Спецификация к проекту Mash

MASH

It's simple!

© Mash, 2019.

Оглавление

Введение 3

Пишем “Hello world!” 6

Процедуры и функции 6

Переменные 6

Массивы 9

Типы данных 10

Константы и перечисления 10

Математические и логические операции 10

Условия if & else 12

Цикл for и for each 13

Циклы while & whilst 13

Switch & Case 14

Явные указатели 15

Классы 16

Полиморфные методы для классов 18

Статический полиморфизм (наследования) 19

Рефлексия 20

Интроспекция 21

Сборщик мусора 21

Многопоточность 22

Исключения 24

Поддержка рекурсии 25

Методы с переменным числом аргументов 25

SVM - Архитектура ВМ 26

SVM - Архитектура потока 27

SVM - Пишем свою библиотеку 27

SVM - Встраиваем Mash в своё ПО 28

SVM - Некоторые особенности 29

SVM - Формат исполняемых файлов 30

SVM API 31

Дополнительная информация 33

Введение

В этом документе изложена вся информация, которая может потребоваться для начала работы с Mash, начиная от самого языка программирования, заканчивая архитектурными особенностями SVM (Stack-based Virtual Machine) и библиотек для неё.

Для начала ответим на очевидный вопрос - зачем вообще нужен Mash? Если бы ответа на него не было, то я бы не тратил на Mash время, логично? - логично!

Каждый язык программирования создается с целью решения определенной проблемы.

Mash - простой язык, решающий сразу несколько проблем (ну... кроме проблем с производительностью конечно, по крайней мере пока.)

Очевидная проблема динамически типизированных языков, таких как Python или JS, о которой говорят многие разработчики - это неоправданно высокая гибкость языка, в отдельных его областях способствующая лишь проблемам с читаемостью кода.

Для примера возьмем JavaScript.
Наверняка вы слышали шутки про JavaScript.
“Что JS разработчик делает первым делом перед решением простой задачи? Правильно - пишет очередной фреймворк!”

Почему так происходит? JS - очень динамичный язык, поддерживающий несколько парадигм программирования. Реализовать один и тот же функционал можно кучей разных способов, отличающихся как по алгоритмической, так и по эстетической части. Иногда JS разработчики просто не в силах понять чужой код, без хорошей документации. А документацию разработчики писать не очень любят (именно поэтому эту спецификацию я выложил только сейчас...).

Что на счет Python? Тут дела обстоят не намного лучше. Python очень удобный язык - это факт. Но все-же у него есть некоторые проблемы с читаемостью кода. В Python мы можем на лету объявлять новые переменные в структуре класса. Классы объявляются одним большим куском кода, несущем в себе весь функционал. Это конечно хорошо, но что если разработчик решит заглянуть в чужой код? Для того чтобы понять что делает тот или иной класс порой нужно

потратить 5-10 минут времени. А это довольно ценный ресурс. Если учесть что Python - это удобный ЯП для построения сложных абстракций, то эта проблема только масштабируется.

Возможно что мое мнение по этому поводу слегка раздуто или же вовсе ошибочное, ведь я не профессиональный Python/JS разработчик. Но впечатление от знакомства с этими языками у меня именно такое.

В Mash я в первую очередь постарался заложить хорошую читаемость кода. Даже если код написан другим разработчиком с минимумом комментариев - он должен быть читаемым. Для этого в языке есть некоторые ограничения. Основные ограничения это:

1. Разделение объявления класса и реализации его методов.

   Пример, как нельзя объявлять классы в Mash:
   ```
   class MyClass:
     proc Foo(x):
       return x * x
     end
   end
   ```

   И сразу привожу пример, как нужно:
   ```
   class MyClass:
     proc Foo
   end

   proc MyClass::Foo(x):
     return x * x
   end
   ```

2. Если метод F возвращает какой-либо объект, имеющий метод F1, то мы не можем сразу вызвать метод F1.

   Пример, что нельзя сделать в Mash:
   ```
   MyClass -> Foo(123) -> Bar(456)
   ```

Пример того, что сделать на Mash можно:

```
SubClass ?= MyClass -> Foo(123)
SubClass -> Bar(456)
```

Возможно это ограничение немного глупое, но оно явно будет способствовать написанию простого и читаемого кода.

Вторая проблема, которую решает Mash - это его гибкость в вопросе расширения функционала нативными библиотеками и встраиваемость. Очень часто возникает потребность в реализации скриптов при написании какого-либо ПО. Встроить Mash в свое ПО очень просто, ровно также очень просто встроить свой функционал, написанный на Object Pascal или же C/C++ в Mash.
Это ооочень просто! И это я считаю крутым достижением!

Чтобы начать использовать Mash в своем проекте, как сценарный язык - вам достаточно скачать Mash, собранный для целевой ОС, затем написать пару строк кода на вашем языке и все, Mash готов к работе.

Даже Lua, считающийся самым простым и удобным для этого решением в сравнении с Mash в уступает по многим показателям.

Mash легко использовать, он может быть легко масштабирован вместе с вашим проектом, поддерживает запуск нескольких скриптов параллельно и многопоточность.

Вся мощь Mash SDK может стать частью вашего ПО за пару минут.

Mash - простое кроссплатформенное решение, подходящее для любых задач.

Пишем “Hello world!”

В отличии от многих других языков с динамической типизацией, все программы на Mash должны начинаться с метода main().

```
uses <crt>

proc main():
     println("Hello world!")
end
```

Следует сказать, что Mash не зависит от регистра символов.

Процедуры и функции

В Mash все методы делятся на процедуры (**proc**) - методы, которые не возвращают никаких значений через **return** и функции (**func**) - которые обязательно должны вернуть значение.

Пример процедуры вы уже видели выше. Пример функции:

```
func Summ(a, b):
     return a + b
end
```

Ключевое слово **return** нужно для возврата значений из функции. Это очевидно, но мне кажется, что я все-равно должен написать об этом.

Переменные

В первую очередь стоит уделить внимание одной из особенностей языка - неявным указателям.

Все переменные в Mash - это ячейки памяти, которые всего-лишь хранят указатель на объект в памяти. В отличии от Python, Ruby и других языков, в Mash работа с неявными указателями осуществляется явно. Странно звучит. Лучше

привести пример кода, где мы объявим переменную, выделим память под объект и положим указатель на него в нашу новую переменную:

```
x ?= 10
```

Все просто. Оператор ?= нужен для присваивания переменной **указателя** на значение.

Т.е. после выполнения например такого кода:

```
a ?= 10
b ?= a
b++
```

Переменная a будет равна 11.

Рассмотрим случай, когда нужно все-таки положить в переменную b копию значения из переменной a.

```
a ?= 10
b ?= copy(a)
b++
```

Здесь уже a = 10, b = 11.
Рассмотрим пример, когда нужно изменить значение объекта, указатель на который лежит в переменной.

```
a ?= 10
a = 20
```

Оператор = позволяет нам сделать это.

При передаче переменных в методы в качестве аргументов - мы передаем неявные указатели на них.
Это стоит учитывать при написании кода на Mash. И это пожалуй единственная сложная вещь в этом языке.

```
proc Foo(x):
    x = 10
```

```
end

. . .

x ?= 5
Foo(x)
println(x)
```

Выведет 10.

Если мы попытаемся присвоить значение через оператор = пустой переменной - это закончится выбросом Null-Pointer исключения (Access Violation / Segmentation Fault).

Для того, чтобы использовать = в переменной должно изначально быть хоть что-то. Следует написать a ?= new или a ?= null, перед такой операцией. Ну или просто использовать оператор ?= и проблем не возникнет.

Стоит также упомянуть про зоны видимости переменных. Переменные могут быть глобальными (объявленными вне методов) и локальными (внутри методов).

Глобальные переменные в Mash следует объявлять через ключевое слово **var**, пример:

```
var GlobalVar = 10,
    GlobalVar2 = "Hello",
    GlobalVar3
```

Если мы хотим объявить глобальную переменную и сразу инициализировать её, то мы должны использовать оператор =.

Стоит подметить, что при объявлении глобальных переменных мы можем вызывать функции и конструкторы классов. Да и в целом в Mash можно размещать почти любой код между методов и он будет выполнен перед входом в метод main(). Использовать такой подход я не рекомендую, для написания вашего кода, просто говорю что так можно. В целом эту механику я реализовал в языке для того, чтобы использовать её в основном при разработке SDK. Ну и иногда вы можете прибегать к ней. Но не следует размещать основную логику вашего приложения в глобальной зоне видимости.

Также можно подметить и то, что объявление переменных через **var** будет работать и внутри методов, как локальное объявление переменных, ровно также как и простое объявление переменных через оператор ?=.

Массивы

Массив в Mash - это массив указателей. Т.е. в Mash массивы могут включать в себя объекты с разными типами данных и могут быть не выровненными.

Объявлять массивы можно несколькими способами. Через **new**, перечисляя объекты или же через оператор задания диапазонов *“..”*.

Примеры:
```
arr1 ?= new[10][10] // создает пустой массив 10*10
arr2 ?= [1, 2, 3.14, “Hello”, [“New Array!”, 3, 4], 5]
arr3 ?= 1..100 //[1, 2, 3, .., 100]
```

Индексация элементов массива начинается с 0. На данном этапе стоит упомянуть строки. Они доступны для работы, как массивы, индексация также начинается с 0, но для того чтобы изменить какой-либо символ в строке, нужно использовать оператор ?=.

Пример:
```
s ?= “Hello!”
s[0] ?= “h”
```

Это связано с особенностями архитектуры SVM, на основе которой работает Mash.

Типы данных

В Mash определение типов данных и их изменение в процессе работы происходит автоматически (для простых типов данных).

Стоит сказать лишь, что поддерживается 8 типов данных:

целое число без знака, целое число со знаком, число с плавающей точкой, строки, массивы, классы, указатели (на нативные объекты), stream-ресурсы.

Константы и перечисления

Для объявления константы нужно явно указать тип данных: **word**, **int**, **real**, **str** или **stream**. Возможно в будущем все объявления констант будут производиться через ключевое слово **const**... Объявление констант выглядит вот так:

int MyConst1 -10
stream Res1 *"picture.png"*

stream - этот оператор позволяет добавить в ресурсы вашего Mash приложения какой-либо файл. (Да, у Mash приложений есть секция ресурсов). Работа с ресурсами должна производиться во внешних библиотеках, константа типа **stream** - это экземпляр класса TStream с загруженным содержимым файла.

Enum - перечисления. Объекты перечислений скрывают за собой целые числа без знака от 0 до N. Пример перечисления:
enum MyEnum [mFirst, mSecond, mThird]

Математические и логические операции

Здесь все практически также, как и в других языках программирования.
Поддерживаются основные математические операции:

1. + сложение
2. - вычитание
3. * умножение
4. / деление
5. \ деление нацело
6. % остаток от деления
7. ++ инкремент
8. -- декремент

Логические операции:

1. & логическое и
2. | логическое или
3. ^ логическое исключающее или
4. ~ логическое отрицание
5. << побитовый сдвиг влево
6. >> побитовый сдвиг вправо
7. == сравнение
8. > больше
9. < меньше
10. >= больше или равно
11. <= меньше или равно
12. <> не равно
13. **in** проверка принадлежности элемента к массиву

Быстрые присваивания:

1. +=
2. -=
3. *=
4. /=
5. \=
6. %=
7. &=
8. |=
9. ^=

Пример:

```
a = 10 * b - c
```

Условия if & else

Логические ветвления - это неотъемлемая часть любого языка, ровно также как и циклы.

```
if <условие>:
     . . .
end
if <условие>:
     . . .
else:
     . . .
end
```

Пример кода:

```
if a > b:
     c += 10
else:
     c -= 10
end
```

Цикл for и for each

Mash поддерживает циклы for и for each.
Цикл for задается такой вот конструкцией:

```
for(<объявление переменных>; <условие>; <изменения>):
    . . .
end
```

Пример:

```
for(i ?= 0; i < len(arr); i++):
    arr[i] ?= copy(i)
end
```

Цикл for each может быть задан двумя способами:

1. ```
   for i in arr:
       println(i)
   end
   ```

2. ```
   for i back arr:
       println(i)
   end
   ```

Разница в том, что при for in будет осуществлен пробег по массиву с начала к концу, а при for back - с конца к началу.

Циклы while & whilst

Это самые обычные циклы while, только у whilst - проверка условия осуществляется после итерации цикла.

```
while/whilst <условие>:
    . . .
end
```

Switch & Case

```
switch <значение>:
    case <значение 1>:
        . . .
    end

    case <значение 2>:
        . . .
    end

    . . .

    case <значение N>:
        . . .
    end

    else:
        . . .
end
```

(Лучше просто оставить здесь это и не давать лишних комментариев.)

Явные указатели

Иногда возникает необходимость в использовании указателей. Например может потребоваться хранить указатель на экземпляр класса, в одной из переменных класса.

Это довольно опасное решение, которое может привести к замыканию для сборщика мусора и как следствие - к утечке памяти.

У SVM сборщик мусора работает по принципу подсчета указателей (Reference Counting).

Эта проблема легко решается через явные указатели.

```
a ?= ptr(b)  // получить явный указатель на b
c ?= rel(a)  // получить обратно объект по указателю
c++          // -> a++
```

Явные указатели не вызывают зацикливаний. Утечек памяти не будет. В Mash они нужны по-сути только для этого.

Классы

Mash - это язык программирования, в котором очень легко можно построить довольно сложные абстракции. Объявление классов в Mash разделено на 2 части.

1. Описание класса, его структура и т.д.
2. Реализация его методов.

Сразу привожу пример:

```
class MyClass:
    private:
        var a, b, c

    public:
        proc create
        func summ
end

proc MyClass::Create(a, b, c):
    $a ?= a
    $b ?= b
    $c ?= c
end

func MyClass::Summ():
    return $a + $b + $c
end
```

Разберем его по порядку. Сначала идет объявление структуры класса, в котором перечисляются все переменные и методы, из которых состоит класс. Объявление класса может быть разделено на 3 зоны видимости: **private**, **protected** и **public**. Сразу стоит подметить, что зоны видимости в Mash - это всего-лишь условность, которая позволяет разработчику более ясно выразить свои мысли в описании класса и не более того.

Переменные у класса объявляются через **var**. В объявлении класса нельзя сразу инициализировать переменные

(возможно в будущем этот функционал будет добавлен в Mash).

Мы можем определить конструктор класса - это процедура с именем *“create”*. При создании экземпляра класса он может быть вызван.

Как вы могли заметить, реализация методов класса должна содержать в имени метода имя класса, за которым следует имя метода класса через *“::”*.

Что-ж... Мы объявили класс, теперь нужно его использовать в коде. Создадим экземпляр нашего класса:

```
x ?= new MyClass(10, 20, 30)
```

Все просто. Но как я сказал ранее, конструктор класса **может** быть вызван при объявлении, а может быть и не вызван. Пример:

```
x ?= new MyClass
```

Здесь конструктор уже вызван не будет. Мы можем вызвать его позже или же вручную инициализировать класс, как нам угодно.

Допустим что у нас есть x - инициализированный экземпляр класса MyClass. Как нам вызвать метод Summ() или обратиться к его полям?

Для этого в Mash есть оператор *“->”*. Пример:

```
println( x -> Summ() ) // Выведет 60
```

Стоит заметить кое-что. Любой вызов метода должен содержать (), как в C++ или других языках.

Стоит упомянуть про символ $ в методах класса. Как мы знаем, почти все методы класса взаимодействуют с экземпляром класса, от которого они вызываются. Т.е. методы класса должны иметь **this** - указатель на экземпляр класса. Mash это поддерживает и у каждого метода всегда есть возможность обратиться к **this**, пример: **this** -> a = 1. Но многократно писать “**this** ->” - не очень удобно, а Mash должен быть удобным языком. Символ $ - это сокращение для “**this** ->”.

Полиморфные методы для классов

Допустим у нас стоит задача - реализовать функцию, которая будет складывать две переменные класса - *a* и *b*. И есть два совершенно разных класса *Foo* и *Bar*, в которых есть поля *a* и *b*.

Что делать? Все просто - берем и реализовываем её:

```
func MyFunc_For_Foo_And_Bar(x):
    return x->a + x->b
end
```

В Mash не требуется явно указывать к какому типу данных относится переменная или класс, чтобы сделать с ней какое-либо действие. Класс переменной указывается лишь один раз - при создании экземпляра.

Статический полиморфизм (наследования)

Простая задача, для ООП языка - есть класс MyClass, нужно создать новый класс, который имеет на борту все то же, что и MyClass, но при этом расширяет (или заменяет некоторые) возможности MyClass.

```
class ExtendedMyClass(MyClass):
    func Average
end

proc ExtendedMyClass::Average():
    return ($a + $b + $c) / 3
end
```

Теперь мы можем объявить наш новый класс и использовать его новые возможности:

```
x ?= new ExtendedMyClass(10, 20, 30)
println( x -> Average() ) // Выведет 20
```

Но что если нам нужно переопределить при этом у класса метод Summ? Мы просто заново объявляем метод Summ, но уже в новом классе:

```
class ExtendedMyClass(MyClass):
    func Summ
end

proc ExtendedMyClass::Summ(x):
    return $a + $b + $c + x
end
```

Готово. Точно также мы можем поступить и с конструктором.

Мы можем создать класс, который будет наследником от двух или более классов:

```
class ExtendedMyClass(MyClass, AnotherClass, ThirdClass):
    . . .
end
```

В таком случае нам лучше определить новый конструктор у нового класса. Наследование - это конечно хорошо, но неопределенное поведение нам не нужно.

Рефлексия

Что если нам нужно переопределить метод, тот же Summ у z - уже созданного экземпляра MyClass и у нас нет желания объявлять новый класс, пересоздавать z и т.д., ради переопределения одного метода у одного экземпляра класса?

1. Объявим для начала новый метод Summ, который должен принадлежать какому-либо классу:

   ```
   func class::Summ(x):
       return $a + $b + $c + x
   end
   ```

2. Теперь просто возьмем и переопределим его:

   ```
   z -> Summ ?= class::Summ
   ```

Да, все так просто. Теперь при вызове z -> Summ(123) будет вызван наш новый метод. Переопределение будет работать для этого экземпляра класса.
Стоит предупредить, что использовать оператор = здесь нельзя (в целях оптимизации было реализовано довольно рискованное решение для Virtual Class Table’s, которое допускает изменение константных значений и может привести к неопределенному поведению).

Интроспекция

Иногда в языках с динамической типизацией может потребоваться узнать тип объекта. В Mash есть реализация RTTI для классов и простых объектов.

Для классов:

```
if x -> type == MyClass:
     . . .
end
```

Для простых типов данных есть функция typeof().

Пример:

```
if typeof(x) in [TypeWord, TypeInt, TypeReal]:
     . . .
end
```

Сборщик мусора

В Mash сборщик мусора работает по принципу подсчета указателей (Reference Counting). Для сбора мусора его нужно вызывать с помощью gc().

Многопоточность

Многопоточность - неотъемлемая часть современных языков программирования. И Mash конечно же поддерживает её. В отличии от Python у SVM нет GIL, она использует системные потоки. Для синхронизации потоков в Mash есть поддержка критических секций и атомарных типов.

Использовать многопоточность можно несколькими способами.

1. Быстрый запуск выполнения кода в отдельном потоке:
   ```
   launch:
       . . .
   end
   ```

   Пример:
   ```
   for i in 1..10:
       launch:
           Sleep(1000 - i * 100)
           println(i)
       end
   end
   ```

2. Быстрый запуск выполнения кода в отдельном потоке с возможностью синхронизации:
   ```
   async <name>:
       . . .
   end
   ```

   Пример:
   ```
   async a:
       Foo()
   end

   async b:
       Bar()
   end

   wait a, b
   ```

3. Используя класс TThread:
   ```
   class MyThr(TThread):
   ```

```
    proc Execute
end

proc MyThr::Execute():
    Foo()
end
. . .
thr ?= new MyThr(false)
```

4. Используя Async() и Thread().

```
proc MyThreadProc(a, b, c):
    . . .
end
. . .
thr ?= Async(MyThreadProc, 10, 20, 30)
или:
thr ?= Thread(MyThreadProc, 10, 20, 30)
. . .
thr -> Resume()
```

Важное замечание - каждый поток имеет свой контекст, в т.ч. пространства глобальных переменных. Это значит, что если нужно изменить глобальную переменную так, чтобы у всех потоков она также поменялась - нужно изменять её по значению.

Вы можете передавать потокам свои переменные и экземпляры классов при создании.

Исключения

Для перехвата и обработки исключений в Mash предусмотрена конструкция:

```
try:
    . . .
catch E:
    . . .
finally:
    . . .
end
```

Поля **finally** и **catch** не обязательны, т.е. код будет работать и без них. Однако я рекомендую всегда обрабатывать исключения через **catch** блок.

В **catch** блок передается экземпляр класса исключения (Exception)

Чтобы вызвать искусственное исключение в Mash есть оператор **raise** <объект>.

Пример:

```
try:
    a ?= 0
    b ?= 10 / a
catch E:
    println(E)
end
```

Вывод:

```
Exception <EZeroDivide> with message: 'Floating point division by zero'.
```

Пример использования **raise**:

```
raise new Exception(“Exception description”)
```

Поддержка рекурсии

Из-за особенностей SVM, Mash не поддерживает рекурсию в привычном виде.

Для того, чтобы вызвать метод рекурсивно, нужен оператор **safe**.

Пример:

```
func Foo(x):
     . . .
     safe t ?= Foo(x + 1)
     . . .
end
```

Методы с переменным числом аргументов

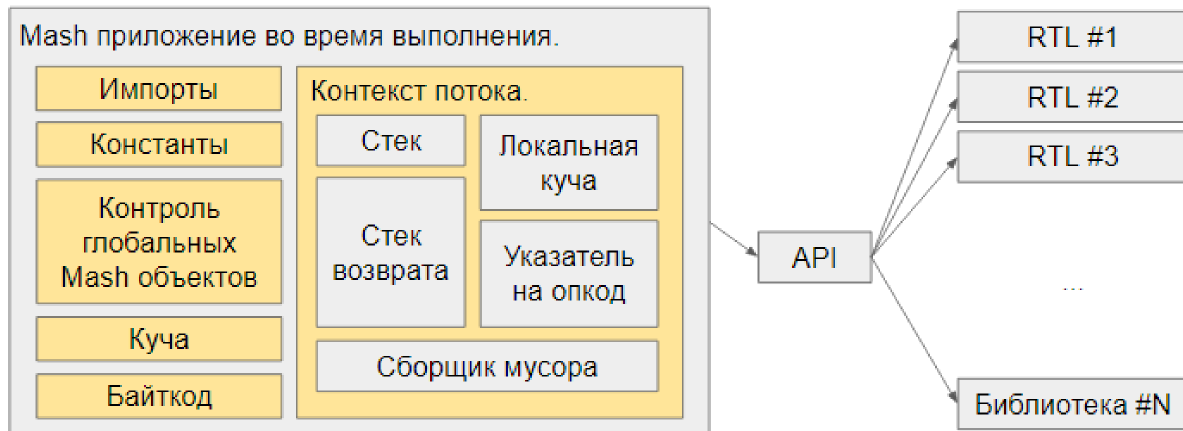
Сразу пример:

```
func Foo(a, b, [another]):
     r ?= a + b
     for i in another:
          r += i
     end
     return r
end
```

Для того, чтобы принять переменное число аргументов, в объявлении метода последний из аргументов должен быть заключен в [...] - методу Foo в данном случае будут переданы a, b и остальные аргументы в виде массива another.

SVM - Архитектура ВМ

SVM (Stack-based Virtual Machine) - ВМ со стековой архитектурой, как вы уже догадались.
(Да, верно, за 2 года я так и не дал названия этой части проекта...)



(Рассматривать архитектуру SVM будет проще, если перед глазами будет эта схема.)

В архитектуре SVM ваши приложения выполняются в SVM-потоках, которые имеют свою внутреннюю архитектуру.

Помимо потоков выполнения кода у SVM есть менеджер секции импортов для нативных методов из библиотек, менеджер секции ресурсов (константы), массив инструкций (байткод) и стоит отметить, что все SVM-объекты аллоцируются в куче (heap), а не на стеке и затем сохраняются в массив статической адресации.

Под контролем глобальных Mash объектов подразумевается контроль SVM-объектов в памяти, которые не могут быть привязаны к одному конкретному потоку. Т.е. например вы создали поток, создали внутри него несколько экземпляров классов и убили поток, вернув при этом указатели на экземпляры классов потоку-родителю или другому потоку.

Если при смерти потока освобождать всю занимаемую им память - это приведет к Access violation/Segmentation fault ошибкам.

SVM - Архитектура потока

Поток SVM имеет стек указателей общего назначения, стек точек возврата (для выполнения call & return инструкций), указатель на текущую инструкцию (Instruction Pointer), а также свой массив статической адресации.

SVM - Пишем свою библиотеку

Для того, чтобы написать и начать использовать свою нативную библиотеку нужно описать импорты из svm.lib на нативном ЯП, на котором вы собираетесь её написать.
Если вы хотите использовать для написания библиотеки FPC/Delphi, то все что вам нужно для старта - это подключить к проекту svm.inc и написать простой метод.
Пример:

```
library MyLib;
{$I ‘svm.inc’}

procedure MyFunc(pctx: pointer); stdcall;
begin
  __Return_Word(pctx, __Next_Word(pctx) * 10);
end;

exports MyFunc name “MyFunc”;

begin
end.
```

Использование:

```
import MyFunc “MyLib.lib” “MyFunc”

proc main():
     println( MyFunc(10) )
end
```

Аргументы принимаются через API по порядку.

SVM - Встраиваем Mash в своё ПО

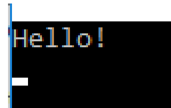
Для этого опять же нужно описать импорты из svm.lib на нужном вам языке. Мы подключаем svm.inc и смотрим что у нас получится:

```
program MySoftware;

{$I 'svm.inc'}

procedure TestFunc(pctx: pointer); stdcall;
begin
  writeln(__Next_String(pctx));
  readln;
end;

var
  vm: pointer;
  fpath: string;
begin
  SVM_Init;

  vm := SVM_CreateVM;

  SVM_RegAPI(vm, @TestFunc);

  fpath := 'test.vmc';
  SVM_LoadExeFromFile(vm, @fpath);

  SVM_Run(vm);

  SVM_FreeVM(vm);
  SVM_Free;
end.
```

Код на Mash:

```
regapi TestFunc 0

proc main():
     TestFunc("Hello!")
end
```

Компилируем Mash приложение...
Компилируем наше ПО...
Запускаем:



Радуемся

SVM - Некоторые особенности

В процессе разработки SVM я выяснил, что выбранный мной из-за своей кроссплатформенности FPC, на ОС Windows 32/64 плохо поддерживает возврат исключений из dll. Т.е. пришлось реализовать свой векторный обработчик исключений (SEH/VEH), конкретно для Windows версии SVM.

Он полностью рабочий, однако т.к. мануалов на эту тему для FPC почти нигде нет - может работать с багами.

SVM - Формат исполняемых файлов

<table>
<tr><td>Название секции</td><td>Размер в байтах</td></tr>
<tr><td>Заголовок SVM формата</td><td>0 или 10</td></tr>
<tr><td>Размер секции описания библиотек</td><td>N = 2 (word)</td></tr>
<tr><td>Записи вида:
<table><tr><td>Размер X (2 байта)</td><td>Имя библиотеки
(X байт)</td></tr></table></td><td>N</td></tr>
<tr><td>Размер секции описания импортов</td><td>N = 4 (cardinal)</td></tr>
<tr><td>Записи вида:
<table><tr><td>Номер
библиотеки
(2 байта)</td><td>Длина
метода X
(1 байт)</td><td>Имя метода
(X байт)</td></tr></table></td><td>N</td></tr>
<tr><td>Размер секции ресурсов</td><td>N = 4 (cardinal)</td></tr>
<tr><td>Записи вида:
<table><tr><td>Флаг типа
(1 байт)</td><td>Значение</td></tr></table></td><td>N</td></tr>
<tr><td>Исполняемый код</td><td>Остальное</td></tr>
</table>

SVM API

Теперь рассмотрим все API функции, которые вам доступны из svm.lib (Все нужные импорты описаны в svm.inc, версии на других ЯП появятся чуть позже).

Для начала рассмотрим API, который позволяет использовать Mash в качестве скриптового языка для вашего ПО.

Первым делом стоит упомянуть про методы инициализации и освобождения памяти, если Mash выполнил поставленные задачи и больше вам не нужен. Эти два метода глобальны для svm.lib и поэтому должны вызываться по 1-му разу:

```
procedure SVM_Init; stdcall;
procedure SVM_Free; stdcall;
```

После инициализации библиотеки - нам нужно создать экземпляр ВМ, для его создания и освобождения памяти отвечают следующие методы:

```
function  SVM_CreateVM(CustomArgs: boolean = false):Pointer; stdcall;
procedure SVM_FreeVM(SVM:Pointer); stdcall;
```

Зарегистрировать методы своего ПО можно через RegAPI:

```
procedure SVM_RegAPI(SVM:Pointer; ExtFunc:Pointer); stdcall;
```

Для запуска Mash приложения написаны 2 API метода:

```
procedure SVM_Run(SVM:Pointer); stdcall;
procedure SVM_LoadExeFromFile(SVM:Pointer; MainClassPath:PString); stdcall;
```

Для расширения функционала Mash (написания библиотек и интеграции с вашим ПО) реализованы следующие методы, для работы со стеком (получение аргументов и возврат результатов):

```
function  __Next_Type(SVM: Pointer): byte; stdcall;
function  __Next_Word(SVM: Pointer): longword; stdcall;
function  __Next_Int(SVM: Pointer): int64; stdcall;
function  __Next_Float(SVM: Pointer): double; stdcall;
procedure __Next_String(SVM: Pointer; str: PString); stdcall; overload;
function  __Next_String(SVM: Pointer): string; overload;
function  __Next_Bool(SVM: Pointer): boolean; stdcall;
function  __Next_Ref(SVM: Pointer): pointer; stdcall;
function  __Next_Object(SVM: Pointer): pointer; stdcall;

procedure __Return_Word(SVM: Pointer; val: longword); stdcall;
procedure __Return_Int(SVM: Pointer; val: int64); stdcall;
procedure __Return_Float(SVM: Pointer; val: double); stdcall;
procedure __Return_String(SVM: Pointer; val: PString); stdcall; overload;
```

```
procedure __Return_String(SVM: Pointer; val: string); overload;
procedure __Return_Bool(SVM: Pointer; val: boolean); stdcall;
procedure __Return_Ref(SVM: Pointer; val: pointer; dcbp: pointer); stdcall;
procedure __Return_Object(SVM: Pointer; val: pointer); stdcall;


procedure __Make_CallBack(SVM: Pointer; addr: cardinal); stdcall;

function  __New_Set(SVM: Pointer): pointer; stdcall;
procedure __Push_To_Set(pSet: TSVMMem; Ptr: pointer); stdcall;

function __New_Word(pVM: PSVM; val: longword): pointer; stdcall;
function __New_Int(pVM: PSVM; val: int64): pointer; stdcall;
function __New_Float(pVM: PSVM; val: double): pointer; stdcall;
function __New_String(pVM: PSVM; val: PString): pointer; stdcall; overload;
function __New_String(SVM: Pointer; val: string): pointer; overload;
function __New_Bool(pVM: PSVM; val: boolean): pointer; stdcall;
function __New_Ref(pVM: PSVM; val: pointer; dcbp: PDestructorCallBack): pointer;
stdcall;
```

Mash при вызове помещает аргументы в стек в обратном порядке.

Следующие API методы необходимы для работы отладчика:

```
procedure SVM_CheckErr(SVM:Pointer; E:Exception); stdcall;
procedure SVM_Continue(SVM:Pointer); stdcall;
procedure SVM_SetDbgCallBack(SVM:Pointer; DbgCB:Pointer); stdcall;
```

Дополнительная информация

Сайт: mash-lang.tech
Форум: forum.mash-lang.tech
Репозиторий: github.com/RoPi0n/mash-lang
Хабр: @RoPi0n

Автор:
Чернов П. С. - Студент НГТУ.

Руководитель:
Исаева Е. В. - Кафедра высшей математики НГТУ.

Проект не преследует коммерческих целей, разработка ведется исключительно на моей инициативе.
Единственные цели проекта - это создание отличного языка программирования и повышение моих скиллов разработчика.

Однако... Ссылка на Patreon пусть будет тут...
Patreon: www.patreon.com/RoPi0n