



Использование Lua в Рабочем месте QUIK



Содержание

Термины и сокращения	3
1. Возможные подходы написания скриптов Lua для плагина QLua в Рабочем месте QUIK	3
2. Взаимодействие потоков Lua скрипта	9
3. Создание индикаторов технического анализа с помощью скриптов Lua	14
3.1 Как устроены индикаторы в QUIK.....	14
3.2 Минимальный код индикатора	14
3.3 Изменение свойств индикатора	16
3.4 Рисование прямой линии	17
3.5 Подсчёт среднего	18
3.6 Доступ к данным	20
3.7 Расчёт EMA	22
3.8 Индикатор с несколькими линиями.....	26
3.9 Функция OnDestroy	27
3.10 Magician birthday	28
4. Отправка транзакций из Lua скрипта	31
5. Использование функции PrintDbgStr() на практике	41

Ваши пожелания и комментарии к данному Руководству
направляйте по электронной почте на адрес: quiksupport@arqatech.com



Термины и сокращения

- **Плагин QLua** – интерпретатор языка Lua, реализованный в Рабочем месте QUIK, работу которого обеспечивает библиотека qlua.dll.
- **Скрипт Lua** – файл с расширением .lua или .luac (скомпилированный в байт-код скрипт Lua) содержащий сценарии, написанные с помощью языка программирования Lua, интерпретируемые плагином QLua.
- **РМ QUIK** – Рабочее место QUIK.

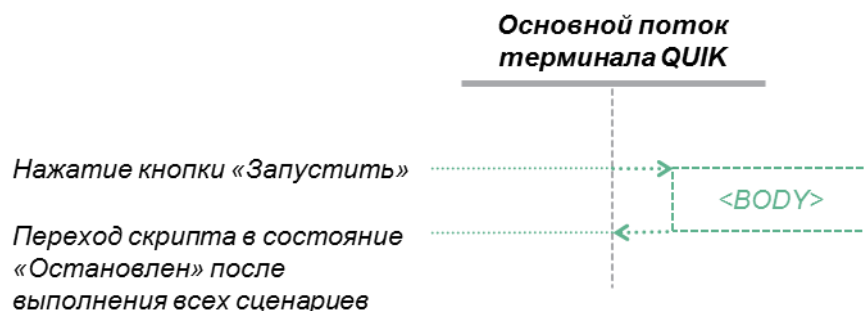
1. Возможные подходы написания скриптов Lua для плагина QLua в Рабочем месте QUIK

Плагин QLua обрабатывает скрипты Lua в кодировке ASCII. Для корректного отображения кириллицы в РМ QUIK рекомендуется кодировка Windows-1251 (Cp1251).

Запуск скрипта Lua выполняется в диалоге «Доступные скрипты» РМ QUIK (меню **Сервисы / Lua скрипты...**). Для запуска добавьте необходимый скрипт и нажмите на кнопку «Запустить». Запуск скрипта всегда начинается с обработки тела скрипта вне каких-либо функций, обозначим его <BODY>. Можно выделить три подхода при создании Lua скрипта:

1. **Вся необходимая логика описывается в области <BODY>.** В этом случае сценарии, описанные в теле скрипта вне каких-либо функций, после запуска выполняются только один раз, и скрипт переходит в состояние «Остановлен». Данный подход применим для Lua скриптов, целью которых является разовый подсчет необходимых данных. Скрипты с такой структурой выполняются в основном потоке РМ QUIK, таким образом, необходимо иметь в виду, что время работы сценариев в скриптах данного типа должно быть сравнительно небольшим, а также не рекомендуется использование функции **sleep()**, иначе будут заметны «подвисания» в работе РМ QUIK.

Схема выполнения Lua скрипта с данным подходом:



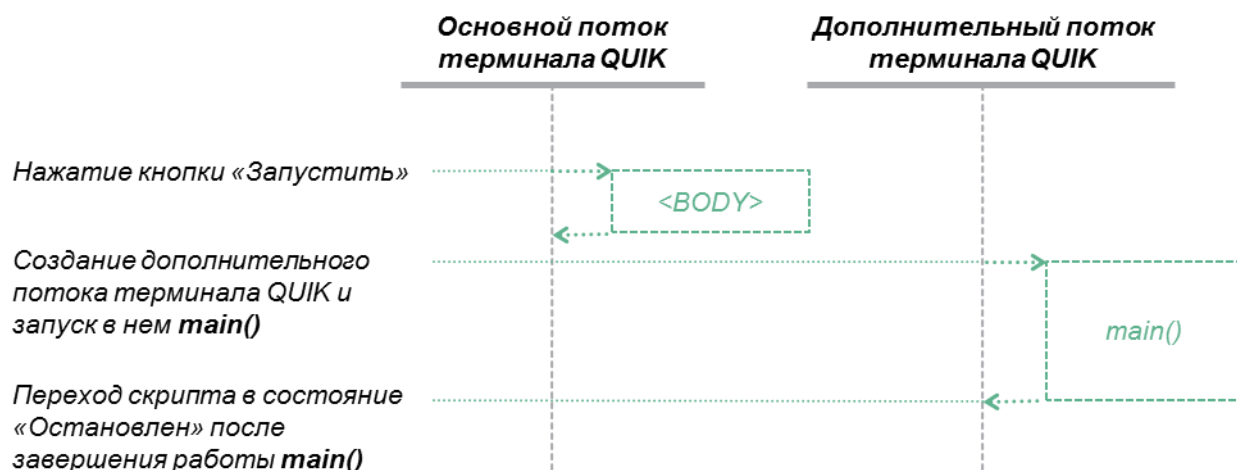
Пример Lua скрипта, выводящего сообщение с количеством сделок:

NumberOfTrades.lua

```
--Вывод сообщения с количеством сделок
--<BODY
number_of_trades = getNumberOf("trades")
message("Общее количество сделок: " .. number_of_trades)
--BODY>
```

2. Вся необходимая логика описывается в функции с предопределенным именем **main()**. В этом случае после запуска скрипта первоначально выполняются сценарии, описанные в <BODY>, если они присутствуют. Далее в отдельном потоке выполняется функция **main()**.

Схема выполнения Lua скрипта с данным подходом:



Во время выполнения функции **main()** Lua скрипт не мешает работе основного функционала РМ QUIK, таким образом, внутри функции **main()** использование функции **sleep()** не приводит к «подвисанию» РМ QUIK и позволяет периодически приостанавливать скрипт и возобновлять его работу через некоторый промежуток времени. Скрипт считается работающим, пока работает функция **main()**. При завершении работы функции **main()** скрипт переходит в состояние «Остановлен». При принудительной остановке скрипта нажатием кнопки «Остановить» ему дается 5 секунд на завершение работы.

При принудительной остановке скрипта возможна потеря системных ресурсов, так как функция main() завершается принудительно.

Пример Lua скрипта, выводящего сообщение с количеством сделок каждые 60 секунд:

NumberOfTradesEvery60Sec.lua



```

--Вывод сообщения с количеством сделок через каждые 60 секунд
--при наличии подключения к серверу

--<BODY
message("<BODY>", 2)
message("Скрипт запущен.")
--BODY>

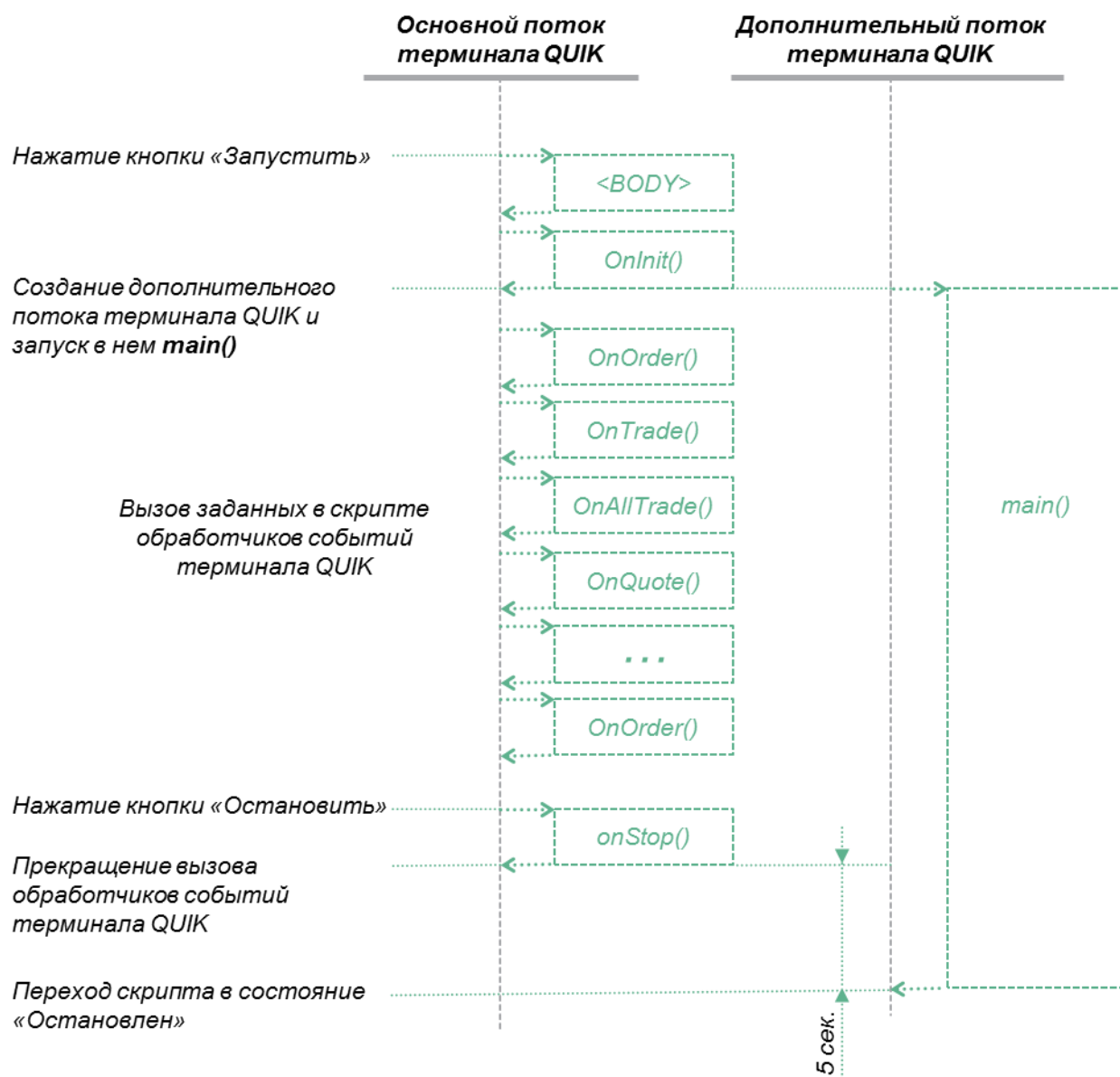
function main()
    message("Уже main()", 2)
    while isConnected() == 1 do
        number_of_trades = getNumberOf("trades")
        message("Общее количество сделок: " .. number_of_trades)
        sleep(60000)
    end
end

--<BODY
message("Здесь тоже <BODY>", 2)
--BODY>

```

- 3. Событийная модель.** При выборе данного подхода предоставляется гибкая среда выполнения пользовательских сценариев внутри QUIK, позволяющая мгновенно получать интересующие события от РМ QUIK, производя нужную обработку этих событий. Для обработки того или иного события необходимо в скрипте прописать функцию с predetermined названием. Описание данных функций приведено в разделе 2.2 «Функции обратного вызова» Руководства пользователя Интерпретатора языка Lua. Скрипт Lua может содержать несколько функций с predetermined названиями, являющимися обработчиками событий, таких как новая сделка, новая обезличенная сделка, изменение котировок и т.д.
- Схема выполнения Lua скрипта с данным подходом:





Как и в предыдущей структуре скрипта Lua, после его запуска первоначально выполняются сценарии, описанные в `<BODY>`, если они присутствуют. Далее происходит вызов обработчика с именем **OnInit()**, если он присутствует. В обработчике **OnInit()** пользователь имеет возможность инициализировать все необходимые переменные и библиотеки перед запуском отдельного потока. После завершения функции **OnInit()** происходит создание отдельного потока PM QUIK, и в этом потоке начинается выполнение функции **main()**, которая обязательно должна присутствовать в скрипте. Скрипт считается работающим, пока работает функция **main()**. При завершении работы функции **main()** скрипт переходит в состояние «Остановлен».

Все функции обработки событий, в отличие от функции **main()**, выполняются в рамках основного потока PM QUIK, поэтому пользователю необходимо оптимизировать время исполнения таких функций, чтобы время их работы было сравнительно небольшим, иначе будут заметны «подвисания» в работе PM QUIK.



Пример структуры скрипта Lua:

```
--минимальная структура скрипта при использовании событийной модели
is_run = true

function OnStop()
    is_run = false
end

function main()
    while is_run do
        sleep(100)
    end
end
```

В вышеописанном примере мы объявляем глобальную переменную логического типа **is_run**, которая имеет значение **true** до момента нажатия кнопки «Остановить» в диалоге «Доступные скрипты» РМ QUIK. В функции **main()** выполняется цикл, который через каждые 100 миллисекунд проверяет состояние переменной **is_run**. Если переменная имеет значение **false**, цикл завершается и, соответственно, завершает работу функция **main()**, что приводит к переходу скрипта в состояние «Остановлен». Задержка в 100 миллисекунд внутри цикла функции **main()** приведена для примера, ее значение можно увеличить или снизить при необходимости. Вызов функции **sleep()** внутри цикла потока **main()** обязателен. Значение аргумента функции **sleep()** может быть равно 1, но в этом случае задержка приближена к значению прерывания системного таймера (по умолчанию 15.625мс).

При остановке скрипта нажатием кнопки «Остановить» ему дается 5 секунд на завершение работы.

При принудительной остановке скрипта возможна потеря системных ресурсов, так как функция main() завершается принудительно.

Также необходимо иметь в виду, что функция **OnStop()** выполняется в основном потоке РМ QUIK, и, так как в момент ожидания завершения скрипта (5 секунд) он занимает основной поток РМ QUIK, то другие обработчики событий уже не вызываются, а само РМ QUIK может некоторое время находиться в «подвисшем» состоянии. При необходимости изменить таймаут, отведенный скрипту на завершение работы, в обработчике **OnStop()** укажите новое значение в миллисекундах, возвращаемое данной функцией. Пример Lua скрипта, выводящего сообщение с количеством сделок при каждом получении/изменении сделки:

NumberOfTradesOnCallbacks.lua

```
--Вывод сообщения с количеством сделок при каждом получении/изменении сделок
```



```

--<BODY
message("Скрипт запущен.")
--BODY>

function OnInit(script)
    message("OnInit()", 2)
    is_run = true
end

function OnTrade(trade)
    message(string.format("Обработка сделки №%i по инструменту %s [%s]",
                        table_of_trades[1].trade_num,
                        table_of_trades[1].sec_code,
                        table_of_trades[1].class_code))
    number_of_trades = getNumberOf("trades")
    message("Общее количество сделок: " .. number_of_trades)
end

function OnStop()
    message("OnStop()", 2)
    is_run = false
    return 2000
end

function main()
    message("main()", 2)
    while is_run do
        sleep(100)
    end
end

--<BODY
message("Здесь тоже <BODY>", 2)
--BODY>

```



2. Взаимодействие потоков Lua скрипта

При использовании событийной модели Lua скрипт выполняется в двух потоках: функции обратного вызова выполняются в основном потоке PM QUIK, а функция **main()** в дополнительном потоке PM QUIK (подробнее см. п. [1](#)). При этом для предотвращения «подвисаний» PM QUIK необходимо каким-либо образом оптимизировать сценарии, описанные в функциях обратного вызова. Одним из способов такой оптимизации является перенос логики обработки полученных сигналов в функцию **main()**. Данный подход сводит количество сценариев в функции обратного вызова до одного, а именно добавление в глобальную Lua таблицу (очередь) записи о том, что функция сработала и вернула определенные значения. Таким образом, мы получаем очередь событий, которые необходимо обработать в другом потоке.

Пример реализации очереди FIFO («первым пришёл — первым ушёл») для обработки срабатывания функций обратного вызова в функции **main()**:

```
--Обработка событий PM QUIK в функции main() посредством очереди FIFO

function OnInit(script)
    is_run = true
    MAIN_QUEUE = {}
end

function OnOrder(order)
    table.insert(MAIN_QUEUE, {callback = "OnOrder", value = order})
end

function OnTrade(trade)
    table.insert(MAIN_QUEUE, {callback = "OnTrade", value = trade})
end

function OnAllTrade(all_trade)
    table.insert(MAIN_QUEUE, {callback = "OnAllTrade", value = all_trade})
end

function OnQuote(class_code, sec_code)
    local quote = getQuoteLevel2(class_code, sec_code)
    table.insert(MAIN_QUEUE, {callback = "OnQuote", value = quote})
end

function OnStop()
    is_run = false
    return 2000
end
```



```

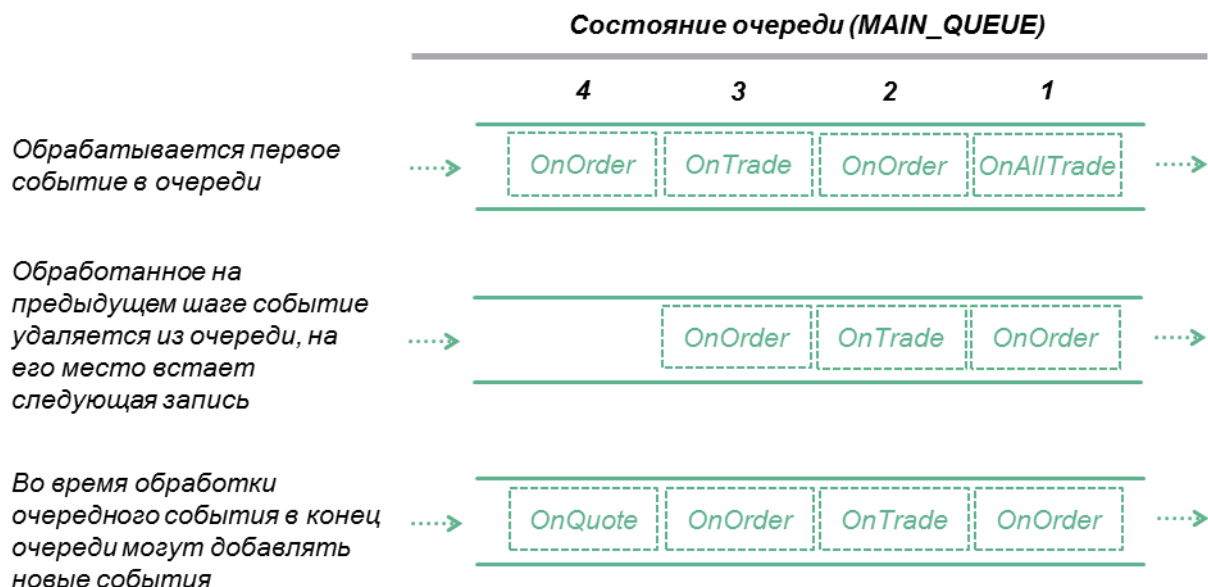
function main()
    while is_run do
        if #MAIN_QUEUE > 0 then
            ProcessingCallbakc(MAIN_QUEUE[1])
            table.sremove(MAIN_QUEUE, 1)
            message("Размер очереди " .. tostring(#MAIN_QUEUE))
        end
        sleep(1000)
    end
end

function ProcessingCallbakc(value)
    message(string.format("Обработка события %s начата", value.callback))
    sleep(3000) --эмуляция продолжительного алгоритма обработки события
    message(string.format("Обработка события %s завершена", value.callback))
end

```

В данном примере используются потокобезопасные функции работы с таблицами Lua **sinsert()** и **sremove()**, которые предоставляет плагин QLua. Формат вызова потокобезопасных функций совпадает с форматом вызова стандартных функций Lua **insert()** и **remove()**, но при выполнении потокобезопасной функции блокируется выполнение кода в другом потоке до окончания работы вызываемой функции. Таким образом, исключаются ситуации коллизий, когда два потока одновременно вносят изменения в одну и ту же таблицу Lua, что может приводить к неопределенным ситуациям. Из примера видно, что во время инициализации дополнительного потока РМ QUIK объявляется глобальная Lua таблица **MAIN_QUEUE** (имя переменной может быть любым), в которую и будут записываться последовательно все вызовы событий при срабатывании функций обратного вызова. В дополнительном потоке в функции **main()** отслеживается размер таблицы **MAIN_QUEUE**, и, если таблица не пуста, выполняется обработка очереди событий. После обработки события запись из очереди удаляется, при этом во время обработки записи с номером «1» в конец очереди беспрепятственно могут добавляться новые записи при срабатывании функций обратного вызова. После удаления записи с номером «1», все нижестоящие записи сдвигаются на 1 шаг, и запись с номером «2» становится записью с номером «1», и таким образом выполняется обработка всей очереди. По окончании обработки всей очереди скрипт продолжает работать и ожидает новых событий в очереди. Схематично очередь FIFO, описанную в примере, можно представить следующим образом:





Таким образом, используя данный подход, мы разгружаем основной поток PM QUIK и переносим все сложные алгоритмы обработки событий PM в дополнительный поток. При этом ни одно событие не будет потеряно и будет обработано в порядке общей очереди. Если необходимо создавать очередь, отфильтровывая только определенные события, например изменение котировок только по определенному инструменту или классу, то в обработке события можно добавить условие, которое будет следить, что в очередь попадают только необходимые события, а ненужные пропускаются. При необходимости выделения приоритетных событий нужно выделить для них отдельную очередь и первоначально выполнять ее обработку, а после – все остальные.

Пример очереди FIFO с выделением приоритета событию получения/изменения сделки и фильтрации изменения котировок только по определенному классу:

```
--Обработка событий PM QUIK в функции main() посредством очереди FIFO
--с выделением приоритета OnTrade и фильтрации OnQuote

function OnInit(script)
    is_run = true
    MAIN_QUEUE = {}
    MAIN_QUEUE_TRADES = {}
end

function OnOrder(order)
    table.insert(MAIN_QUEUE, {callback = "OnOrder", value = order})
end

function OnTrade(trade)
    table.insert(MAIN_QUEUE_TRADES, trade)
```



```

end

function OnAllTrade(all_trade)
    table.sinsert(MAIN_QUEUE, {callback = "OnAllTrade", value = all_trade})
end

function OnQuote(class_code, sec_code)
    if class_code == "SPBFUT" then
        local quote = getQuoteLevel2(class_code, sec_code)
        table.sinsert(MAIN_QUEUE, {callback = "OnQuote", value = quote})
    end
end

function OnStop()
    is_run = false
    return 2000
end

function main()
    while is_run do
        if #MAIN_QUEUE > 0 and #MAIN_QUEUE_TRADES == 0 then

            ProcessingCallbakc(MAIN_QUEUE[1])
            table.sremove(MAIN_QUEUE, 1)
            message("Размер общей очереди " .. tostring(#MAIN_QUEUE))

        elseif #MAIN_QUEUE_TRADES > 0 then

            ProcessingOnTrade(MAIN_QUEUE_TRADES[1])
            table.sremove(MAIN_QUEUE_TRADES, 1)
            message("Размер очереди сделок " .. tostring(#MAIN_QUEUE_TRADES))

        end
        sleep(1000)
    end
end

function ProcessingCallbakc(value)
    message(string.format("Обработка события %s начата", value.callback))
    sleep(3000) --эмуляция продолжительного алгоритма обработки события
    message(string.format("Обработка события %s завершена", value.callback))
end

function ProcessingOnTrade(trade)
    message(string.format("Обработка сделки №%s начата", trade.trade_num))
    sleep(3000) --эмуляция продолжительного алгоритма обработки сделки
    message(string.format("Обработка сделки №%s завершена", trade.trade_num))
end

```



```
end
```

В данном примере мы добавили новую Lua таблицу **MAIN_QUEUE_TRADES**, которая предназначена для создания очереди события добавления/изменения сделки, а в функции **main()** выделили приоритет для данной очереди, добавив условие обработки общей очереди **MAIN_QUEUE** только в том случае, если очередь сделок пуста. Также добавили для события **OnQuote()** фильтрацию по классу инструментов, и таким образом мы уменьшили очередь **MAIN_QUEUE**, не добавляя в нее ненужные нам события.



3. Создание индикаторов технического анализа с помощью скриптов Lua

В данном разделе рассматривается процесс создания индикатора на примере «скользящей средней» (Moving Average).

3.1 Как устроены индикаторы в QUIK

Основой для построения всех индикаторов в QUIK является источник данных (далее ИД). ИД представляет собой массив, в котором элементы являются структурами и имеют 6 полей:

1. Open;
2. High;
3. Low;
4. Close;
5. Volume;
6. Time.

Фактически это значит, что все элементы массива в источнике данных представляют собой свечи. В случае тиковых данных поля с 1-го по 4-е будут иметь одно значение, совпадающее со значением параметра в этот момент времени. Источники данных могут быть интервальными графиками (тики, 1 минута, 5 минут и т.д.), рассчитанными по Таблице обезличенных сделок или по изменениям параметра торгуемого инструмента.

Индикатор представляет собой функцию, которая для элемента массива ИД может вернуть одно или несколько чисел, в зависимости от количества линий, отображаемых на графике.

Индикатор не может выступать источником данных для другого индикатора.

3.2 Минимальный код индикатора

Пример 1 (ex1.lua)

```
Settings=
{
    Name = "Example1"
}
function Init()
    return 1
end

function OnCalculate(index)
    return nil
end
```

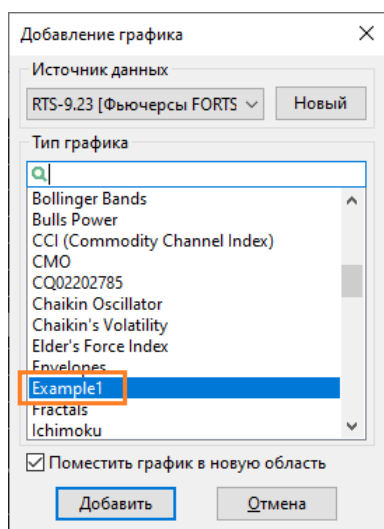


Рассмотрим подробнее, что происходит при добавлении такого индикатора на график.

При создании нового индикатора (пункт **Добавить график (индикатор)**... контекстного меню графика) PM QUIK сканирует папку LuaIndicators в директории PM QUIK на наличие в ней скриптов, отвечающих следующим требованиям:

1. В скрипте определена глобальная таблица Lua с именем Settings;
2. Определена функция Init();
3. Определена функция OnCalculate().

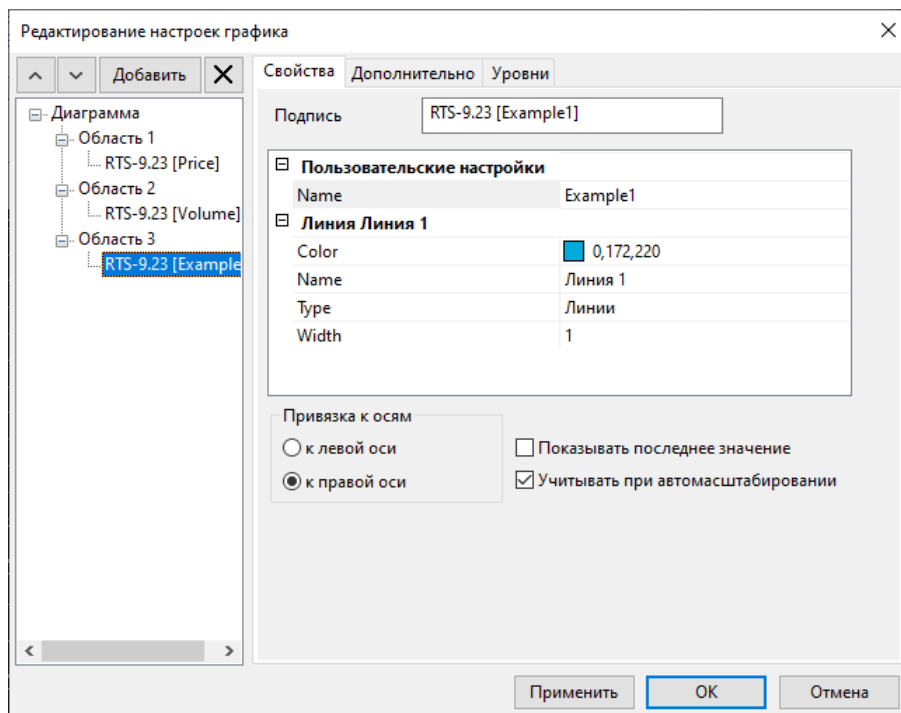
Поле Name в таблице Settings определяет имя индикатора, с которым он отображается в диалоге:



Если скрипт не удовлетворяет перечисленным выше требованиям или содержит синтаксические ошибки языка Lua, то он не отображается в данном диалоге.

При выборе индикатора Example1 и нажатии кнопки «Добавить» индикатор добавляется на график. Для просмотра его настроек необходимо выбрать пункт контекстного меню графика «Редактировать...»:





Как видно на рисунке выше, значение поля `Settings.Name` попало в подпись нового индикатора и отображается в поле `Name` группы «Пользовательские настройки». Также в диалоге свойств индикатора присутствуют параметры одной линии с именем «Линия 1». Функция `Init` вернула «1», это говорит PM QUIK, что индикатор будет состоять из одной линии. Так как параметры этой линии в коде не описаны (см. пример выше), то значения полей `Color`, `Name`, `Type`, `Width` инициализируются значениями по умолчанию.

На графике не появляются новые линии, так как функция `OnCalculate` всегда возвращает `nil`. Это значение говорит PM QUIK, что значение индикатора для указанной свечи источника данных не определено.

3.3 Изменение свойств индикатора

В качестве примера рассмотрим код:

```
Settings=
{
    Name = "Example2",
    period = 5,
    line =
    {
        {
            Name = "MA",
            Color = RGB(255, 0, 0),
            Type = TYPE_LINE,
            Width = 2
        }
    }
}
```




```

}
function Init()
    return 1
end

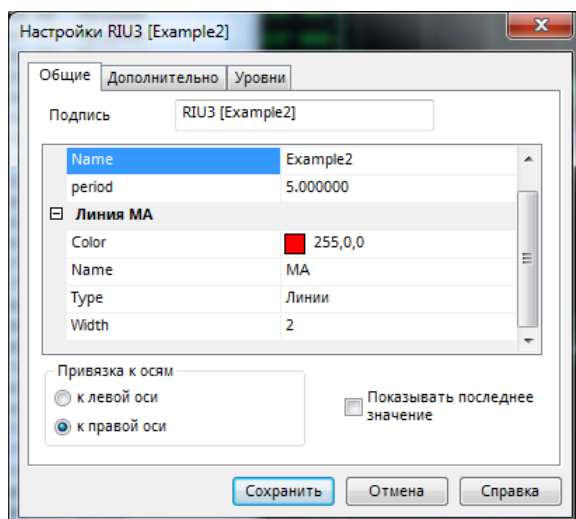
function OnCalculate(index)
    return nil
end

```

Здесь в таблицу Settings добавились поля period и line. Поле line является массивом таблицы с индексным доступом. Это значит, что все элементы таблицы доступны через численные индексы: line[1], line[2] и т.д.

Индикатор в примере возвращает только одну линию, поэтому и свойства описаны только для одной линии с индексом 1.

В диалоге настроек это выглядит следующим образом:



Всё, что не относится к описанию параметров линий, попадает в группу «Пользовательские настройки». Тип параметра определяется начальным значением. Поле Name имеет строковый тип, поле period – числовой, так как в коде оно инициализировано значением «5».

Параметры линии теперь отличаются от значений по умолчанию. Например, цвет линии определяет функция RGB(255, 0 ,0).

3.4 Рисование прямой линии

Для этого достаточно изменить только функцию OnCalculate следующим образом:

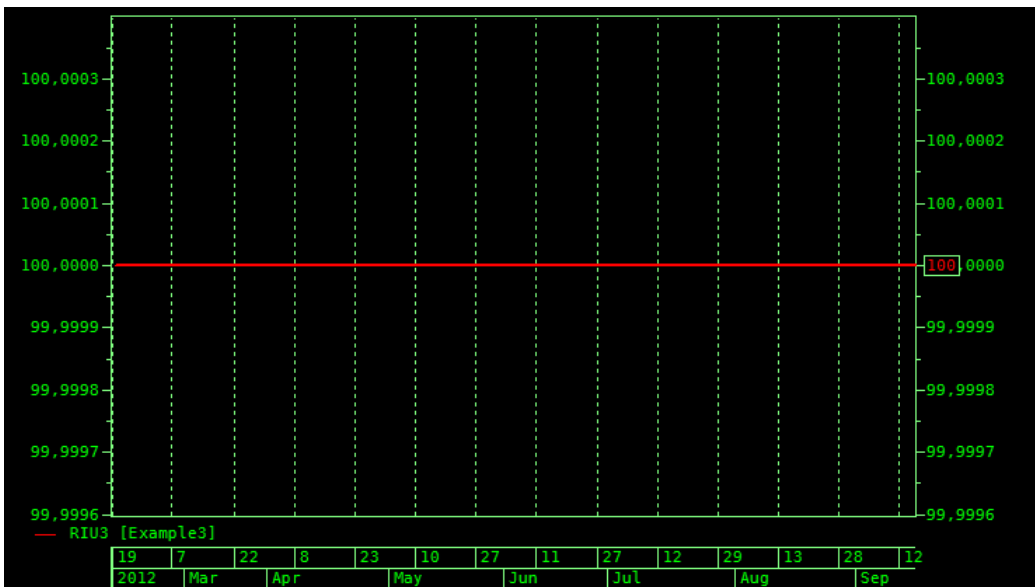
```

function OnCalculate(index)
    return 100
end

```



```
end
```



3.5 Подсчёт среднего

Подсчет среднего значения на заданном интервале по ценам закрытия свечи:

```
Settings=
{
    Name = "Example3",
    period = 5,
    line =
    {
        {
            Name = "MA",
            Color = RGB(255, 0, 0),
            Type = TYPE_LINE,
            Width = 2
        }
    }
}

function Init()
    return 1
end

function OnCalculate(index)
    if index < Settings.period then
        return nil
    else
        local sum = 0
        for i = index-Settings.period+1, index do
```



```

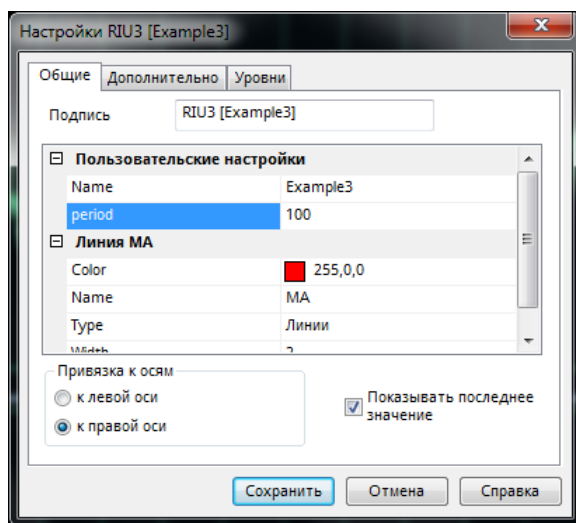
    sum = sum +C(i)
end
return sum/Settings.period
end
end

```

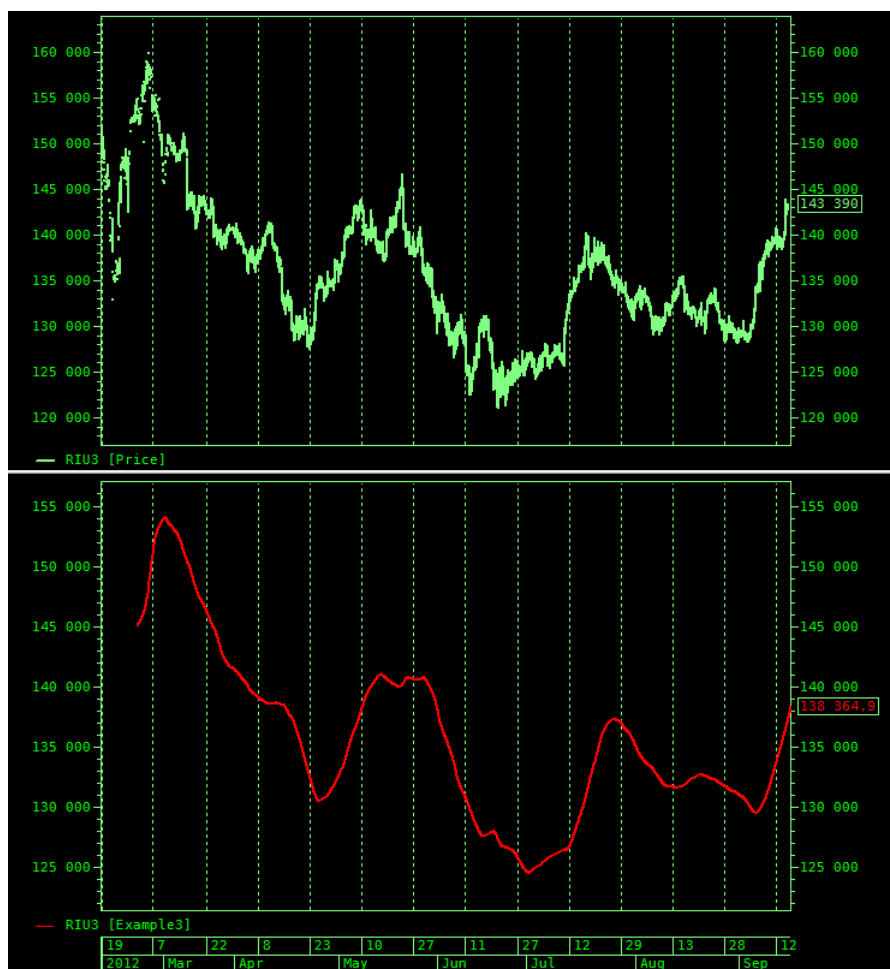
Важные моменты в коде примера:

1. Проверяется переданный индекс свечи. Если он меньше заданного периода, то возвращается nil. Данных для расчета недостаточно, поэтому значение индикатора не определено на индексах свечей, меньших, чем задано в Settings.period.
2. Для расчета индикатора везде используется поле таблицы Settings.period.

Если в свойствах индикатора поменять значение поля period и сохранить данные:



то график выглядит следующим образом:



Этот пример иллюстрирует тот факт, что все значения из диалога настроек индикатора после нажатия кнопки «Сохранить» попадают в работающую виртуальную машину Lua и становятся доступны в функциях скрипта. При этом никак не затрагивается исходный код скрипта на диске и индикаторы, уже созданные с его помощью. Если повторно добавить этот индикатор на график, то получим предыдущую картинку.

3.6 Доступ к данным

Каждый индикатор привязан к источнику данных. Для доступа к данным скрипт использует следующие функции:

- O(i);
- H(i);
- L(i);
- C(i);
- V(i);
- T(i).



За исключением функции T(), все они возвращают соответствующее значение для указанного бара – Open, High, Low, Close и Volume. Функция T() возвращает таблицу, которая содержит время указанного бара.

Среднее значение можно рассчитать не только по цене закрытия. Усложним код, добавив функцию:

```
Settings=
{
  Name = "Example3",
  period = 5,
  value_type = "C",
  line =
  {
    {
      Name = "MA",
      Color = RGB(255, 0, 0),
      Type = TYPE_LINE,
      Width = 2
    }
  }
}

function dValue(i,param)
  local v = param or "C"
  if v == "O" then
    return O(i)
  elseif v == "H" then
    return H(i)
  elseif v == "L" then
    return L(i)
  elseif v == "C" then
    return C(i)
  elseif v == "V" then
    return V(i)
  elseif v == "M" then
    return (H(i) + L(i))/2
  elseif v == "T" then
    return (H(i) + L(i)+C(i))/3
  elseif v == "W" then
    return (H(i) + L(i)+2*C(i))/4
  else
    return C(i)
  end
end

function Init()
```



```

    return 1
end

function OnCalculate(index)
    if index < Settings.period then
        return nil
    else
        local sum = 0
        for i = index-Settings.period+1, index do
            sum = sum + dValue(i, Settings.value_type)
        end
        return sum/Settings.period
    end
end
end

```

В диалоге настроек графика появляется ещё одна переменная – `value_type`. Меняя её значение, можно изменять поведение функции `dValue()` и, соответственно, входные значения для расчета среднего значения.

3.7 Расчёт EMA

EMA вычисляется по следующей итерационной формуле:

$$EMA_i = \alpha * P_i + (1 - \alpha) * EMA_{i-1}$$

Нет возможности обратиться напрямую к предыдущим рассчитанным значениям индикатора. Это значит, что для вычислений текущего значения необходимо хранить и предыдущие значения. Для таких целей в Lua используется механизм замыканий. Определение такой функции нужно вынести в отдельный файл и в отдельный каталог, который не сканируется при создании индикатора.

Пример файла с функцией расчета EMA (`ma.lua`):

```

function round(num, idp)
    if num == nil then return nil end
    local mult = 10^(idp or 0)
    return math.floor(num * mult + 0.5) / mult
end

function dValue(index, v_type)
    v_type = v_type or "C"
    if v_type == "O" then
        return O(index)
    elseif v_type == "H" then
        return H(index)
    elseif v_type == "L" then

```



```

        return L(index)
    elseif v_type=="C" then
        return C(index)
    elseif v_type=="V" then
        return V(index)
    end
    return 0
end

function average(_start, _end, v_type)
    local sum=0
    for i = _start, _end do
        sum=sum+dValue(i, v_type)
    end
    return sum/(_end-_start+1)
end

function cached_EMA()
    local cache={}
    return function(ind, _p, v_t, kk)
        local n = 0
        local p = 0
        local period = _p
        local v_type = v_t
        local index = ind
        local k = kk or 2/(period+1)
        if index == 1 then
            cache = {}
        end
        if index < period then
            cache[index] = average(1,index, v_type)
            return nil
        end
        p = cache[index-1] or dValue(index, v_type)
        n = k*dValue(index, v_type)+(1-k)*p
        cache[index] = n
        return n
    end
end

function cached_DTEMA()
    local cache_EMA={}
    local cache_DMA={}
    local cache_TMA={}
    return function(ind, _p, v_t, kk)
        local n_ema = 0
        local p_ema = 0

```



```

        local n_dma = 0
        local p_dma = 0
        local n_tma = 0
        local p_tma = 0
        local period = _p
        local v_type = v_t
        local index = ind
        local dv = dValue
        local k = kk or 2/(period+1)
        if index == 1 then
            cache_DMA = {}
            cache_EMA = {}
            cache_TMA = {}
        end
        if index < period then
            cache_EMA[index] = average(1,index, v_type)
            return nil
        end
        p_ema = cache_EMA[index-1] or dv(index, v_type)
        n_ema = k*dv(index, v_type)+(1-k)*p_ema
        cache_EMA[index] = n_ema

        p_dma = cache_DMA[index-1] or cache_EMA[index-1]
        n_dma = k*n_ema + (1-k)*p_dma
        cache_DMA[index] = n_dma

        p_tma = cache_TMA[index-1] or cache_DMA[index-
1] or cache_EMA[index-1]
        n_tma = k*n_dma + (1-k)*p_tma
        cache_TMA[index] = n_tma
        return round(n_dma, 2), round(n_tma, 2)
    end
end
end

```

В этом примере, кроме функции `cached_EMA`, присутствует функция для расчета ДЕМА и ТЕМА.

В папке с РМ QUIK создадим папку `Include`, куда и сохраним файл `ma.lua`.

Код индикатора с использованием сохраненного файла принимает следующий вид:

```

dofile(getWorkingFolder() .. "\\Include\\ma.lua")

Settings =
{
    Name = "EMA",

```




```

        period = 50,
        value_type = "C",
        line=
        {
            {
                Name = "1",
                Color = RGB(255, 0, 0),
                Type = TYPE_LINE,
                Width = 2
            }
        }
    }
}

function Init()
    myEMA = cached_EMA()
    return 1
end

function OnCalculate(index)
    return myEMA(index, Settings.period, Settings.value_type)
end

```

Функция `getWorkingFolder` возвращает путь папки с файлом `info.exe`. В примере последний параметр не используется и по умолчанию в функции инициализируется значением $2/(\text{Setting.period}+1)$.

Пример полученного индикатора:



3.8 Индикатор с несколькими линиями

Добавим в код индикатора еще одну линию EMA с собственными параметрами:

```
dofile(getWorkingFolder() .. "\\Include\\ma.lua")

Settings =
{
    Name = "Two EMA",
    period1 = 50,
    value_type1 = "C",
    period2 = 50,
    value_type2 = "C",

    line=
    {
        {
            Name = "EMA 1",
            Color = RGB(255, 0, 0),
            Type = TYPE_LINE,
            Width = 2
        },
        {
            Name = "EMA 2",
            Type = TYPE_LINE,
            Width = 2
        }
    }
}

function Init()
    myEMA1 = cached_EMA()
    myEMA2 = cached_EMA()
    return 2
end

function OnCalculate(index)
    ema1 = myEMA1(index, Settings.period1, Settings.value_type1)
    ema2 = myEMA2(index, Settings.period2, Settings.value_type2)
    return round(ema1,2), round(ema2,2)
end
```



Вид данного графика в РМ QUIK:



3.9 Функция OnDestroy

Иногда в коде индикатора бывает необходимо не только заниматься расчетами, но и использовать какие-либо системные ресурсы, например файлы. Для понимания, когда их нужно освободить, существует функция OnDestroy.

Если функция OnDestroy определена в скрипте, то она вызывается при удалении индикатора или закрытии окна с графика данного индикатора. Пример кода:

```
Settings={}
Settings.Name = "FileOp"
Settings.mode = 0

file = nil

function Log(s)
    local x = tostring(Settings.Name)
    if file ~=nil then
        file:write(x.. " : " .. s .. "\n")
        file:flush()
```



```

        end
    end
end
function Init()
    file = io.open(getScriptPath() .. "\\zigzag.log", "a+")
    Log("Init return 1")
    return 1
end
function OnCalculate(i)
    Log("OnCalculate(" .. i .. ")")
end
function OnDestroy()
    Log("OnDestroy()")
    if file~=nil then
        file:close()
    end
end
end

```

3.10 Magician birthday

Модифицируем функцию `cached_EMA`:

```

function cached_EMA_Ex(__period, __k)
    local cache={}
    local period = __period
    local k = __k or 2/(period+1)
    return function(ind, v_t)
        local n = 0
        local p = 0
        --local period = _p
        local v_type = v_t
        local index = ind
        if index == 1 then
            cache = {}
        end
        if index < period then
            cache[index] = average(1,index, v_type)
            return nil
        end
        p = cache[index-1] or dValue(index, v_type)
        n = k*dValue(index, v_type)+(1-k)*p
        cache[index] = n
        return n
    end
end
end

```



и напишем индикатор:

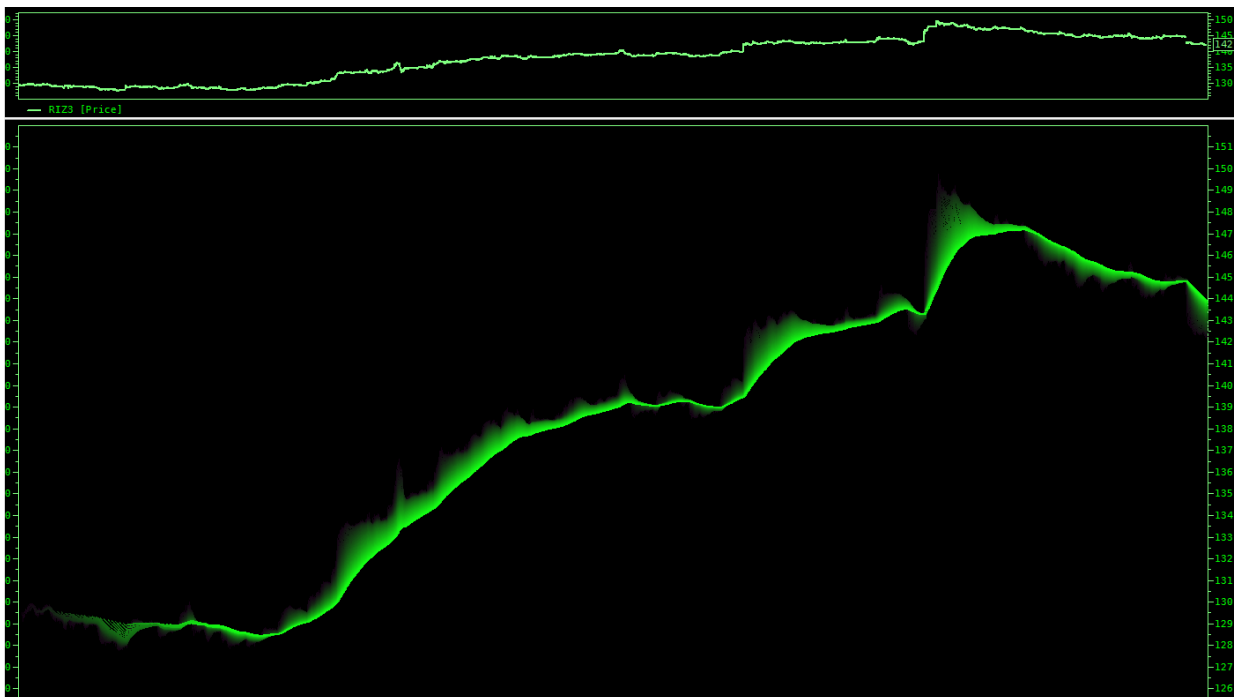
```
dofile(getWorkingFolder() .. "\\Include\\ma.lua")

Settings={}
Settings.StartN = 5
Settings.Nstep = 5
Settings.N = 100
Settings.Name = Settings.N .. " MA"

gtMA={}
function Init()
    Settings.line = {}
    for i = 1, Settings.N do
        gtMA[i] = cached_EMA_Ex(Settings.StartN + (i-1)*Settings.Nstep,
1/i)

        Settings.line[i] = {}
        Settings.line[i] = {Color = RGB(20, 255/Settings.N*i, 20), Type =
TYPE_LINE, Width = 1}
    end
    return Settings.N
end
function OnCalculate(idx)
    local res={}
    for i=1, Settings.N do
        res[i] = gtMA[i](idx,"C")
    end
    return unpack(res)
end
```





4. Отправка транзакций из Lua скрипта

Для отправки транзакций на сервер QUIK из скрипта Lua плагин QLua предоставляет функцию **sendTransaction()**, а для получения результата выполнения транзакции на сервере QUIK функцию обратного вызова **OnTransReply()**. Рассмотрим структуру данных функций и способы их использования.

Функция sendTransaction()

Функция **sendTransaction()** принимает в качестве аргумента таблицу Lua с параметрами транзакции. Если транзакция не прошла проверку на стороне РМ QUIK, то функция возвращает описание ошибки, иначе – транзакция отправляется на сервер QUIK. Рассмотрим следующий пример:

```
transaction = {TRANS_ID='400001',
               ACTION='NEW_ORDER',
               CLASSCODE='TQBR',
               ACCOUNT='L01-00XXXXXX',
               CLIENT_CODE='QX',
               SECCODE='LKOH',
               OPERATION='B',
               PRICE='2650.5',
               QUANTITY='300',
               TYPE='L'}

result = sendTransaction(transaction)

if result ~= "" then
    message(string.format("Транзакция %s не прошла проверку на стороне терминала QUIK [%s]",
                          transaction.TRANS_ID, result))
else
    message(string.format("Транзакция %s отправлена",
                          transaction.TRANS_ID))
end
```

В данном примере поля Lua таблицы **transaction** аналогичны полям .tri-файла с параметрами транзакций (подробное описание формата .tri-файла и примеры транзакций см. в Руководстве пользователя QUIK, раздел 6 «Совместная работа с другими приложениями», меню Импорт транзакций / Формат .tri-файла с параметрами транзакций). При этом для корректной обработки данных числовые значения (цена, количество и т.д.) должны передаваться в виде строковых значений.

Для использования функционала отправки транзакций средствами Lua скриптов необходимо знать, что существует два формата .tri-файла – специальный и универсальный. Особенность



специального формата в том, что для некоторого набора транзакций зарезервированы специальные наименования полей и их значения, которые описаны в Руководстве пользователя QUIK в разделе 6 «Совместная работа с другими приложениями». В предыдущем примере использован специальный формат транзакции.

Особенностью универсального формата является то, что с помощью него можно сформировать транзакцию, используя непосредственно описание поля транзакции, а не зарезервированное наименование поля. Это позволяет пользователю формировать любые доступные ему транзакции. Рассмотрим следующий пример:

```
transaction = {}
transaction['TRANS_ID'] = '400001'
transaction['ACTION'] = 'Ввод заявки'
transaction['CLASSCODE'] = 'TQBR'
transaction['Торговый счет'] = 'L01-000XXXXX'
transaction['Примечание'] = 'QX'
transaction['Инструмент'] = 'ЛКОН'
transaction['К/П'] = 'Купля'
transaction['Цена'] = '2650.5'
transaction['Лоты'] = '300'
transaction['Тип'] = 'Лимитная'

result = sendTransaction(transaction)

if result ~= "" then
    message(string.format("Транзакция %s не прошла проверку на стороне терминала QUIK [%s]",
                          transaction.TRANS_ID, result))
else
    message(string.format("Транзакция %s отправлена",
                          transaction.TRANS_ID))
end
```

В данном примере описана транзакция из предыдущего примера, но с помощью универсального формата. Так как описание транзакции в универсальном формате может содержать имена полей с пробелами и кириллицей, то здесь используется другой синтаксис работы с полями таблицы Lua. Стоит отметить, что поля TRANS_ID (пользовательский номер транзакции), ACTION (вид транзакции) и CLASSCODE (код класса, по которому выполняется транзакция) обязательны и их наименования одинаковы как в универсальном формате, так и в специальном. Еще одной особенностью универсального формата является зависимость от языковых установок РМ QUIK (меню «Система / Настройки / Языковые установки...»). Приведенный выше пример актуален, если в РМ QUIK выбран язык Russian. При использовании в РМ QUIK языка English аналог предыдущего примера выглядит следующим образом:

```
transaction = {}
```




```

transaction['TRANS_ID'] = '400001'
transaction['ACTION'] = 'Enter order'
transaction['CLASSCODE'] = 'TQBR'
transaction['Trading account'] = 'L01-00XXXXXX'
transaction['Broker reference'] = 'QX'
transaction['Security'] = 'LKOH'
transaction['Direction'] = 'Buy'
transaction['Price'] = '2650.5'
transaction['Lots'] = '300'
transaction['Type'] = 'Limit'

result = sendTransaction(transaction)

if result ~= "" then
    message(string.format("Транзакция %s не прошла проверку на стороне терминала QUIK [%s]",
                           transaction.TRANS_ID, result))
else
    message(string.format("Транзакция %s отправлена",
                           transaction.TRANS_ID))
end

```

Структура специального формата .tri-файла подробно с примерами описана в Разделе 6 Руководства пользователя QUIK, поэтому остановимся на рассмотрении формирования tri-файла в универсальном формате. Не будем рассматривать каждый вид транзакций, так как из-за большого количества рынков и особых настроек разных брокеров, все они могут иметь свои особенности и рассмотрение всех их выходит за рамки данной темы. Рассмотрим универсальный способ формирования транзакции в Lua для любой доступной пользователю транзакции.

В РМ QUIK добавим таблицу «Карман транзакций» (меню «Создать окно / Все типы окон...»), откроется окно с параметрами создаваемой таблицы, где в разрезе классов видны доступные транзакции и доступные параметры выбранной транзакции. Выбираем необходимые параметры транзакции (звездочкой отмечены параметры обязательные для заполнения пользователем, остальные параметры, если их не указать, примут значения по умолчанию).



Создание кармана транзакций

Имя таблицы: Карман

Доступные классы

Q

FORTS: РПС: Спрэды между фьючерсами

FORTS: Спрэды между фьючерсами

БК

БК 1

БК 2

БК 3

БК РПС: А1-Акции и паи

БК РПС: А1-Облигации(пусто)

ГК: РПС: РПС

Доступные транзакции

Q

Ввод заявки на спрэд фьючерсов

Снятие заявки на спрэд фьючерсов

Ввод Айсберг-заявки

Столбцы

Доступные параметры

Q

Класс

Транзакция

*Инструмент

*Торговый счет

*Тип

*К/П

*Своп-Цена

*Количество

Добавить

Добавить все

Убрать

Очистить

Заголовки столбцов

Q

*К/П

*Тип

*Инструмент

*Количество

Да

Отмена

Справка

После нажатия кнопки «Да» откроется создаваемый «Карман транзакций», в который нужно добавить необходимые транзакции, выбрав пункт контекстного меню «Положить в карман» и заполнив все необходимые параметры транзакции на форме.

Карман						
	Торговый счет	К/П	Тип	Инструмент	Цена	Количество
1	NF000XX	Покупка	Лимитированная	Si-9.16	69 000	100
2	NF000XX	Продажа	Рыночная	Si-9.16	65 272	100

После этого данные транзакции необходимо сохранить в .tri-файл, выбрав пункт контекстного меню «Сохранить транзакции в tri-файл». Получаем описание созданных транзакций, где символом «;» разделены поля транзакции:

```
TRANS_ID=1;CLASSCODE=SPBFUT;ACTION=Ввод заявки;Торговый
счет=NF000XX;К/П=Покупка;Тип=Лимитированная;Класс=SPBFUT;Инструмент=SiU6;Цена=69000;К
оличество=100;Условие исполнения=Поставить в очередь;Комментарий=;Переносить
заявку=Нет;Дата экспирации=20160414;

TRANS_ID=2;CLASSCODE=SPBFUT;ACTION=Ввод заявки;Торговый
счет=NF000XX;К/П=Продажа;Тип=Рыночная;Класс=SPBFUT;Инструмент=SiU6;Цена=65272;Количес
тво=100;Условие исполнения=Поставить в очередь;Комментарий=;Переносить
заявку=Нет;Дата экспирации=20160414;
```

При выгрузке .tri-файла из РМ QUIK выгружаются все поля транзакции, как обязательные, так и необязательные, при этом необязательные поля



принимают значения по умолчанию. При формировании транзакции из Lua скрипта необязательные поля можно не задавать, в этом случае они автоматически принимают значения по умолчанию.

Полученную структуру транзакции и возможные значения ее параметров можно использовать в формировании транзакций из Lua скрипта:

```
transaction = {}
transaction['TRANS_ID'] = '400001'
transaction['CLASSCODE'] = 'SPBFUT'
transaction['ACTION'] = 'Ввод заявки'
transaction['Торговый счет'] = 'HF000XX'
transaction['К/П'] = 'Продажа'
transaction['Тип'] = 'Лимитированная'
transaction['Инструмент'] = 'SiU6'
transaction['Цена'] = '69000'
transaction['Количество'] = '100'

result = sendTransaction(transaction)

if result ~= "" then
    message(string.format("Транзакция %s не прошла проверку на стороне терминала QUIK [%s]",
                          transaction.TRANS_ID, result))
else
    message(string.format("Транзакция %s отправлена",
                          transaction.TRANS_ID))
end
```

Функция OnTransReply()

Функция **OnTransReply()** предоставляет возможность получения информации о результате обработки транзакции на стороне сервера QUIK.

Во всех примерах в качестве TRANS_ID мы указывали одно и то же значение, это не является ошибкой, так как сервер QUIK не требует уникальности данного поля, его уникальность должен поддерживать пользователь. Данное поле предоставляет возможность однозначного сопоставления поданной пользователем транзакции и полученного с сервера QUIK ответа на транзакцию. Если пользователь не поддерживает уникальность поля TRANS_ID, он теряет возможность корректного определения, по какой транзакции пришел ответ с сервера.

Событие **OnTransReply()** срабатывает для всех транзакций с полем TRANS_ID, таким образом, результаты обработки транзакций, которые были отправлены с помощью функционала Trans2quik.dll, QPILE или динамической загрузки транзакций из файла так же



могут быть перехвачены в скрипте Lua. Для транзакций, отправленных вручную через графический интерфейс РМ QUIK, функция **OnTransReply()** не вызывается.

Пример реализации отправки транзакций из Lua скрипта с получением информации о результатах их выполнения:

```
--отправка транзакции на продажу по рыночной цене каждую секунду
--с информированием о результате выполнения данной транзакции

--определяем структуру для транзакции 'Ввод заявки' по классу TQBR
NewOrder_TQBR = {}
NewOrder_TQBR['TRANS_ID'] = ''
NewOrder_TQBR['ACTION'] = 'Ввод заявки'
NewOrder_TQBR['CLASSCODE'] = 'TQBR'
NewOrder_TQBR['Торговый счет'] = ''
NewOrder_TQBR['Примечание'] = ''
NewOrder_TQBR['Инструмент'] = ''
NewOrder_TQBR['К/П'] = ''
NewOrder_TQBR['Цена'] = ''
NewOrder_TQBR['Лоты'] = ''
NewOrder_TQBR['Тип'] = ''

function OnInit(script_path)
    is_run = true
    --для поддержания уникальности TRANS_ID задаем первый номер транзакции текущим временем
    системы
    TRANS_ID = os.time()
end

function OnTransReply(trans_reply)
    --информирует о каждом получении результата обработки транзакций
    message(string.format("Получен ответ на транзакцию %i. Статус - %i [%s]",
        trans_reply.trans_id,
        trans_reply.status,
        trans_reply.result_msg))
end

function OnStop()
    is_run = false
    return 2000
end

function main()
    while is_run do
        if isConnected() == 1 then
            --задаем параметры транзакции
            NewOrder_TQBR['TRANS_ID'] = tostring(TRANS_ID)
```



```

NewOrder_TQBR['Торговый счет'] = 'L01-00XXXXXX'
NewOrder_TQBR['Примечание'] = 'QX'
NewOrder_TQBR['Инструмент'] = 'LКОН'
NewOrder_TQBR['К/П'] = 'Продажа'
NewOrder_TQBR['Цена'] = '0'
NewOrder_TQBR['Лоты'] = '1'
NewOrder_TQBR['Тип'] = 'Рыночная'

local result = sendTransaction(NewOrder_TQBR)

if result ~= "" then
    message(string.format("Транзакция %s не прошла проверку на стороне терминала QUIK [%s]", NewOrder_TQBR.TRANS_ID, result))
else
    message(string.format("Транзакция %s отправлена",
                          NewOrder_TQBR.TRANS_ID))
end

--увеличиваем TRANS_ID
TRANS_ID = TRANS_ID + 1
end
sleep(1000)
end
end

```

В данном примере использовалась так называемая асинхронная отправка транзакций, когда после отправки транзакции скрипт не ждет результата ее обработки, а продолжает выполнение следующих сценариев. При этом в основном потоке РМ QUIK при приходе результата обработки транзакции и вызове функции **OnTransReply()** скрипт отправляет сообщение со статусом транзакции и текстовым ответом торговой системы или сервера QUIK. Полное описание всех полей результата транзакции при вызове функции **OnTransReply()** и возможные варианты статусов приведены в разделе 4.24 «Транзакции» Руководства пользователя Интерпретатора языка Lua. Все результаты обработки транзакций, которые отправлялись из скрипта Lua, доступны в «Таблице транзакций» РМ QUIK (меню «Создать окно / Все типы окон...»).

Рассмотрим пример реализации алгоритма синхронной отправки транзакций, когда скрипт после каждой отправки транзакции ожидает результат ее выполнения в течение некоторого таймута, и после этого продолжает свою работу, либо после разрыва связи РМ QUIK с сервером:

```

--реализация алгоритма синхронной отправки транзакций

NewOrder_TQBR = {}
NewOrder_TQBR['TRANS_ID'] = ''

```



```

NewOrder_TQBR['ACTION'] = 'Ввод заявки'
NewOrder_TQBR['CLASSCODE'] = 'TQBR'
NewOrder_TQBR['Торговый счет'] = ''
NewOrder_TQBR['Примечание'] = ''
NewOrder_TQBR['Инструмент'] = ''
NewOrder_TQBR['К/П'] = ''
NewOrder_TQBR['Цена'] = ''
NewOrder_TQBR['Лоты'] = ''
NewOrder_TQBR['Тип'] = ''

function OnInit(script_path)
    is_run = true
    TRANS_ID = os.time()
    --объявляем глобальную Lua таблицу,
    --которая будет хранить информацию об обработке каждой транзакции
    TRANSACTION_COMPLETED = {}
end

function OnTransReply(trans_reply)
    if trans_reply.trans_id == TRANS_ID then
        message(string.format("Получен ответ на транзакцию %i. Статус - %i [%s]",
                               trans_reply.trans_id,
                               trans_reply.status,
                               trans_reply.result_msg))
    end

    if trans_reply.status >= 2 then
        --если статус транзакции 2 или больше считаем транзакцию обработанной
        --и сохраняем результат ее обработки
        TRANSACTION_COMPLETED[trans_reply.trans_id] = trans_reply
    end
end

function OnStop()
    is_run = false
    return 2000
end

function main()
    while is_run do
        if isConnected() == 1 then
            NewOrder_TQBR['TRANS_ID'] = tostring(TRANS_ID)
            NewOrder_TQBR['Торговый счет'] = 'L01-00XXXXXX'
            NewOrder_TQBR['Примечание'] = 'QX'
            NewOrder_TQBR['Инструмент'] = 'LКОН'
            NewOrder_TQBR['К/П'] = 'Продажа'
            NewOrder_TQBR['Цена'] = '0'

```



```

NewOrder_TQBR['Лоты'] = '1'
NewOrder_TQBR['Тип'] = 'Рыночная'

local is_completed = sendTransactionSync(NewOrder_TQBR)

TRANS_ID = TRANS_ID + 1
end
sleep(500)
end
end

function sendTransactionSync(transaction)
    local result = sendTransaction(transaction)

    if result ~= "" then
        message(string.format("Транзакция %s не прошла проверку на стороне терминала QUIK [%s]", NewOrder_TQBR.TRANS_ID, result))
    else
        message(string.format("Транзакция %s отправлена",
                                NewOrder_TQBR.TRANS_ID))

        --время (сек) ожидания ответа на транзакцию
        local timeout = 30
        --запоминаем время старта ожидания ответа на транзакцию
        local timeStart = os.time()

        --пока текущее время меньше времени старта + время ожидания
        --и установлено соединение с сервером QUIK ждем результата обработки транзакции
        while os.time() < timeStart + timeout and isConnected() == 1 do
            --проверяем, есть ли запись в таблице TRANSACTION_COMPLETED с номером нашей транзакции
            if TRANSACTION_COMPLETED[tonumber(NewOrder_TQBR.TRANS_ID)] then
                return true
            end

            --если время ожидания вышло, и ответ со статусом 2 или больше не получен,
            --то сообщаем об этом и возвращаем false
            if os.time() >= timeStart + timeout then
                message(string.format("Результат обработки транзакции %s не получен с сервера QUIK за таймаут", NewOrder_TQBR.TRANS_ID))
                return false
            end
        end
    end
end
end

```



В данном примере внутри функции обратного вызова **OnTransReply()**, кроме информирования о статусе обработки транзакции, мы в глобальную Lua таблицу **TRANSACTION_COMPLETED** добавляем записи с номерами транзакций и информацией о результате их обработки, а внутри нашей функции **sendTransactionSync()** после отправки транзакции ждем, когда в этой глобальной таблице появится запись с номером отправленной транзакции. В данном примере транзакция будет считаться обработанной, если ее статус больше либо равен «2», так как статусы «0 – транзакция отправлена серверу» и «1 – транзакция получена на сервер QUIK от клиента» не являются окончательными статусами обработки транзакции.



5. Использование функции **PrintDbgStr()** на практике

Для вывода отладочной информации работы скрипта Lua плагин QLua предоставляет функцию **PrintDbgStr()**. В основе функции **PrintDbgStr()** лежит вызов WinAPI функции **OutputDebugString()**, описание которой можно посмотреть на сайте [Microsoft Software Developer Network](https://docs.microsoft.com/ru-ru/windows/win32/api/debugapi/nf-debugapi-outputdebugstring) (MSDN). При вызове функции **OutputDebugString()** выполняется поиск зарегистрированной в системе «программы-ловушки» отладочных сообщений. Если такая программа найдена, ей отправляется сообщение, и дальнейшая обработка данного сообщения зависит от функционала «программы-ловушки». Если «программа-ловушка» не найдена, то вызов **OutputDebugString()** ничего не делает. Преимущество данной функции в том, что если «программа-ловушка» в системе не найдена, то время выполнения данной функции минимально и практически не задерживает работу программы. Это позволяет оставлять вывод отладочной информации даже в готовой программе.

Рассмотрим примеры использования функции **PrintDbgStr()** в Lua скриптах. В следующем примере Lua скрипт каждые 5 секунд отправляет «программе-ловушке» отладочное сообщение в виде текущей даты и времени:

```
is_run = true

function OnStop()
    is_run = false
end

function main()
    while is_run do
        PrintDbgStr("QLua: " .. os.date())
        sleep(5000)
    end
end
```

Данные сообщения нигде не сохраняются и для того, чтобы их увидеть, необходимо иметь доступ к «программе-ловушке». Многие популярные отладчики имеют функционал перехвата таких отладочных сообщений. Для демонстрации мы используем бесплатную утилиту от Microsoft – [DebugView](https://docs.microsoft.com/ru-ru/windows/debug/debugview).

Запустив утилиту DebugView и приведенный выше код скрипта Lua в РМ QUIK, мы увидим отладочные сообщения, отправляемые функцией **PrintDbgStr()**:



#	Time	Debug Print
8	0.87521386	[472] QLua: 04/26/16 10:51:20
10	5.88360071	[472] QLua: 04/26/16 10:51:25
11	10.89935589	[472] QLua: 04/26/16 10:51:30
12	15.91522121	[472] QLua: 04/26/16 10:51:35
13	20.93064499	[472] QLua: 04/26/16 10:51:40
14	25.94589043	[472] QLua: 04/26/16 10:51:45
15	30.96186066	[472] QLua: 04/26/16 10:51:50
16	35.97740936	[472] QLua: 04/26/16 10:51:55
17	40.99300385	[472] QLua: 04/26/16 10:52:00
18	46.00864410	[472] QLua: 04/26/16 10:52:05
19	51.16510010	[472] QLua: 04/26/16 10:52:10
20	56.03960037	[472] QLua: 04/26/16 10:52:15
118	1792.15136719	[472] QLua: 04/26/16 11:21:11
120	1797.16296387	[472] QLua: 04/26/16 11:21:16
121	1802.17846680	[472] QLua: 04/26/16 11:21:21
122	1807.19409180	[472] QLua: 04/26/16 11:21:26
123	1812.20971680	[472] QLua: 04/26/16 11:21:31
124	1817.22497559	[472] QLua: 04/26/16 11:21:36
126	1822.24035645	[472] QLua: 04/26/16 11:21:41

Функционал данной программы довольно широк и подробно описан в прилагаемой справке к программе. Отметим следующее: для того, чтобы отображались только необходимые нам сообщения, необходимо добавить фильтр, который будет отображать только те сообщения, которые содержат или не содержат указанную подстроку:

DebugView Filter

Enter multiple filter match strings separated by the ';' character.
 "" is a wildcard.

Include: QLua

Exclude:

Highlight: Filter 1

Colors

OK, Reset, Cancel, Load, Save

Если используется несколько РМ QUIK, и в каждом из них запущен скрипт Lua, в котором используется функция **PrintDbgStr()**, то понять, из какого РМ получено отладочное сообщение, можно по идентификатору процесса, который указывается в квадратных скобках перед каждым сообщением. Узнать идентификатор процесса для запущенного РМ QUIK можно в Диспетчере задач Windows.

Рассмотрим пример решения задачи контроля активности Lua скрипта и информирования пользователя по почте, если скрипт перестал подавать признаки активности. Для решения данной задачи мы будем использовать функционал WinAPI с реализацией собственной «программы-ловушки» на интерпретируемом языке программирования Python 2.7. Ниже приведен листинг скрипта QLuaController.py:

```
# -*- coding: windows-1251 -*-

import mmap
import struct
import win32event
import time
import smtplib
import sys
```



```

def QLuaControllerStart(Filter = "QLua", TimeOut = 60):
    #Создаем объект события готовности буфера для получения отладочных сообщений
    buffer_ready = win32event.CreateEvent(None, 0, 0, "DBWIN_BUFFER_READY")
    #Создаем объект события получения данных отладочного сообщения
    data_ready = win32event.CreateEvent(None, 0, 0, "DBWIN_DATA_READY")
    #Выделяем память под буфер обмена
    _buffer = mmap.mmap(0, 4096, "DBWIN_BUFFER", mmap.ACCESS_WRITE)
    timeStart = time.time()

    while time.time() < timeStart + int(TimeOut):
        #Устанавливаем событие готовности буфера для получения отладочных сообщений,
        #т.е. регистрируем "программу-ловушку"
        win32event.SetEvent(buffer_ready)
        #Если есть сигнал получения данных отладочного сообщения - обрабатываем его
        if win32event.WaitForSingleObject(data_ready, 1) == win32event.WAIT_OBJECT_0:
            _buffer.seek(0)
            #Получаем идентификатор процесса, отправившего отладочное сообщение
            process_id, = struct.unpack("L", _buffer.read(4))
            #Считываем отладочное сообщение из буфера обмена
            data = _buffer.read(4092)
            #Считываем строку до символа окончания строки, если он присутствует
            if "\0" in data:
                str1 = data[:data.index("\0")]
            #иначе вся строка в буфере является отладочным сообщением
            else:
                str1 = data

            #Проверяем присутствие подстроки Filter в сообщении,
            #если она присутствует, считаем, что
            #отладочное сообщение получено от нашего скрипта Lua
            if str1.find(Filter) >= 0:
                ticks = time.localtime()
                #Выводим отладочное сообщение в консоль
                print "Pid %d [%02d:%02d:%02d]: %s" % (process_id,
                                                         ticks.tm_hour,
                                                         ticks.tm_min,
                                                         ticks.tm_sec,
                                                         str1)

                timeStart = time.time()

            #Если в течение установленного таймаута с момента
            #последнего отладочного сообщения не было получено
            #новых отладочных сообщений скрипта Lua - отправляем письмо
            if time.time() >= timeStart + int(TimeOut):
                SendMail("Script %s non active" % Filter)

```



```

def SendMail(MsgText):
    #От кого
    fromaddr = 'Mr. Robot <someaccount@test.com>'
    #Кому
    toaddr = 'Administrator <administrator@test.com>'
    #Тема письма
    subj = 'Notification from QLuaController'
    #Текст сообщения
    msg_txt = 'Notice:\n\n ' + MsgText + '\n\nBye!'
    #Создаем письмо (заголовки и текст)
    msg = "From: %s\nTo: %s\nSubject: %s\n\n%s" % ( fromaddr,
                                                    toaddr,
                                                    subj,
                                                    msg_txt)

    #Логин почтового ящика
    username = 'someaccount'
    #Пароль почтового ящика
    password = 'somepassword'

    #Инициализируем соединение с сервером по протоколу smtp
    server = smtplib.SMTP("smtp.test.com:25")
    #Переводим соединение в защищенный режим (Transport Layer Security)
    server.starttls()
    #Авторизуемся на сервере
    server.login(username, password)
    #Отправляем письмо
    server.sendmail(fromaddr, toaddr, msg)
    print "Message send to %s" % toaddr
    #Закрываем соединение с сервером
    server.quit()

if len(sys.argv) == 1:
    QLuaControllerStart()
elif len(sys.argv) == 2:
    QLuaControllerStart(sys.argv[1])
elif len(sys.argv) == 3:
    QLuaControllerStart(sys.argv[1], int(sys.argv[2]))

```

Вышеописанный пример реализации «программы-ловушки» отладочных сообщений выполняет вывод полученных отладочных сообщений в консоль и отправляет на predetermined адрес сообщение, если от скрипта Lua в течение заданного таймута не было новых сообщений.

Примеры запуска файла из командной строки (для работы скрипта необходим установленный интерпретатор Python 2.7 и выше):



1. Запуск со значениями фильтра и таймаута по умолчанию (фильтр – «QLua», таймаут – 60 секунд) для контроля Lua скрипта:

```
is_run = true

function OnStop()
    is_run = false
end

function main()
    while is_run do
        PrintDbgStr("QLua: " .. os.date())
        sleep(5000)
    end
end
```

```
D:\QUIK>QLuaController.py
Pid 5168 [12:45:10]: QLua: 04/29/16 12:45:10
Pid 5168 [12:45:15]: QLua: 04/29/16 12:45:15
Pid 5168 [12:45:20]: QLua: 04/29/16 12:45:20
Pid 5168 [12:45:25]: QLua: 04/29/16 12:45:25
Pid 5168 [12:45:30]: QLua: 04/29/16 12:45:30
Pid 5168 [12:45:35]: QLua: 04/29/16 12:45:35
Message send to Administrator <administrator@test.com>
```

2. Запуск со значением фильтра «MyQLuaScript» и таймаутом по умолчанию для контроля Lua скрипта:

```
is_run = true

function OnStop()
    is_run = false
end

function main()
    while is_run do
        PrintDbgStr("MyQLuaScript: " .. os.date())
        sleep(5000)
    end
end
```

```
D:\QUIK>QLuaController.py MyQLuaScript
```



```
Pid 5168 [12:55:10]: MyQLuaScript: 04/29/16 12:55:10
Pid 5168 [12:55:15]: MyQLuaScript: 04/29/16 12:55:15
Pid 5168 [12:55:20]: MyQLuaScript: 04/29/16 12:55:20
Pid 5168 [12:55:25]: MyQLuaScript: 04/29/16 12:55:25
Pid 5168 [12:55:30]: MyQLuaScript: 04/29/16 12:55:30
Pid 5168 [12:55:35]: MyQLuaScript: 04/29/16 12:55:35
Message send to Administrator <administrator@test.com>
```

3. Запуск со значением фильтра «MyQLuaScript» и таймаутом 10 секунд для контроля Lua скрипта:

```
is_run = true

function OnStop()
    is_run = false
end

function main()
    while is_run do
        PrintDbgStr("MyQLuaScript: " .. os.date())
        sleep(1000)
    end
end
```

```
D:\QUIK>QLuaController.py MyQLuaScript 10
Pid 5168 [13:05:10]: MyQLuaScript: 04/29/16 13:05:10
Pid 5168 [13:05:11]: MyQLuaScript: 04/29/16 13:05:11
Pid 5168 [13:05:12]: MyQLuaScript: 04/29/16 13:05:12
Pid 5168 [13:05:13]: MyQLuaScript: 04/29/16 13:05:13
Pid 5168 [13:05:14]: MyQLuaScript: 04/29/16 13:05:14
Pid 5168 [13:05:15]: MyQLuaScript: 04/29/16 13:05:15
Message send to Administrator <administrator@test.com>
```

