Scriptly is Duktape v2.2.0.

- **STM32F746ZG-NUCLEO144**:
  As recommended in the Duktape documentation an adequate target board to run Duktape in default counfiguration must have at least 256-384kB system flash memory (code) and 256kB system RAM [9]. My choice was thus to use the STM32F746ZG-NUCLEO144 board, whose microcontroller has the following features:

  - **Core** [10]:
    ARM® 32-bit Cortex®-M7 CPU with FPU, adaptive real-time accelerator (ART Accelerator™) and L1-cache: 4KB data cache and 4KB instruction cache, allowing 0-wait state execution from embedded Flash memory and external memories, frequency up to 216 MHz, MPU, 462 DMIPS/2.14 DMIPS/MHz (Dhrystone 2.1), and DSP instructions.

- **Memories** [10]:
  - Up to 1MB of Flash memory.
  - 1024 bytes of OTP memory
  - SRAM: 320KB (including 64KB of data TCM RAM for critical real-time data) + 16KB of instruction TCM RAM (for critical real-time routines) + 4KB of backup SRAM (available in the lowest power modes)
  - Flexible external memory controller with up to 32-bit data bus: SRAM, PSRAM, SDRAM/LPSDR SDRAM, NOR/NAND memories.

  For a complete overview of its features please refer to the official STMicroelectronics documentation.

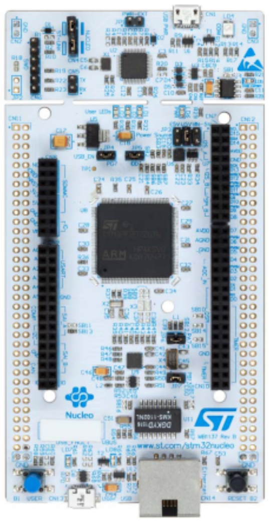


Figure 3.1: STM32F746ZG-NUCLEO144. [14]

However Scriptly is easily portable on many target boards just modifying little pieces of code. Furthermore, by enabling low memory options inside the Duktape configuration file, it is possible to minimize its memory footprint making it portable on very-low-memory enviroments.

3.1.2 A first approach to Duktape

As starting point the ChibiOS demo "RT-STM32F746ZG-NUCLEO144" is used. The project is an example already configured to run on the specific board and it just makes the three leds blink with a certain frequency.

Following the instructions presented in the Duktape Programmer's Guide:

- The preconfigured sources and header contained in Duktape/src-noline/ folder are included inside the ChibiOS project build.

- The Makefile has been modified to include the necessary libraries and to enable the following recommended compiler options for GCC/clang:
  - -std=c99: recommended to ensure C99 semantics which improve C type detection and allows Duktape to use variadic macros [4].
  - -Wall: recommended to catch potential issues early [4].
  - -Os: optimize for smallest footprint, which is usually desired when embedding Duktape. -O2 is a good compromise for performance optimized builds [4].
  - -fomit-frame-pointer: omit frame pointer, further reduces footprint but may interfere with debugging (leave out from debug builds) [4].
  - -fstrict-aliasing: use strict aliasing rules, Duktape is compatible with these and they improve the resulting C code [4].

Furthermore it is fundemental to set the stack size of the ChibiOS native thread executing Duktape to a value major than 16384 Bytes to avoid unexpected behaviors.

After this steps a Duktape context can be initialized in the main function, in order to check the correct behavior, using the following test code:

```c
duk_context *ctx = duk_create_heap_default();
duk_eval_string(ctx, "1+2");
int result = duk_get_int(ctx, -1));
duk_destroy_heap(ctx);
```

The instructions above create an heap area and an initial context; evaluate the JavaScript string "1+2", write the result into an integer variable named "result" and finally destroy the heap. The content of the variable can be observed in ChibiOS degug perspective once the application is flashed and run.

3.2 Scriptly

Scriptly is the new layer added to ChibiOS to program on microcontrollers through JavaScript. It uses the Duktape API to export ChibiOS device drivers and functions, exploiting the features of JavaScript and providing a JS API to access the microcontroller peripherals through the object-oriented paradigm. In this section its architecture and the adopted choiches to minimize memory requirements are explained in detail.

3.2.1 General architecture

In order to make Scriptly perfectly integrated with ChibiOS, the higher level of HAL has been the reference model for its development.

Inspired by the ChibiOS layer, Scriptly is composed of:

- A configuration file to enable and disable pieces of code.
- An header and source file to provide the API call for the initialization of a Scriptly context.
- Several subsystems, each one creating a JS object exporting device drivers or RTOS functions.

Furthermore a shell has been developed in order to program at run-time using the JavaScript API provided by Scriptly.

3.2.2 Internal file organization

Not only the general structure but also the internal file organization is inspired by the ChibiOS HAL subsystems. For each created JS object, exporting a device driver or functions provided by the RTOS, the code is in fact divided between an header and a source file, structured as follows:

- The header is splitted in the following sections:
  - Constants.
  - Pre-compile time settings.
  - Derived constants and error checks.
  - Data structures and types.
  - Macros.
  - External declarations.

- The source is splitted in the following sections:
    - Local definitions.
    - Exported variables.
    - Local variables and types.
    - Local functions.
    - Exported functions.

The goal of this structure is to make the code more readable, falicitating the modification, deletion and addition of pieces of code and guiding developers who can use this template to add new Scriptly subsystems.

3.2.3 Scriptly configuration

Scriptly configuration header file allows application developers to enable or disable Scriptly subsystems basing on the developer choiches. This behavior is obtained through some macro whose value is checked in the Scriptly subsystems files using conditional directives.

Enabling and disabling pieces of code, it is possible to save both code memory and RAM, avoiding to allocate useless objects in the heap area of a context and speeding up the context initialization phase calling only the necessary functions.

En example of the macros defined in the configuration file is the following:

```
#if !defined(SCRIPTLY_USE_PAL) || defined(__DOXYGEN__)
#define SCRIPTLY_USE_PAL                            TRUE
#endif
```

After verifying that the macro is not already defined, a directive to the pre-compiler is used to indicate that the code inside the Port Abstraction Layer subsystem must be enabled. A developer could modify this line setting SCRIPTLY_USE_PAL to FALSE, disabling the corresponding subsystem.

The source and header file dedicated to the PAL export is thus enclosed between the following conditional pre-compiler directives:

```
#if (SCRIPTLY_USE_PAL == TRUE) || defined(__DOXYGEN__)
/* CODE */
#endif
```

In this way, in case SCRIPTLY_USE_PAL is set to FALSE the code is not compiled.

3.2.4 Scriptly library

One of the goals of Scriptly is the easy integration inside a ChibiOS application. By including only the header "scriptly.h" inside the main source, it is possible to initialize Duktape contexts populating their heap with JS objects to access the microcontroller peripherals and to call ChibiOS functions from JavaScript.

All the necessary inclusions are already done inside "scriptly.h" code:

```
#include "ch.h"
#include "hal.h"
#include "duktape.h"
#include "scriptlyconf.h"
```

As well as the inclusions of the other Scriptly subsystems:

```
#include "scriptly_events.h"
#include "scriptly_mtx.h"
/* ... */
#include "scriptly_shell_cmd.h"
```

The header contains the declaration of the only C function that developers must use to initialize a Duktape context:

```
void scriptly_heap_init(duk_context *ctx);
```

The function receives a Duktape context pointer as argument and can be invoked to create the demanded JavaScript driver objects inside the global enviroment of the context.

Its body, defined inside the source file, is composed of several group of instructions like the following:

```
#if (SCRIPTLY_USE_SERIAL == TRUE) || defined(__DOXYGEN__)
scriptly_create_SD_objects(ctx);
#endif
```

So, in this example, after verifying that the Scriptly subsystem is enabled in the configuration file, a function to create the JS objects exporting the serial driver is called, opportunely populating the context heap.

To create and initialize a context with its own heap area, populated by the objects defined in Scriptly, the application developer can thus use these two lines of code:

```
duk_context *ctx = duk_create_heap_default();
if(ctx) scriptly_heap_init(ctx);
```

Another section of "scriptly.h" is dedicated to the error checks on the configuration header file. As shown in the code below, the configuration macros are defined as false, in case they are not yet defined, with the following directives:

```
#if !defined(SCRIPTLY_USE_PWM)
#define SCRIPTLY_USE_PWM                FALSE
#endif
```

3.2.5 Exporting RTOS functions as JS object methods

By using the Duktape API, it is possible to export C functions to JavaScript and to put them as properties of an object. A good example in Scriptly is the JS thread object, whose methods hide calls to ChibiOS functions to handle the state of a native thread.

As reported in the Duktape Programmer's Guide, only one native thread can execute code within a single heap at any time [4]. For this reason a single JavaScript thread object is created inside the heap of a Duktape context when the Scriptly context initialization function is called.

The Scriptly subsystem exporting the thread object contains the external declaration of the following function to create the JS object inside the heap of a Duktape context:

```
void scriptly_create_Thd_object(duk_context *ctx);
```

The function pushes an empty object in the value stack of the Duktape context passed as argument, storing its stack index into a variable:

```
duk_idx_t Thdobj_idx = duk_push_object(ctx);
```

Then the following Duktape API call is used to set the functions contained inside the function list "thd_funcs" as properties of the thread object.

```
duk_put_function_list(ctx, Thdobj_idx, thd_funcs);
```

At last the thread object is registered as property of the global object of the Duktape context, naming it "Thd".

```
duk_put_global_string(ctx, "Thd");
```

The function list associates the Duktape/C functions to the JS object methods and looks like the following:

```
static const duk_function_list_entry thd_funcs[] = {
{ "sleepMilliseconds",
   scriptly_thread_sleep_milliseconds, 1 },

#if (SCRIPTLY_USE_EVENTS == TRUE) || defined(__DOXYGEN__)
{ "evtWaitOneTimeout", scriptly_event_wait_one_timeout,
   DUK_VARARGS },
/* ... */
#endif

{ NULL, NULL, 0 }
};
```

It is an array of triples { name, function, nargs }, ending with a NULL triple, where **name** is the name of the method of the JS object, **function** is the Duktape/C function wrapping the ChibiOS API function, and **nargs** is the number of arguments of the JavaScript function.

As written in the theory chapter extra arguments are dropped while the missing ones are substitued with undefined. DUK_VARARGS is used when this behavior is not required and indicates a variable number of arguments.

Taking as example the second function in the list above, the prototype of a Duktape/C function looks as follows:

```
static duk_ret_t
   scriptly_event_wait_one_timeout(duk_context *ctx);
```

The Duktape context pointer is passed as argument of the Duktape/C function and an integer value of duk_ret_t type is returned. In case the return value is 1, the value on the top of the context value stack is interpreted as return value of the method; in case 0 is returned, it indicates that no return value must be pushed into the stack; at last negative return values identifies an error and automatically throw an error object [4].

The body of Duktape/C functions defined in Scriptly is always structured in sections. Considering the same Duktape/C function as example, the first part is dedicated to a verification on the number of arguments:

```
const int n_args = duk_get_top(ctx);

if(n_args < 1 || n_args > 2)
    return duk_type_error(ctx, "In function
       OS.evtWaitOneTimeout: invalid number of
       arguments: %d\n\r", n_args);
```

The API call "duk_get_top()" provides the index of the top value of the stack plus one. When calling the JS thread object method, the underlying Duktape/C function is called and the active stack frame contains only the arguments passed to the function from JavaScript. For this reason, pratically speaking the Duktape API call returns the number of arguments passed to the function. The value of "n_args" is then checked and, in case it is different from the expected one, the Duktape API call "duk_type_error()" is called to create and push an error object into the value stack with a type error code and a custom error message string.

Subsequently, arguments inside the active stack frame are stored into variables using the appropriate Duktape/C function. Their values are also checked before passing it to the wrapped function, throwing error objects in case they do not suit the C function prototype.

```
eventmask_t events;

double events_double = duk_require_number(ctx, 0);
if(!isInteger(events_double))
    return duk_type_error(ctx, "In function
       OS.evtWaitOneTimeout: integer required, found %f
       (stack index %d)\n\r", events_double, 0);
if(!isInRange(events_double,0,UINT32_MAX))
    return duk_range_error(ctx, "In function
       OS.evtWaitOneTimeout: argument out of range,
       expected [0,%d], found %d (stack index %d)\n\r",
       UINT32_MAX, (eventmask_t) events_double, 0);
events = (eventmask_t) events_double;
```

In the example above the Duktape/API call "duk_require_number" receives the context and the stack index as arguments and returns a double value taken from the stack, throwing an error in case it is not actually a number. In order to check the argument value, the function verifies that the inserted number is an integer and that it is within the desired range, throwing custom error objects if not. At last, if all the verification are passed, the value is casted to the type required by the wrapped C function. As the number of arguments is variable in this case, the same check is done for the second argument inside an if condition which verifies that the passed arguments are effectively two.

The wrapped C function is then called passing the casted arguments:

```
eventmask_t ret = chEvtWaitOneTimeout(events,timeout);
```

At last the return value is pushed into the stack and the Duktape/C function returns 1 to indicate the presence of a return value.

```
eventmask_t ret = chEvtWaitOneTimeout(events,timeout);
duk_push_int(ctx,ret);
return 1;
```

3.2.6 Exporting the ChibiOS events subsystem

The events handling, briefly explained in section 2.3.2, is an important feature of ChibiOS which is partially exported to Scriptly.

Respect to the ChibiOS event handling, in Scriptly event source objects are not declarable using JavaScript. The recommended approach to program in JavaScript using Scriptly, in fact, is to execute Duktape contexts in different native threads. Furthermore respecting the rules dictated by the Duktape documentation, multiple native threads must execute code in different heap areas at the same time, to avoid undesired behaviors [4]. Because of these constraints, Duktape contexts in Scriptly do not share the same heap and objects declared by a JavaScript thread of execution are not visible by the others. It is thus impossible in Scriptly to declare an event source from JavaScript code on which another context can register without adopting complex programming strategies.

However Scriptly event sources C pointers can be put as properties of JS objects by C code and it is possible to register and unregister on them using the object methods.

Considering for example the exported JS serial object, better discussed in section 3.2.8.1, it has a pointer to the ChibiOS serial driver structure as property. This ChibiOS structure contains an event source that can be accessed by the Duktape/C functions of the JS object.

The following example code, shows how the Duktape/C function corresponding to the "register" JS method can access to the pre-defined event source:

```c
duk_push_this(ctx);

duk_get_prop_string(ctx, -1, "driver_pointer");
SerialDriver *SDptr = (SerialDriver *)
   duk_get_pointer(ctx,-1);

/* CODE NOT REPORTED */

chEvtRegister(chnGetEventSource(SDptr), SDel, id);
```

After pushing the this binding into the stack, the function pops the serial driver pointer property of the object from the stack. Subsequently the ChibiOS function to register the thread on an event source is called. The event source is passed as argument using "chnGetEventSource()" ChibiOS API call which returns the event source of the serial driver structure.

3.2.6.1 Creating a JS prototype: the event listener

Another faced issue to export the events in Scriptly is the creation of a JavaScript prototype which export the event listener abstract data type. In fact a thread must be able to create as event listener objects as needed when executing JavaScript code in order to extabilish a many-to-many relation between threads and event sources.

JavaScript uses function objects as object constructor just preceding their call with the keyword **new**. So, in order to export the event listener class to JavaScript, a Duktape/C function call to allocate a new event listener object is defined inside the Scriptly event listener subsystem.

By using a specific Duktape API call, the Duktape/C function verifies that it is called as a constructor, throwing an error in the negative case:

```
if (!duk_is_constructor_call(ctx))
    return duk_type_error(ctx, "Function
       EvtListener_ctor must be called as constructor");
```

The this binding, linked to the new constructed object, is then pushed into the value stack of the context:

```
duk_push_this(ctx);
```

An event listener object is dynamically allocated using "malloc()" and its pointer is put as property named "data" to the this binding:

```
event_listener_t *elptr = (event_listener_t *)
   malloc(sizeof(event_listener_t));

duk_push_pointer(ctx,elptr);
duk_put_prop_string(ctx,-2,"data");
```

Furthermore a boolean flag called "deleted", checked by the finalizer, is also put as property of the this binding:

```
duk_push_boolean(ctx,false);
duk_put_prop_string(ctx,-2,"deleted");
```

Finally a Duktape/C function is set as finalizer of the object:

```
duk_push_c_function(ctx,scriptly_el_dtor,1);
duk_set_finalizer(ctx,-2);
```

Inside the Duktape/C finalizer function the "deleted" flag is obtained as boolean property of the first argument, which corresponds to the object being finalized, and stored inside a variable:

```
duk_get_prop_string(ctx, 0, "deleted");
bool deleted = duk_to_boolean(ctx,-1);
duk_pop(ctx);
```

Since the destructor may be called several time, its value is checked to verify that the object is not yet deleted. Memory is subsequently de-allocated calling "free()" and the boolean flag is set to true and put again as property of the object:

```
if (!deleted) {
    duk_get_prop_string(ctx,0,"data");
    free(duk_to_pointer(ctx,-1));
    duk_pop(ctx);

    duk_push_boolean(ctx,true);
    duk_put_prop_string(ctx,0,"deleted");
}
```

At last no value is pushed into the value stack and 0 is returned.

In the same file an external function is defined in order to export the event listener prototype inside the heap of a Duktape context. Inside its body the Duktape/C constructor function is pushed into the stack.

```
duk_push_c_function(ctx,scriptly_el_ctor,0);
```

Then its prototype property is filled with an object whose properties are the methods of the event listener object:

```
duk_push_object(ctx);
duk_put_function_list(ctx, -1, el_funcs);
duk_put_prop_string(ctx,-2,"prototype");
```

At last the function object to construct the event listener object is put in the global enviroment of the context:

```
duk_put_global_string(ctx,"EvtListener");
```

3.2.7 Exporting ChibiOS mutexes

As for the event sources it is not possible for a context to declare a mutex object visible to the other contexts residing in different heaps. For this reason mutex objects must be inserted inside Duktape heaps from C code.

However Scriptly also provides pre-defined mutex objects put as property of specific JS objects exporting device drivers.

For example the JS serial object has a property named "mutex_pointer" which hides a pointer to a C mutex structure. In the header file of the Scriptly serial subsystem as many mutex objects are declared as the number of serial driver objects created in a context heap.

```
#if (SCRIPTLY_USE_MTX == TRUE) || defined(__DOXYGEN__)

#if (STM32_SERIAL_USE_USART1 == TRUE) ||
   defined(__DOXYGEN__)
static mutex_t sMtx1;
static bool sMtx1IsInitialized = false;
#endif

#if (STM32_SERIAL_USE_USART2 == TRUE) ||
   defined(__DOXYGEN__)
static mutex_t sMtx2;
static bool sMtx2IsInitialized = false;
#endif

/* ... */

#endif  /* SCRIPTLY_USE_MTX == TRUE */
```

In case the mutex subsystem is enabled, the macro STM32_SERIAL_USE_USART1, defined in "mcuconf.h" ChibiOS header, is also checked to verify if the microcontroller peripheral is used. A static mutex variable is then declared togheter with a boolean flag indicating if the ChibiOS mutex structure is initialized or not.

Inside the function to create the JS serial objects during the context initialization phase, the following piece of code is used to put the mutex pointer as property of the serial object named "mutex_pointer":

```
#if (SCRIPTLY_USE_MTX == TRUE) || defined(__DOXYGEN__)
if(!sMtxIsInitialized){
    chMtxObjectInit(&sMtx1);
    sMtx1IsInitialized = true;
}
duk_push_pointer(ctx, (void *) &sMtx1);
duk_put_prop_string(ctx, Sobj_idx, "mutex_pointer");
#endif
```

The check on the boolean flag is necessary to avoid multiple mutex initialization when creating serial objects in different heaps.

All the JS objects having a mutex pointer as property provides methods to access the peripheral in mutual exclusion. The corresponding Dukape/C functions are externally declared inside the Scriptly mutex subsystem in order to be inserted in the function list of the exported device driver objects. Their code is very simple as showed in the code of the Duktape/C function to lock the mutex of a driver object:

```
duk_ret_t scriptly_mtx_lock(duk_context *ctx){

    duk_push_this(ctx);

    duk_get_prop_string(ctx, -1, "mutex_pointer");
    mutex_t *mtxptr = (mutex_t *)
       duk_get_pointer(ctx,-1);

    chMtxLock(mtxptr);

    return 0;
}
```

3.2.8 Exporting ChibiOS device drivers

This section is dedicated to the adopted approach to create JavaScript objects exporting ChibiOS device drivers.

3.2.8.1 The serial driver

The main goal of Scriptly is to export the ChibiOS device drivers to JavaScript. By explaining how the serial driver has been exported in Scriptly, it is possible the generalize a method which is also applied to the other drivers.

The whole header and source file code is enclosed between two pre-compiler directives checking the value of the macro defined in the configuration file in order to enable the code only when the subsystem is needed.

```
#if (SCRIPTLY_USE_SERIAL == TRUE) || defined(__DOXYGEN__)
/* SUBSYSTEM CODE */
#endif
```

Furthermore an error message is generated and showed during compilation in case the corresponding HAL driver is not enabled.

```
#if HAL_USE_SERIAL == FALSE
#error "Scriptly Serial requires HAL_USE_SERIAL"
#endif
```

The Scriptly serial subsystem header contains the following external function declaration, automatically called by the Scriptly function initializing a Duktape context:

```
void scriptly_create_SD_objects(duk_context *ctx);
```

This function is used to create inside the heap of a Duktape context as many serial objects as the number of enabled virtual serial ports.

Its body contains several sections of code, each one creating a JS object exporting a serial port driver. These sections are enclosed between two pre-compiler directives like the following:

```
#if (STM32_SERIAL_USE_USART1 == TRUE) ||
   defined(__DOXYGEN__)
/* CODE */
#endif
```

The macro "STM32_SERIAL_USE_USART1", defined inside "mcuconf.h" ChibiOS configuration file, permits to verify if the peripheral is enabled in ChibiOS, populating the global enviroment of the Duktape context only with the necessary objects, saving RAM memory and avoiding calling useless functions.

Inside each section of code, firstly an empty object is pushed into the stack saving its index in a variable:

```
duk_idx_t Sobj_idx = duk_push_object(ctx);
```

A pointer to the driver structure, which exists only if the peripheral is enabled, is then put as property of the pushed object, naming it "driver_pointer":

```
duk_push_pointer(ctx, (void *) &SD1);
duk_put_prop_string(ctx, Sobj_idx, "driver_pointer");
```

The two instructions above permits to define a unique Duktape/C function for the same method of different serial driver objects. In fact ChibiOS device drivers API calls always require the pointer to the driver structure as argument and, since each JS serial object hides a different pointer, the same function should be defined more times if this approach is not used.

After putting the mutex pointer as property as shown in section 3.2.7, the function objects contained in the function list are also put as property of the serial object:

```
duk_put_function_list(ctx, Sobj_idx, serial_funcs);
```

Finally the object, filled with its properties, is inserted in the global enviroment of the Duktape context:

```
duk_put_global_string(ctx, "S1");
```

As usual the function list is an array of triples like the following:

```
{ "readTimeout", scriptly_serial_readt, DUK_VARARGS },
```

In addition to the function above, the function list contains Duktape/C function wrapping the major part of the operations provided by the ChibiOS serial driver, permitting to write and read on the serial port also using a timeout. Furthermore functions to manage events and to lock and unlock the peripheral through a mutex are provided.

Scriptly does not to export the C functions to initialize and stop the driver. This choice has been taken in order to avoid to export complex ChibiOS configuration structures and to make the JavaScript API easier and more accessible. For this reason the initialization of a ChibiOS device driver must be done from C code.

Another faced issue in exporting the serial driver is the usage of buffers. Many ChibiOS device driver functions, in fact, receive as argument an uint8_t array to read or write inside it. Taking into account this situations the Duktape API provides several calls.

In particular, considering the Duktape/C functions wrapping ChibiOS API calls to write on a serial port, the following Duktape API call has been used to buffer-coerce the value passed as argument of the JavaScript method, returning a pointer to the buffer data:

```c
void *duk_to_buffer(duk_context *ctx, duk_idx_t idx,
   duk_size_t *out_size);
```

The Duktape/C functions to read from a serial port must require a buffer object passed from JavaScript in order to write inside its underlying data structure. This behavior is obtained using the following Duktape API call:

```c
void *duk_require_buffer_data(duk_context *ctx,
   duk_idx_t idx, duk_size_t *out_size);
```

The recommended JavaScript code to use the read function is the following:

```javascript
var buf = new Uint8Array(n);
Sx.readTimeout(buf,m,timeout);
```

3.2.8.2 The PWM driver

The ChibiOS Pulse Width Modulator device driver is also exported to JavaScript. The adopted approach is the same of the serial driver: a function is declared to create several JS PWM objects when the Scriptly function to initialize a Duktape context is called.

As for the serial driver, the body of the function contains the following steps:

- An empty object is pushed in the value stack of the context passed as argument.
- The driver pointer is put as property of the object.
- The mutex pointer is put as property.
- The function objects are put as properties.

- The whole object is inserted in the global enviroment of the Duktape context passed as argument.

The function list includes methods to enable and disable a channel, enable and disable the periodic and transition notifications, lock and unlock the peripheral. As for the other drivers the function to configure the peripheral is not exported to JavaScript. In particular in the PWM case this is an obliged choice because the configuration structure, passed as argument to the start function, associates C callback functions to the periodic activation or de-activation edge of the square wave. Since it is not possible to define C functions from JavaScript, as it would mean to define them dynamically, this behavior can not be exported.

3.2.9 Exporting the Port Abstraction Layer

Another HAL subsystem exported to JavaScript is the Port Abstraction Layer.

Similarly to the device driver wrappers a C function is defined to create in the heap of a context as many objects as the number of ports of the board. Its body contains several sections like the following:

```
#if (STM32_HAS_GPIOA == TRUE) || defined(__DOXYGEN__)
GPIOobj_idx = duk_push_object(ctx);

duk_push_pointer(ctx, (void *) GPIOA);
duk_put_prop_string(ctx, GPIOobj_idx, "base_address");
duk_put_function_list(ctx, GPIOobj_idx, pal_funcs);

duk_put_global_string(ctx, "GPIOA");
#endif
```

Since different boards may have a different number of ports the macros defined in "stm32_registry.h" are checked before creating the object. Differently from the device drivers, which hides a pointer to the driver structure, the base address of the port is put as property of the object and its value is passed to the ChibiOS PAL API calls by the Duktape/C functions wrapping them.

3.2.10 Scriptly shell

The ChibiOS Shell, explained in section 2.3.3, has been modified in order to launch Scriptly commands and control the microcontroller at run-time.

In particular a Duktape context called has been initialized inside the shell thread function before the infinite loop:

```
shell_ctx = duk_create_heap_default();

if(shell_ctx) scriptly_heap_init(shell_ctx);
else{

    chprintf(chp, "Heap creation
       failed"SCRIPTLY_SHELL_NEWLINE_STR);
    scriptlyShellExit(MSG_OK);
}
```

The lines of code where the shell command is executed have been modified in order to evaluate the input string as JS code, in case the command does not exist in the command lists:

```
if (cmdexec(shell_local_commands, chp, cmd, n, args) &&
   ((scp == NULL) || cmdexec(scp, chp, cmd, n, args))) {

    if (duk_peval_string(shell_ctx, input_line) != 0) {

        chprintf(chp, "%s"SCRIPTLY_SHELL_NEWLINE_STR,
           duk_safe_to_string(shell_ctx, -1));
    } else {

        chprintf(chp, "= %s"SCRIPTLY_SHELL_NEWLINE_STR,
           duk_safe_to_string(shell_ctx, -1));
    }
    duk_pop(shell_ctx);
}
```

The functions "cmdexec()", in fact, returns the integer value 1 if the command is not found inside the command list passed as argument. In this case the Duktape API call is used to evaluate the line passed as input to the shell as a JS string, printing on the I/O channel of the shell the result or an error message.

The function to terminate the shell has been also modified , adding a line inside its body to destroy the heap of the shell Duktape context.

```
if(shell_ctx) duk_destroy_heap(shell_ctx);
```

Furthermore a command to create a new native thread, which evaluates JavaScript strings, has been added inside the shell command list:

```
static void cmd_scriptly_create_thread
   (BaseSequentialStream *chp, int argc, char *argv[]) {

    if (argc != 1) {
        shellUsage(chp, "scriptly_create_thread");
        return;
    }

    scriptly_thd = chThdCreateFromHeap(NULL,
       SCRIPTLY_THD_WA_SIZE, "scriptly", NORMALPRIO + 1,
       scriptlyThread, (void *) argv[0]);
}
```

After a check on the number of arguments, the command dynamically creates a native thread using "chThdCreateFromHeap()" ChibiOS API call. The array element "argv[0]", containing the string passed as argument of the shell command, is passed as argument to the thread function.

Inside the thread function a new context, whose scope is limited to the function, is declared:

```
duk_context *ctx = duk_create_heap_default();
```

Then, in case the heap creation successes, the context is populated using the Scriptly initialization function:

```
if(ctx) scriptly_heap_init(ctx);
```

The string provided as argument of the thread function is then evaluated and the heap destroyed:

```
duk_peval_string(ctx, (char *) p))
duk_destroy_heap(ctx);
```

4
Results

This chapter presents the results of my thesis work at STMicroelctronics. Scriptly features are explained in detail, analyzing each component. At last a critical discussion about the usage of JavaScript engines in the embedded field is reported.

4.1 Scriptly

The result of my thesis work is Scriptly, a C library completely based on open-source components and optimized for low memory embedded microprocessors with the purpose of extending embedded C application development through scripting.

Scriptly can be considered as a new ChibiOS layer, accessing the functionalities offered by the underlying layers and providing high level services to applications.

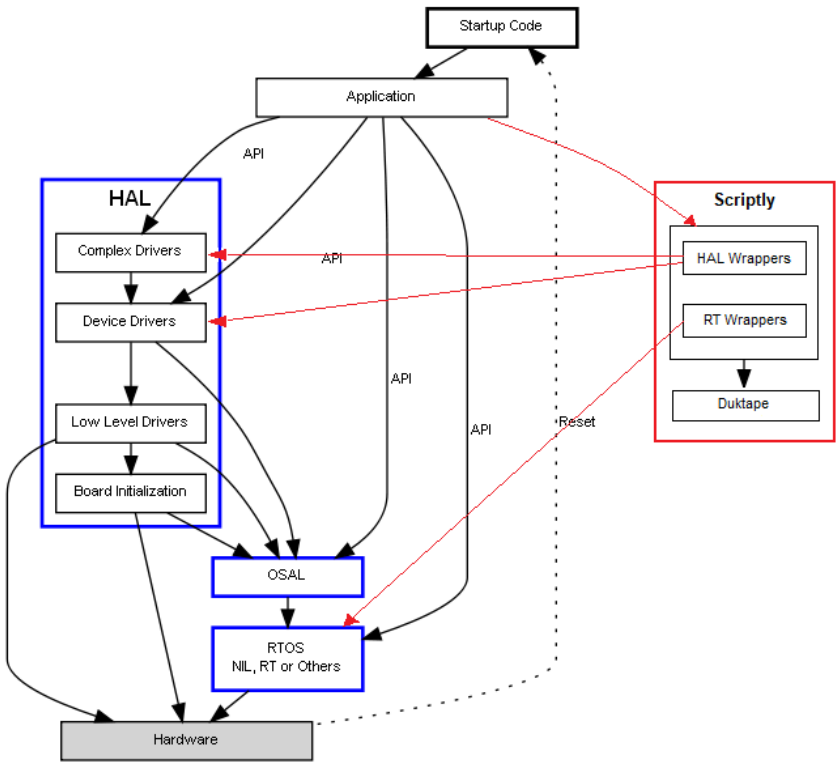


Figure 4.1: ChibiOS Modified Architecture

JavaScript features, from the object oriented paradigm to the dynamicity, are exploited in Scriptly to provide developers a JavaScript API offering an easier and more accessible way to program microcontrollers.

Using the API provided by Duktape JavaScript engine, in fact, many ChibiOS device drivers and functions have been exported to JavaScript permitting developers to design hybrid C/JavaScript applications and to access microcontroller peripherals either at run-time, through the Scriptly Shell, or evaluating JS strings from C code.

4.1.1 Scriptly objects and subsystems

By including the Scriptly library inside your build, it is possible to populate Duktape context heaps with objects whose methods hide calls to the ChibiOS API and access some of the most common microncontroller peripherals. This section is dedicated to the device drivers and RTOS functionalities accessible through JavaScript.

4.1.1.1 The thread object

Scriptly provides JavaScript API calls to handle the status of native threads executing Duktape contexts and to make them wait for one or more events.[1]

4.1.1.2 The serial driver object

Scriptly exports the serial driver object to JavaScript, permitting to write and read on virtual serial ports also using the timeout feature.[1] However, to avoid exporting complex C structures, C code must be used to initialize and configure the driver.

4.1.1.3 The PWM driver object

The exported PWM Driver permits to change the period of a PWM unit, enable and disable channels, and to enable and disable periodic and transition notifications.[1] The initialization of the PWM driver and the association of callbacks to the wave transitions, instead, must necessarely be done from C code using the ChibiOS API. It is not possible, in fact, to define C functions using JavaScript as this would mean defining them dynamically.

4.1.1.4 The port abstraction layer

The PAL driver wrapper exports functions to read and write on groups and pads as well as setting their mode.[1] Respect to ChibiOS PAL driver, IOBus structures, describing a group of contiguous digital I/O lines handled as bus, are not exported to JavaScript. Furthermore the matching between callbacks and pad events must

[1]For more details about the JS API refer to the document appendix.

be done from C code.

4.1.1.5 Events handling

The ChibiOS events handling is partially exported to Scriptly. Through a JS constructor, a thread can declare an event listener object for each event source it is interested in. After registering on one or more event sources by passing its event listener object and a numeric identifier, it is possible to wait for one, any or all the events of interest. By using "getAndClearFlags()" event listener method, the thread can understand the condition that makes the event source broadcast the event. The main limitation is that, differently from C, JavaScript threads cannot declare an event source object visible to the other contexts since they reside in different heaps. Event sources, indeed, must be declared from C and exported to context heaps.

An example of JavaScript code in which events are used is reported below:

```
var el = new EvtListener();
var id = 4;
S3.evtRegister(el,id);
var mask = 16; /* 2^4 */
var servedEvt = Thd.evtWaitAnyTimeout(mask);
var flags = el.evtGetAndClearFlags;
S3.evtUnregister(el);
```

4.1.1.6 Mutual exclusion

In Scriptly it is possible use hardware resources in mutual exclusion calling methods acting on pointers to ChibiOS mutex structures put as properties of the JS device driver objects. As for event sources, mutexes can not be declared from a JavaScript thread of execution and must be inserted in context heaps through C code.

As example, in order to lock and unlock a serial driver object, the following JS API calls must be used:

```
S1.lock;
/* critical section */
S1.unlock
```

4.1.1.7 The shell

Scriptly commands shell is an extension of the ChibiOS shell. In addition to the default ChibiOS commands, it permits to evaluate JS code strings provided to the input terminal and to create a new JavaScript thread of execution through a simple extra command.

4.1.2 Features

Scriptly has been designed to run on low-memory enviroments and to be easily portable on different platforms and RTOSs. In this section, these two features are discussed in more detail.

4.1.2.1 Optimization for low memory enviroments

The components and the design choices make Scriptly library optimized for low-memory enviroments. ChibiOS and Duktape, in fact, are both focused on compact footprint. Furthermore Scriptly provides a configuration file in order to enable and disable its subsystems, reducing code size and saving RAM memory by avoiding populating context heaps with undesired JS objects.

4.1.2.2 Portability

Scriptly is designed to be easily portable on different platforms and RTOSs. The underlying HAL architecture, in fact, is divided in different layers presenting the same interface for different hardware specific function implementations and the ChibiOS OSAL enables the HAL drivers to be fully RTOS-aware. Moreover using ChibiOS macros, Scriptly always verifies that the peripherals are available on the specific platform, creating the correspondent JS objects only in this case.

4.1.3 Performance analysis

Scriptly performance strongly depends on Duktape. As the other JavaScript engines, in fact, Duktape executes interpreted code and uses the garbage collector to automatically manage memory de-allocation [4].

Differently from compilers, which analyze and translate the whole code during the compilation phase, interpreters analyze, translate and execute each instructions at run-time. The execution time of compiled programs is thus much lower respect to interpreted ones [11]. Furthermore interpreters reduce predictability: while the execution time of a compiled program can be easily predicted basing on the clock cycles required by machine code instructions, it is more difficoult to predict the time required by the interpreter to analyze and translate the code at run-time. On the other hand, the entire cycle that includes compiling and running the program is longer than the time required for its interpretation, for this reason the usage of interpreters makes debugging easier and faster.

Regarding performance, the garbage collector brings some disadvantages. Objects finalization algorithms include many garbage collector interventions that reduce determinism and affect performance. This risk can be reduced by adopting a Javascript programming approach that limits the creation of new objects and consequently the garbage collector operations [12]. Taking these factors into account, Scriptly initializes the context heaps by populating it only with objects explicitly requested through the configuration file, also verifying that the peripheral they reflect is enabled.

4.1.4 Memory usage

4.1.4.1 Code size

Considering code memory requirements, compiling using -O2 option and enabling all the Scriptly subsystems, the binary file of the developed demo is 253.876 bytes respect to 27.088 bytes size of the starting project. The code size is thus increased and in case of low size flash memory, Duktape configuration macros and -Os compiler option could be used to optimize footprint.

4.1.4.2 RAM usage

Regarding RAM, Scriptly needs at least 70kB to create and initialize a Duktape heap and further memory to allocate objects [4]. To optimize Scriptly for low-memory enviroments, a configuration file is provided permitting to minimize the creation of objects basing on the needs of developers. Furthermore using specific Duktape low memory options, initial memory usage can be reduced to 27kB or 3kB moving objects to ROM [4].

The JavaScript interpreter can read its program either from ROM or from RAM, executing it without flashing the code. Microcontrollers RAM is usually limited and so the code size must be minimized. Regarding this, the expressive nature of high level languages such as JavaScript permits to obtain complex behaviors using very few lines of code [13].

The garbage collector automatically frees memory. With this simplification, the embedded developer may be disinterested in it, focusing on other aspects of the application. Furthermore the garbage collector eliminates many slow, small memory leaks that lead to long term instabilities that can be extremely challenging to isolate and fix. This alone makes it important for embedded systems that must operate reliably for months or years [12].

4.1.5 Personal considerations

Programming microcontrollers through Javascript is definitely an interressing novelty in recent years.

Compared to C, Javascript is certainly easier to use and more accessible to less experienced programmers. The scripting language, in fact, is more expressive and automatic memory management through the garbage collector allows developers to disinterest in the deallocation of objects.

The use of the interpreter is another very interesting aspect for programming microcontrollers. The greater advantage of the latter is indeed the ability to execute code at run-time, a very useful feature for speeding-up debugging.

On the other hand, although the performance of microcontrollers is increasing year by year, I do not think Javascript can replace C language for applications with strong real-time constraints. Real-time systems require predictability and determinism, features not guaranteed by programming through Javascript. However, an hybrid programming approach like that proposed by Scriptly is a good trade-off, permitting to use C for tasks with strong real-time constraints and JavaScript for the less critical tasks.

Memory requirements though acceptable are perhaps still excessive for the cheapest microcontrollers on the market. RAM size in fact limits the number of heap areas, and hence of contexts, that can be created. The integration of Javascript engines on embedded applications is therefore definitely a trend not to be overlooked but that probably will explode in a few years when microcontrollers will evolve further.

5
Conclusion

5.1 Achievement

After absorbing the core concepts of JavaScript, Duktape and ChibiOS, the JavaScript engine was successfully integrated into the RTOS. Scriptly, the result of my work, is C library based on open-source components and optimized for low memory embedded microprocessors. Exporting several ChibiOS device drivers and functions to JavaScript, Scriptly permits to develop C/JS hybrid applications according to the developer's choiches. C could be used for task with strong Real-Time constraints, JavaScript could be used for less critical parts of the application. Furthermore Scriptly permits to execute code either at run-time, through a commands shell, or evaluating strings from C code.

5.2 Limitations and Possible Improvements

Scriptly is still not yet in a definitive state. Several improvements can be made to further increase the Scriptly experience and to reduce its requirements:

- A first improvement could be to provide custom memory management functions, optimized for embedded devices and for RTOS systems, instead of the default ANSI C malloc(), realloc() , and free(). Furthermore, although Duktape protected calls have been used in Scriptly to avoid fatal errors a custom fatal error handler, writing the fatal error information to a flash file and rebooting the device, could be provided [4].

- Secondly the Events and Mutex subsystems could be improved in Scriptly. As described in the previous chapter, it is not possible to create from JavaScript code an event source or a mutex object visible to all the contexts and objects of these types must be pushed from C code. An interesting way to solve this problem could be the usage of memory pools of event source and mutex objects with an identifier and a reference counter to understand when an object must be created, referenced or destroyed.

- Another improvement could be to initialize at compilation time, through the Makefile, C strings with a scripts contained in .js files.
- At last more HAL drivers could be wrapped following the methods presented in this document.

5.3 Future Challenge

A possible future challenge could be the development of a Google Blockly Application to permit visual programming. Just moving and interlocking the custom blocks of the developed application it would be possible to automatically generate correct JavaScript code to be executed by a Scriptly context. This allows less experienced programmers to start using a microcontroller and to understand the basic concepts of programming with a simplified approach.

Bibliography

[1] Jon Bruner, 2016, Peter Hoddie on JavaScript for embedded systems, https://www.oreilly.com/ideas/peter-hoddie-on-javascript-for-embedded-systems

[2] Arno Slatius, 2014, JavaScript and Embedded Systems: Are They a Good Fit?, https://www.sitepoint.com/javascript-embedded-systems-good-fit/

[3] Wikipedia: JavaScript, https://en.wikipedia.org/wiki/JavaScript

[4] Sami Vaarala et al., 2017, Sami Vaarala, http://duktape.org/guide.html

[5] Giovanni Di Sirio, 2017, ChibiOS products, http://chibios.org/dokuwiki/doku.php?id=chibios:product:start

[6] Giovanni Di Sirio, 2017, ChibiOS/HAL, http://chibios.org/dokuwiki/doku.php?id=chibios:product:hal:start

[7] Giovanni Di Sirio, 2017, ChibiOS/HAL Architecture, http://chibios.org/dokuwiki/doku.php?id=chibios:product:hal:architecture

[8] Giovanni Di Sirio, 2017, ChibiOS events, http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel_events

[9] https://github.com/svaarala/duktape/blob/master/doc/low-memory.rst

[10] STMicroelectronics, 2016, STM32F746xx/STM32F746xx datasheet

[11] Wikipedia: interprete (informatica), https://it.wikipedia.org/wiki/Interprete_(informatica)

[12] Peter Hoddie, 2014, JavaScript for embedded devices, http://www.embedded-computing.com/embedded-computing-design/javascript-for-embedded-devices

[13] Sebastian Peyrott, 2017, JavaScript for Microcontrollers and IoT, https://auth0.com/blog/javascript-for-microcontrollers-and-iot-part-1/

[14] STMicroelectronics, 2016, STM32 Nucleo 144 board databrief

A
Scriptly JavaScript API

A.1 Thread Object

```
Thd.sleepMilliseconds()
```

Description:
Delays the invoking native thread for a specified number of milliseconds.
Arguments:

- msec: integer number which specifies the number of milliseconds.

Return Value:
No return value.

```
Thd.evtWaitOneTimeout()
```

Description:
Waits for one of the specified events.
Arguments:

- events: mask of the events the function should wait for.
- timeout: the number of ticks before the operation timeouts. If not passed timeout is not used.

Return Value:
The mask of the lowest event id served and cleared.
Zero if the operation timeouts.

```
Thd.evtWaitAnyTimeout()
```

Description:
Waits for any of the specified events.
Arguments:

- events: mask of the events the function should wait for.
- timeout: the number of ticks before the operation timeouts. If not passed timeout is not used.

Return Value:
The mask of the lowest event id served and cleared.
Zero if the operation timeouts.

```
Thd.evtWaitAllTimeout()
```

Description:
Waits for all the specified events.
Arguments:
- events: mask of the events the function should wait for.
- timeout: the number of ticks before the operation timeouts. If not passed timeout is not used.

Return Value:
The mask of the lowest event id served and cleared.
Zero if the operation timeouts.

A.2 Event Listener Object

```
EvtListener.getAndClearFlags()
```

Description:
Delays the invoking native thread for a specified number of milliseconds.
Arguments:
No argument required.
Return Value:
Flags associated to the EvtListener object.

A.3 Serial Object

```
Sx.write()
```

Description:
Direct blocking write.
Arguments:
- buffer: data buffer object.
- n: the maximum amount of data to be transferred (optional).

Return Value:
The number of bytes effectively transferred.

```
Sx.read()
```

Description:
Input queue read.
Arguments:
- buffer: data buffer object.
- n: the maximum amount of data to be transferred (optional).

Return Value:
The number of bytes effectively transferred.

```
Sx.put()
```

Description:
Output queue single byte write.
Arguments:
- b: the byte value to be written in the queue.

Return Value:
Zero if the operation succeeds.
-2 if the queue has been reset.

```
Sx.get()
```

Description:
Input queue single byte write.
Arguments:
No argument required.
Return Value:
Zero if the operation succeeds.
-2 if the queue has been reset.

```
Sx.writeTimeout()
```

Description:
Direct blocking write with timeout.
Arguments:
- buffer: data buffer object.
- n: the maximum amount of data to be transferred.
- timeout: the number of ticks before the operation timeouts.

Return Value:
The number of bytes effectively transferred.

```
Sx.readTimeout()
```

Description:
Input queue read with timeout.
Arguments:
- buffer: data buffer object.
- n: the maximum amount of data to be transferred.
- timeout: the number of ticks before the operation timeouts.

Return Value:
The number of bytes effectively transferred.

```
Sx.putTimeout()
```

Description:
Output queue single byte write with timeout.
Arguments:
- b: the byte value to be written in the queue.
- timeout: the number of ticks before the operation timeouts.

Return Value:
Zero if the operation succeeds.
-2 if the queue has been reset.

```
Sx.getTimeout()
```

Description:
Input queue single byte write with timeout.
Arguments:
- timeout: the number of ticks before the operation timeouts.

Return Value:
Zero if the operation succeeds.
-2 if the queue has been reset.

```
Sx.evtRegister()
```

Description:
Registers an Event Listener on the Event Source of the Serial Object.
Arguments:
- event listener: event listener object.
- id: numeric identifier assigned to the Event Listener.

Return Value:
No return value.

```
Sx.evtUnregister()
```

Description:
Unregisters an Event Listener from the Event Source of the Serial Object.
Arguments:

- event listener: event listener object.

Return Value:
No return value.

```
Sx.lock()
```

Description:
Lock the Sx mutex.
Arguments:
No argument required.
Return Value:
No return value.

```
Sx.unlock()
```

Description:
Unlock the Sx mutex.
Arguments:
No argument required.
Return Value:
No return value.

A.4 GPIOx Object

```
GPIOx.readGroup()
```

Description:
Reads a group of bits.
Arguments:

- mask: group mask, a logic AND is performed on the input data.
- offset: group bit offset within the port.

Return Value:
The group logic states.

```
GPIOx.writeGroup()
```

Description:
Writes a group of bits.
Arguments:
- mask: group mask, a logic AND is performed on the output data.
- offset: group bit offset within the port.
- bits: bits to be written. Values exceeding the group width are masked.

Return Value:
No return value.

```
GPIOx.setGroupMode()
```

Description:
Pads group mode setup.
Arguments:
- mask: group mask.
- offset: group bit offset within the port.
- mode: group mode.

Return Value:
No return value.

```
GPIOx.readPad()
```

Description:
Reads an input pad logic state.
Arguments:
- pad: pad number within the port.

Return Value:
The logic state.

```
GPIOx.writePad()
```

Description:
Writes a logic state on an output pad.
Arguments:
- pad: pad number within the port.
- bit: logic value.

Return Value:
No return value.

```
GPIOx.setPad()
```

Description:
Sets a pad logic state to the high value.
Arguments:
- pad: pad number within the port.

Return Value:
No return value.

```
GPIOx.clearPad()
```

Description:
Sets a pad logic state to the low value.
Arguments:
- pad: pad number within the port.

Return Value:
No return value.

```
GPIOx.togglePad()
```

Description:
Toggles a pad logic state.
Arguments:
- pad: pad number within the port.

Return Value:
No return value.

```
GPIOx.setPadMode()
```

Description:
Pad mode setup.
Arguments:
- pad: pad number within the port.
- mode: pad mode.

Return Value:
No return value.

A.5 PWM Object

```
PWMx.changePeriod()
```

Description:
Changes the period the PWM peripheral.
Arguments:
- period: new cycle time in ticks.

Return Value:
No return value.

```
PWMx.enableChannel()
```

Description:
Enables a PWM channel.
Arguments:
- channel: PWM channel identifier.
- width: PWM pulse width as clock pulses number.

Return Value:
No return value.

```
PWMx.disableChannel()
```

Description:
Disables a PWM channel and its notification.
Arguments:
- channel: PWM channel identifier.

Return Value:
No return value.

```
PWMx.enablePeriodicNotification()
```

Description:
Enables the periodic activation edge notification.
Arguments:
No argument required.
Return Value:
No return value.

```
PWMx.disablePeriodicNotification()
```

Description:
Disables the periodic activation edge notification.
Arguments:
No argument required.
Return Value:
No return value.

```
PWMx.enableChannelNotification()
```

Description:
Enables a channel de-activation edge notification.
Arguments:

- channel: PWM channel identifier.

Return Value:
No return value.

```
PWMx.disableChannelNotification()
```

Description:
Disables a channel de-activation edge notification.
Arguments:

- channel: PWM channel identifier.

Return Value:
No return value.

```
PWMx.lock()
```

Description:
Lock the PWM mutex.
Arguments:
No argument required.
Return Value:
No return value.

```
PWMx.unlock()
```

Description:
Unlock the PWM mutex.
Arguments:
No argument required.
Return Value:
No return value.