Так что, теперь, если вы принимаетесь за написание программы, у вас уже не должно возникать вопроса: «С чего начать?», ибо на прошлом уроке мы этот вопрос прекрасно разобрали. Но может возникнуть следующий вопрос: «А как продолжить?». Вот научились мы работать со стаканом, написали запись стакана в файл (чисто ради тренировки), а дальше-то что? Как реального робота создать?

Вообще, чтобы подобные вопросы не возникали («Как начать?», «Как продолжить?», «Как закончить?») полезно иметь определенный план действий. Вот сейчас мы с вами и составим такой план. Для начала разобьем процесс написания робота по шагам (начиная с текущего состояния):

1. Разработать механизм определения границ лучших цен, с учетом уже выставленных заявок. Для этой цели нам придется написать механизм поиска своих заявок.
2. Разработать механизм выставления заявок, с учетом того факта, что заявки могут быть уже выставлены и могут быть исполнены.
3. Разработать механизм перевыставления заявок при изменении цен.
4. Разработать механизм удаления выставленных заявок и закрытия всех открытых позиций по рынку в заданное время.

Теперь приступим к реализации этого плана. Согласно первому пункту, нам нужна какая-то подпрограмма, которая бы искала выставленные заявки. Давайте для начала определимся, а так ли уж нам нужен поиск заявок. Дело в том, что в qlua есть предопределенная функция OnOrder, которая вызывается каждый раз, когда вводиться новая заявка или когда меняются параметры существующей заявки.  Давайте испытаем эту функцию. Добавим в нашего робота вот такой код:

|  |
| --- |
| **function** OnOrder**(**order**)**        message**(tostring(**order**[**"order\_num"**]),**1**)**  **end** |

Код внутри функции OnQuote можете временно убрать, чтобы не создавать на диске кучу файлов. Кстати, сразу расскажу, как код можно убрать временно. Один из способов – поставить перед каждой строкой кода значок “--”, который обозначает комментарий:

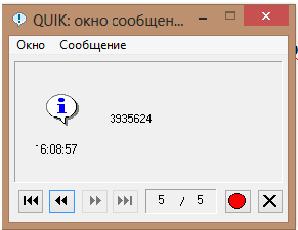
|  |
| --- |
| **function** OnQuote**(**class\_code**,** sec\_code**)**  **if** class\_code**==**p\_classcode **and** sec\_code**==**p\_seccode **then**              --l\_file=io.open("D:\\1\\1\\"..tostring(count)..".txt", "w")              --tb=getQuoteLevel2(class\_code, sec\_code)                --l\_file:write("BID:\n")              --for i=1,tb.bid\_count,1 do              --    l\_file:write(tostring(tb.bid[i].price)..";  "..              --          tostring(tb.bid[i].quantity).."\n")              --end                --l\_file:write("OFFER:\n")              --for i=1,tb.offer\_count,1 do              --    l\_file:write(tostring(tb.offer[i].price)..";  "..              --          tostring(tb.offer[i].quantity).."\n")              --end                --count=count+1              --l\_file:close()  **end**  **end** |

А можно просто поставить return 0:

|  |
| --- |
| **function** OnQuote**(**class\_code**,** sec\_code**)**  **if** class\_code**==**p\_classcode **and** sec\_code**==**p\_seccode **then**  **return** 0  … |

Тогда программа дойдет до этой команды и вернется из функции, не выполняя код, лежащий после return 0.

Итак, опробуем программу. Если сейчас мы запустим этот скрипт, а потом введем заявку, то у нас будет выдано сообщение, содержащее номер заявки:



Если мы эту заявку удалим, то получим сообщение еще раз. Все верно, OnOrder вызывается каждый раз, когда изменяется статус заявки. Удаление заявки меняет ее статус.

Но как узнать, что же мы делаем с заявкой? Вводим новую, удаляем или, может быть, событие OnOrder вызвано исполнением заявки?

На этот вопрос нам даст ответ свойство flags таблицы параметра, передаваемого в OnOrder. Измените функцию OnOrder следующим образом:

|  |
| --- |
| **function** OnOrder**(**order**)**        message**(tostring(**order**[**"order\_num"**]..**" flags="**..**order**[**"flags"**]),**1**)**  **end** |

Теперь вы будете видеть, что когда мы вводим заявку, у нас flags=25, а когда удаляем 26. Если у нас заявка исполняется, то flags=24.

Но на самом деле не нужно сравнивать flags с числом. Дело в том, что это битовые флаги. В компьютере все числа представлены в двоичном виде. В отличие от десятичной системы счисления, в двоичной предусмотрено только две цифры 0 или 1. Соответственно, каждый разряд отличается от соседнего не в 10 раз, а в 2 раза. Таким образом, число 10 в двоичной системе это 2 в десятичной, а 100 это 4. 101 в двоичной это будет 5 в десятичной. Что бы перевести число из десятичной системы в двоичную, надо делить его на 2 и записывать остаток от деления на каждой итерации. Начинаем с младшего разряда. Например, переведем число 25 в двоичную систему. Делим на 2, получаем 12, остаток 1. Делим еще на 2, получаем 6, остаток 0. Затем 3 и остаток 0. Потом 1 и остаток 1 и плюс еще остаток 1. Получим 11001. Каждый бит этого числа имеет определенное свое значение. Естественно, если 1 значит данный признак включен, 0 – выключен. Рассмотрим подробнее эти флаги:

* бит 0 (0x1) Заявка активна, иначе – не активна
* бит 1 (0x2) Заявка снята. Если флаг не установлен и значение бита «0» равно «0», то заявка исполнена
* бит 2 (0x4) Заявка на продажу, иначе – на покупку. Данный флаг для сделок и сделок для исполнения определяет направление сделки (BUY/SELL)
* бит 3 (0x8) Заявка лимитированная, иначе – рыночная
* бит 4 (0x10) Возможно исполнение заявки несколькими сделками
* бит 5 (0x20) Исполнить заявку немедленно или снять (FILL OR KILL)
* бит 6 (0x40) Заявка маркет-мейкера. Для адресных заявок – заявка отправлена контрагенту
* бит 7 (0x80) Для адресных заявок – заявка получена от контрагента
* бит 8 (0x100) Снять остаток
* бит 9 (0x200) Айсберг-заявка

Число 11001 означает, что включены следующие флаги:

* Заявка активна.
* Заявка лимитированная.
* Возможно исполнение заявки несколькими сделками.

При снятии заявки у нас flags=26, что соответствует двоичному числу 11010. В этом случае у нас флаг «Заявка активна» равен нулю, а вот флаг «заявка снята» на этот раз равен единице.

Но, согласитесь, так работать с битовыми флагами неудобно. Поэтому в qlua предусмотрены специальные средства для работы с битовыми флагами. А именно, логические функции. С помощью таких функций, например, можно проверить, установлен или брошен отдельный конкретный флаг. Например, проверить флаг «заявка снята» можно следующим образом:

|  |
| --- |
| **if** bit.band**(**order**[**"flags"**],**2**)>**0 **then**              message**(**"Заявка удалена"**,**1**)**  **end** |

Этот код может показаться странным и непонятным, поэтому, придется изучить эти самые логические функции. Всего их четыре:

* Логическое сложение – функция ИЛИ (OR).
* Логическое умножение – функция И (AND).
* Логическое отрицание – функция НЕ (NOT)
* Исключающее ИЛИ (XOR).

Разберем эти функции подробнее. Функция «ИЛИ» дает 1, если хотя бы один из ее входных битов равен 1, и дает 0, если все нули. Функция «И» дает единицу, если на всех ее входах единица, если хотя бы на одном 0 — то дает 0. Функция «НЕ» превращает 0 в 1 и 1 в 0. Исключающее или дает 0 если все входы одинаковы – только одни единицы или только одни нули. Если есть различия, то на выходе единица.

Таким образом, если мы применим операцию «И» к двоичным числам 1010 и 1100 то на выходе получим  1000, так как у нас только в старшем разряде на обоих входах единицы.

Теперь поговорим о том, как же проверить конкретный бит при помощи этих логических функций. Очень просто. Надо совершить операцию «И» между набором битовых флагов и числом, которое содержит единицу только в проверяемом разряде. В нашем примере bit.band – это как раз функция «И», первый ее аргумент – битовые флаги, второй число 2, которое соответствует в двоичном коде числу 10, содержащему единицу только во втором разряде, который соответствует биту флага «Заявка снята». Понятно, что если этот бит включен – то результат функции будет какое то число, больше чем 0, в противном случае результат будет равен нулю, ибо тогда обнулятся все разряды.

Теперь вроде бы мы пришли к выводу, что поиск заявок нам и не нужен. Но как быть в том случае, если мы робота остановили, потом снова запустили? Например, у нас завис компьютер, мы были вынуждены его перезагрузить и заново запустить квик. Даже если получив номера заявок через OnOrder мы сохраним их в памяти, эти данные при закрытии и повторном запуске робота потеряются.  Как быть? Можно сохранять их в текстовом файле. Даже если файлик потеряется при аварийном завершении работы робота, мы всегда можем набрать его «ручками». Есть и другое решение. Например, можно прочитать таблицу заявок. Но тут возникают еще и другие вопросы. А что, если мы еще и вручную торгуем? Или у нас торгует несколько роботов, и каждый создает заявки. Как найти свои заявки?

Сейчас мы не будет разбирать такие тонкости, дабы не нагружать ваши головы, на которые уже  и так свалилось много информации. Пока мы пишем учебный пример, и я постараюсь сделать его как можно проще. Но в будущем, когда будете разрабатывать своих роботов, обязательно задавайтесь этими вопросами, перед тем как начать программировать.

Итак, продолжаем. Сначала сделаем как проще, будем запоминать номера введенных заявок через OnOrder. Сохранением этих данных в файл пока заморачиваться не будем.

В начале программы объявим переменные, в которых у нас будут запоминаться сведения о введенных заявках. Разумеется, не в самом начале, не там где у нас параметры (переменные на букву p), а ниже, под ними:

|  |
| --- |
| buy\_order**=**""**;**  buy\_price**=**0**;**  buy\_count**=**0**;**  sell\_order**=**""  sell\_price**=**0**;**  sell\_count**=**0**;** |

Тут мы будем запоминать номер заявки, цену, и остаток заявки (баланс). Под остатком понимается количество инструмента, по которому еще не совершена сделка. То есть, если вы выставляете лимитированную заявку, которая исполняется не сразу, то начальное значение остатка – это количество из заявки. По мере исполнения (когда заявка исполняется частично), остаток будет уменьшатся, когда исполниться полностью станет 0. Для того чтобы получить такой остаток, нужно обратиться к атрибуту заявки balance. Не путать с атрибутом qty!

Вот каким образом мы будем получать эти данные:

|  |
| --- |
| **function** OnOrder**(**order**)**        p\_file**:write(os.date()..**" OnOrder\n"**);**        --сначала проверим, по нашему ли инструменту эта заявка  **if** order**[**"sec\_code"**]==**p\_seccode **and** order**[**"class\_code"**]==**p\_classcode **then**              p\_file**:write(os.date()..**"  заявка "**..**order**[**"order\_num"**]..**"\n"**)**                --если заявка активна, то запоминаем ее  **if** bit.band**(**order**[**"flags"**],**1**)>**0 **then**  **if** bit.band**(**order**[**"flags"**],**4**)>**0 **then**                          sell\_order**=**order**[**"order\_num"**]**                          sell\_price**=tonumber(**order**[**"price"**])**                          sell\_count**=tonumber(**order**[**"balance"**])**  **else**                          buy\_order**=**order**[**"order\_num"**]**                          buy\_price**=tonumber(**order**[**"price"**])**                          buy\_count**=tonumber(**order**[**"balance"**])**  **end**  **else**                    --если заявка не активна то сбрасываем информацию о заявке  **if** bit.band**(**order**[**"flags"**],**1**)>**0 **then**  **if** bit.band**(**order**[**"flags"**],**8**)>**0 **then**                               sell\_order**=**""                               sell\_price**=**0                               sell\_count**=**0  **else**                               buy\_order**=**""                               buy\_price**=**0                               buy\_count**=**0  **end**  **end**  **end**  **end**  **end** |

В этой процедуре мы, используя средства работы с битовыми флагами, проверяем, активна ли заявка. Если активна, то запоминаем ее данные, проверяя, что это за заявка, на покупку или на продажу. Если заявка не активна, то мы эти данные сбрасываем. Еще мы информацию о заявке пишем в лог, что бы потом мы могли проверить, правильно ли работает наша подпрограмма.

У вас может возникнуть вопрос: а для чего в программе используется tonumber? Дело в том, что в переменной order, куда функция OnOrder  передает  структуру с параметрами заявки, поля имеют строковый тип. А строка – это не число. Но нам нужно число, потому что далее мы с этими данными будем проводить различные числовые операции, в частности, сравнение. Почему строка? Так решили разработчики, нам остается с этим смириться и просто преобразовывать к нужному нам типу.

Идем дальше. Получив данные о заявке, нам нужно учесть их при расчете крайних цен в стакане. Этой подпрограммы у нас пока нет, мы реализуем ее в функции OnQuote вместо того кода, который выводил у нас стакан, генерируя кучу файлов:

|  |
| --- |
| **function** OnQuote**(**class\_code**,** sec\_code**)**  **if** class\_code**==**p\_classcode **and** sec\_code**==**p\_seccode **then**              tb**=**getQuoteLevel2**(**class\_code**,** sec\_code**)**    **if** buy\_count**==**0 **then**                    v\_bid**=**tb.bid**[math.ceil(**tb.bid\_count**)].**price**+**p\_delta  **else**  **if** **tonumber(**tb.bid**[math.ceil(**tb.bid\_count**)].**quantity**)==**buy\_count **andtonumber(**tb.bid**[math.ceil(**tb.bid\_count**)].**price**)==**buy\_price **then**                          v\_bid**=**tb.bid**[math.ceil(**tb.bid\_count**)-**1**].**price**+**p\_delta  **else**                          v\_bid**=**tb.bid**[math.ceil(**tb.bid\_count**)].**price**+**p\_delta  **end**  **end**    **if** sell\_count**==**0 **then**                    v\_offer**=**tb.offer**[**1**].**price**-**p\_delta  **else**  **if** **tonumber(**tb.offer**[**1**].**quantity**)==**sell\_count **and** **tonumber(**tb.offer**[**1**].**price**)==**sell\_price **then**                          v\_offer**=**tb.offer**[**2**].**price**-**p\_delta  **else**                          v\_offer**=**tb.offer**[**1**].**price**-**p\_delta  **end**  **end**                p\_file**:write(os.date()..**"    "**..**v\_bid**..**","**..**v\_offer**..**"\n"**)**  **end**  **end** |

Как видим, здесь точно так же при помощи getQuoteLevel2 мы получаем стакан, читаем крайние цены, сразу же изменяем их на величину p\_delta – отступ, на который мы будем размещать наши заявки. Если у нас уже выставлена заявка, то мы берем не крайние цены, а следующие за ними. Обратите внимание, что здесь еще идет сравнение цены и количества. И вот почему. Дело в том, что может быть такая ситуация, когда цены в стакане вдруг изменили, и наша заявка стала не крайней, а где то в середине. В этом случае нужно снова рассчитывать цену с краю. Или еще одна ситуация – кто-то выставил заявку то той же цене, как и у нас. Тогда нам надо передвинуть наши заявки. В этом случае тоже считаем цену с краю.

Так же у вас может возникнуть вопрос по math.ceil. Дело в том, что tb.bid\_count представляет собой не целое число, а дробное, хотя и округленного до целого. В общем, просто пока запомните, что в программировании 10 и 10.0 не одно и то же, хотя они и равны. В поле tb.bid\_count как раз содержится число, подобное второму, и при попытке обратится вот так tb.bid[(tb.bid\_count] вылазит сообщение об ошибке. Функция math.ceil как раз преобразует число к первому виду.

В нашем примере OnQuote так же выводит лог, что позволит нам проверить, как работает программа. По этому логу мы можем посмотреть, какие были расчетные цены до ввода заявки, после ввода и после ее отмены. И, кстати, сразу же обнаруживаем недочет:

|  |
| --- |
| 09/16/14 17:02:59    315.3,316.08  09/16/14 17:03:01    315.3,316.07  09/16/14 17:03:02    **316.01(!!!!)**,316.07  09/16/14 17:03:02 OnOrder  09/16/14 17:03:02  заявка 5507321  09/16/14 17:03:02 OnOrder  09/16/14 17:03:02  заявка 5507321  09/16/14 17:03:02    315.3,316.07  09/16/14 17:03:02    315.46,316.07               09/16/14 17:03:03    315.46,316.07 |

Как видим, перед вызовом функции OnOrder у нас крайнее значение неверное (рассчитанное от нашей заявки), но потом оно восстанавливается. Это связано с тем, что сначала приходит событие OnQuote, а уже потом OnOrder. При приходе OnQuote у нас уже есть заявка в стакане, но программа еще не знает, что это наша заявка, она узнает чуть позже, когда придет OnOrder. Другой недочет – при включении программа информация о рассчитанных ценах появляется не сразу, а лишь тогда, когда в стакане что-то измениться. Если меняется редко, то программа может долго не работать при включении, и заработать только спустя некоторое время (когда придет первое изменение стакана). Как исправить эти недочеты я расскажу в следующих уроках. А пока, думаю, информации достаточно.

Полный текст последнего примера приведен в приложении.

Qlua для чайников. Часть 5. Работа с таблица Quik. Поиск заявок. Искусство отладки

На [прошлом уроке](http://robostroy.ru/community/article.aspx?id=783) мы с вами написали заготовку, которая рассчитывает цены выставления наших заявок, на основе крайних цен в стакане (программа считает заданный отступ от этих цен). Если вы не читали прошлый урок, все равно зайдите на него и скачайте приложение – заготовку робота, в этом уроке вам она понадобится.

Как я уже говорил, у нашей программы есть недочеты. Во-первых, из-за того, что события изменения стакана приходит раньше, чем событие выставления заявок, у нас иногда проскакивают неверные цены. Подробнее опять см. прошлый урок. Во-вторых, после запуска у нас робот начинает работать только после того, как произойдут первые изменения в стакане. Как исправить эти недочеты? Давайте подумаем.

Итак, начнем с первой проблемы – рассинхронизация событий. Как вариант решения – все таки сделать поиск заявок, а не читать параметры введенной заявки в событии OnOrder. А сейчас  внимание!!!! – важная информация. Поиск заявок – это одна из задач, которые можно решить при помощи функции SearchItems. Эта функция предназначена для поиска информации в различных таблицах Quik. Таблица заявок – это одна из таких таблиц.  Полный список таких таблиц можно посмотреть здесь <http://help.qlua.org/ch4_5_3.htm>, нас же интересует пока только таблица orders – заявки.

Для поиска заявок пишем функцию find\_orders:

|  |
| --- |
| --Процедура поиска ордеров  **function** find\_orders**()**  **local** NO**=**getNumberOf**(**"orders"**)**        t\_orders **=** SearchItems**(**"orders"**,** 0**,** NO**-**1**,** fn**,** "flags, sec\_code, class\_code"**)**  **if** t\_orders **~=** **nil** **then**  **for** i**=**1**,#**t\_orders**,**1 **do**                    t\_orders\_item**=**getItem**(**"orders"**,** t\_orders**[**i**])**                    remember\_order**(**t\_orders\_item**)**  **end**  **end**  **end** |

Что делает эта функция? Во-первых, она получает количество элементов в таблице orders (количество заявок):

|  |
| --- |
| **local** NO**=**getNumberOf**(**"orders"**)** |

Для чего это нам надо? Для того, что бы вызвать функцию SearchItems, которой необходимо указать диапазон поиска. Диапазон поиска начинается с нуля. Поэтому мы указываем так:

|  |
| --- |
| t\_orders **=** SearchItems**(**"orders"**,** 0**,** NO**-**1**,** fn**,** "flags, sec\_code, class\_code"**)** |

Первый параметр SearchItems – это имя таблицы, в которой мы ищем, в данном случае orders. Второй параметр – начало диапазона поиска, третий конец диапазона поиска, четвертый –поисковая функция, о ней сейчас скажу отдельно. Пятый параметр  — это список полей таблицы ордеров, которые будет анализировать поисковая функция. У каждой таблицы свой набор полей, что касается таблицы orders, то полный список полей можно посмотреть тут <http://help.qlua.org/ch4_6_4.htm>. В нашем же случае используются следующие поля:

* Flags – набор битовых флагов, про них я рассказывал на уроке 4. (ссылка)
* sec\_code – код инструмента.
* class\_code – код класса.

Теперь сама поисковая функция:

|  |
| --- |
| --Поисковая функция  **function** fn**(**flags**,** sec\_code**,** class\_code**)**  **if** sec\_code**==**p\_seccode **and** class\_code**==**p\_classcode **and** bit.band**(**flags**,**1**)>**0 **then**  **return** **true**  **else**  **return** **false**  **end**  **end** |

Эта функция производит анализ входных параметров. Значения входных параметров – это значения полей, перечисленных в пятом параметре функции SearchItems. В частности, мы проверяем поля sec\_code и class\_code – наш ли это инструмент и проверяем первый флаг битовых флагов, который сигнализирует о том, выставленная ли заявка. Работа с флагами так же была описана в уроке 4 (ссылка).

Если заявка найдена, а это значит, что она удовлетворяет заданным условиям – активна и по нашем инструменту – то  эта заявка запоминается. Для этого используется функция remember\_order:

|  |
| --- |
| **function** remember\_order**(**order**)**        p\_file**:write(os.date()..**"  заявка "**..**order**[**"order\_num"**]..**"\n"**)**        --если заявка активна, то запоминаем ее  **if** bit.band**(**order**[**"flags"**],**1**)>**0 **then**              --message("флаг:"..bit.band(order["flags"],4),1)  **if** bit.band**(**order**[**"flags"**],**4**)>**0 **then**                    sell\_order**=**order**[**"order\_num"**]**                    sell\_price**=tonumber(**order**[**"price"**])**                    sell\_count**=tonumber(**order**[**"balance"**])**  **else**                    buy\_order**=**order**[**"order\_num"**]**                    buy\_price**=tonumber(**order**[**"price"**])**                    buy\_count**=tonumber(**order**[**"balance"**])**  **end**  **else**              --если заявка не активна то сбрасываем информацию о заявке  **if** bit.band**(**order**[**"flags"**],**1**)>**0 **then**  **if** bit.band**(**order**[**"flags"**],**4**)>**0 **then**                          sell\_order**=**""                          sell\_price**=**0                          sell\_count**=**0  **else**                          buy\_order**=**""                          buy\_price**=**0                          buy\_count**=**0  **end**  **end**  **end**  **end** |

По сути, эта функция – кусок кода, выдранный из OnOrder и повторяющий ее. Поэтому мы можем просто вызвать remember\_order из функции OnOrder, сократив ее:

|  |
| --- |
| **function** OnOrder**(**order**)**        p\_file**:write(os.date()..**" OnOrder\n"**);**        --сначала проверим, по нашему ли инструменту эта заявка  **if** order**[**"sec\_code"**]==**p\_seccode **and** order**[**"class\_code"**]==**p\_classcode **then**              remember\_order**(**order**)**  **end**  **end** |

А в функции анализа стакана OnQuote добавим вызов find\_orders() :

|  |
| --- |
| **function** OnQuote**(**class\_code**,** sec\_code**)**  **if** class\_code**==**p\_classcode **and** sec\_code**==**p\_seccode **then**                find\_orders**()**  **…**  **…**  **…** |

Теперь, теоретически, у нас должен исправиться недочет. Но не тут-то было. Если посмотреть лог, то можно увидеть, что перед выставлением заявки крайние цены все равно считаются неправильно. Ну что ж, зато есть повод поучиться отлаживать программу. Сразу скажу, что в Квике с отладчиком очень тяжко. Точнее, его нет совсем. Так что придется довольствоваться либо логированием, либо сообщениями.

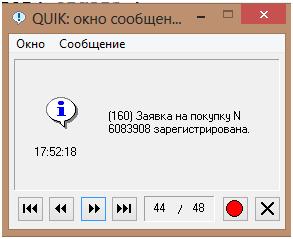
Начнем отладку. Процесс поиска ошибок можно проводить по следующему алгоритму:

1. Выдвигаем гипотезу, где может быть ошибка.
2. Проверяем эту гипотезу.
3. Если ошибка найдена, то исправляем. Иначе см. п. 1.

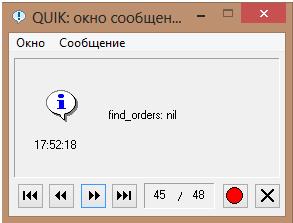
В данном случае логично предположить, что не работает функция поиска. Так что вставляем отладочное сообщение в нее:

|  |
| --- |
| --Процедура поиска ордеров  **function** find\_orders**()**  **local** NO**=**getNumberOf**(**"orders"**)**        t\_orders **=** SearchItems**(**"orders"**,** 0**,** NO**-**1**,** fn**,** "flags, sec\_code, class\_code"**)**        message**(**"find\_orders: "**..tostring(**t\_orders**),**1**)**  **if** t\_orders **~=** **nil** **then**  **for** i**=**1**,#**t\_orders**,**1 **do**                    t\_orders\_item**=**getItem**(**"orders"**,** t\_orders**[**i**])**                    remember\_order**(**t\_orders\_item**)**  **end**  **end**  **end** |

Запускаем. Видим ряд сообщений. Вводим заявку. Сначала видим сообщение о факте ввода заявки:



Затем видим наше сообщение:



Оно говорит о том, что функция SearchItems ничего не нашла. Но потом, с очередным изменением стакана приходит другое сообщение:



Оно уже говорит нам о том, что функция все-таки что-то нашла.

Мы выяснили, что функция SearchItems находит выставленные заявки не сразу, а лишь спустя некоторое время после их выставления.

Итак, нам опять придется искать выход. Еще можно попробовать событие OnTransReply. Оно вызывается, когда происходит транзакция. То есть, если мы выставили заявку, то у нас должно прийти OnTransReply. Теоретически, OnTransReply должно прийти раньше, чем OnOrder. Но лучше все-таки это проверить.  Для этого добавляем в нашего робота вот такую функцию:

|  |
| --- |
| --обработка события транзакции  **function** OnTransReply**(**trans\_reply**)**        nord**=**trans\_reply**[**"order\_num"**]** --Номер заявки  **if** nord**==nil** **or** nord**==**0 **or** nord**==**"0" **then**              message**(**"Заявка не выставилась"**,**1**)**  **return**  **end**  **if** trans\_reply**[**"sec\_code"**]==**p\_seccode **and** trans\_reply**[**"class\_code"**]==**p\_classcode **then**              remember\_order**(**trans\_reply**)**  **end**  **end** |

Но, увы, и это ничего не дало. По-прежнему один такт выдается неверная цена:

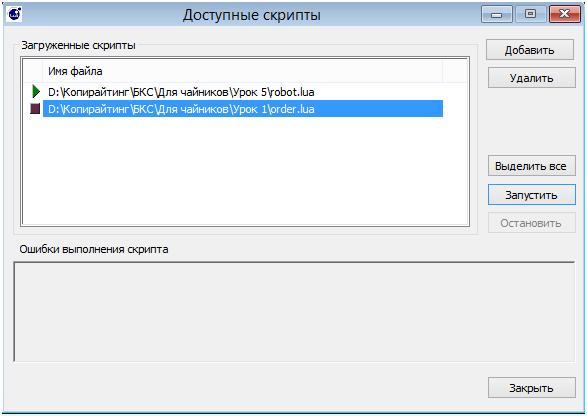
|  |
| --- |
| 09/25/14 14:31:31    300.68,302.2  09/25/14 14:31:31    300.68,302.2  09/25/14 14:31:32    **301.01(!!!!),**302.2  09/25/14 14:31:32 OnOrder  09/25/14 14:31:32  заявка 3217747  09/25/14 14:31:32 OnOrder  09/25/14 14:31:32  заявка 3217747  09/25/14 14:31:33  заявка 3217747  09/25/14 14:31:33    300.68,302.2  09/25/14 14:31:34  заявка 3217747                09/25/14 14:31:34    300.68,302.19 |

Как быть? Продолжать отглючивать (отлаживать).  Воспользуемся тем же методом, что и ранее.  Возможно,  OnTransReply так же приходит после обновления стакана. А может, ошибка и в самом обработчике OnTransReply. Что бы это выяснить, добавляем туда логирование:

|  |
| --- |
| --обработка события транзакции  **function** OnTransReply**(**trans\_reply**)**        p\_file**:write(os.date()..**" OnTransReply\n"**)**        nord**=**trans\_reply**[**"order\_num"**]** --Номер заявки        p\_file**:write(os.date()..**"nord="**..**nord**..**"\n"**)**  **if** nord**==nil** **or** nord**==**0 **or** nord**==**"0" **then**              message**(**"Заявка не выставилась"**,**1**)**  **return**  **end**        p\_file**:write(os.date()..**"сейчас будет проверка условия"**..**trans\_reply**[**"sec\_code"**]..**"   "**..**trans\_reply**[**"class\_code"**]..**"\n"**)**  **if** trans\_reply**[**"sec\_code"**]==**p\_seccode **and** trans\_reply**[**"class\_code"**]==**p\_classcode **then**              p\_file**:write(**"Сейчас запустим remember\_order\n"**)**              remember\_order**(**trans\_reply**)**  **end**  **end** |

Логирование показало, что OnTransReply не запускается вообще. Обычно в таком случае имеет смыл задать вопрос на форуме по qlua (<http://www.quik.ru/forum/lua/>), хотя ответа иногда приходиться ждать несколько часов или дней. Я задал вопрос и получил ответ, что для заявок, введенных вручную, OnTransReply не работает.

Вообще, если вы не планируете вводить заявки вручную во время работы робота, то данная особенность qlua не критична, поэтому сейчас мы не будет останавливаться на решении этой проблемы, а попытаемся протестировать нашего еще не до конца написанного робота вводя заявки программно.  Для этих целей мы можем запустить еще один скрипт, который вводит заявки. Можно воспользоваться тем скрпитом для ввода единичной заявки, что мы писали на первом уроке (<http://robostroy.ru/community/article.aspx?id=773>), только не забудьте поставить наш инструмент и нужные цены.

[](http://robostroy.ru/community/blog/image.ashx?id=2314)

Теперь OnTransReply запускается, но неверная цена все равно проскакивает:

|  |
| --- |
| 09/29/14 12:01:23    304.56,304.78  09/29/14 12:01:48    304.56,304.78  09/29/14 12:01:48    304.56,304.77  09/29/14 12:01:49    304.56,304.77  09/29/14 12:01:49    304.56,304.76  09/29/14 12:01:49    304.56,304.76  09/29/14 12:01:51    304.56,304.76  09/29/14 12:01:51    304.56,304.76  09/29/14 12:01:52    304.56,304.76  09/29/14 12:01:52    304.56,304.76  09/29/14 12:02:09 OnTransReply  09/29/14 12:02:09nord=1848863  09/29/14 12:02:09сейчас будет проверка условияPOLY   TQBR  Сейчас запустим remember\_order  09/29/14 12:02:09  заявка 1848863  09/29/14 12:02:09    304.61,304.76  09/29/14 12:02:09 OnOrder  09/29/14 12:02:09  заявка 1848863  09/29/14 12:02:09 OnOrder  09/29/14 12:02:09  заявка 1848863 |

Чтобы с этим разобраться, придется добавить логирования и в remember\_order, из OnTransReply логирование можно убрать. Что мы выясняем из лога:

|  |
| --- |
| 09/29/14 12:37:22    304.13,305.84  09/29/14 12:37:22    304.13,305.84  09/29/14 12:37:23    304.13,305.84  09/29/14 12:37:23    304.13,305.84  09/29/14 12:37:24    304.13,305.84  09/29/14 12:37:24    304.13,305.84  09/29/14 12:37:24    304.13,305.84  09/29/14 12:37:25    304.13,305.84  09/29/14 12:37:25    304.13,305.84  09/29/14 12:37:26    304.13,305.84  09/29/14 12:37:26 OnTransReply  09/29/14 12:37:26 remember\_order:  заявка 2451199  09/29/14 12:37:26 remember\_order:  заявка активна  09/29/14 12:37:26 remember\_order:  заявка на покупку  09/29/14 12:37:26 remember\_order:  **buy\_price=305(!!!!!)**  09/29/14 12:37:26    **305.01**(!!!!!),305.84  09/29/14 12:37:26 OnOrder  09/29/14 12:37:26 remember\_order:  заявка 2451199  09/29/14 12:37:26 remember\_order:  заявка активна  09/29/14 12:37:26 remember\_order:  заявка на покупку  09/29/14 12:37:26 remember\_order:  buy\_price=305  09/29/14 12:37:26 OnOrder  09/29/14 12:37:26 remember\_order:  заявка 2451199  09/29/14 12:37:26 remember\_order:  заявка активна  09/29/14 12:37:26 remember\_order:  заявка на покупку  09/29/14 12:37:26 remember\_order:  buy\_price=305  09/29/14 12:37:27 remember\_order:  заявка 2451199  09/29/14 12:37:27 remember\_order:  заявка активна  09/29/14 12:37:27 remember\_order:  заявка на покупку  09/29/14 12:37:27 remember\_order:  buy\_price=305  09/29/14 12:37:27    304.13,305.84  09/29/14 12:37:27 remember\_order:  заявка 2451199 |

А выясняем мы то, что заявка качественно запоминается (обратите внимание на пометки в логе). Но, тем не менее, цена почему то на один тик становиться неправильной. Может, OnTransReply тоже позже, чем OnQuote приходит? Добавим в OnQuote логирование.

И тут получается интересное «кино».  Цены остаются правильные, вот, пожалуйста, смотрите лог:

|  |
| --- |
| 09/29/14 12:43:06    303.72,303.89  09/29/14 12:43:20 OnQuote  09/29/14 12:43:20    303.72,303.89  09/29/14 12:43:22 OnTransReply  09/29/14 12:43:22 remember\_order:  заявка 2553806  09/29/14 12:43:22 remember\_order:  заявка активна  09/29/14 12:43:22 remember\_order:  заявка на покупку  09/29/14 12:43:22 remember\_order:  buy\_price=303.8  09/29/14 12:43:22 OnOrder  09/29/14 12:43:22 remember\_order:  заявка 2553806  09/29/14 12:43:22 remember\_order:  заявка активна  09/29/14 12:43:22 remember\_order:  заявка на покупку  09/29/14 12:43:22 remember\_order:  buy\_price=303.8  09/29/14 12:43:22 OnOrder  09/29/14 12:43:22 remember\_order:  заявка 2553806  09/29/14 12:43:22 remember\_order:  заявка активна  09/29/14 12:43:22 remember\_order:  заявка на покупку  09/29/14 12:43:22 remember\_order:  buy\_price=303.8  09/29/14 12:43:22 OnQuote  09/29/14 12:43:22 remember\_order:  заявка 2553806  09/29/14 12:43:22 remember\_order:  заявка активна  09/29/14 12:43:22 remember\_order:  заявка на покупку  09/29/14 12:43:22 remember\_order:  buy\_price=303.8  09/29/14 12:43:22    303.72,303.89  09/29/14 12:43:22 OnQuote  09/29/14 12:43:22 remember\_order:  заявка 2553806  09/29/14 12:43:22 remember\_order:  заявка активна  09/29/14 12:43:22 remember\_order:  заявка на покупку  09/29/14 12:43:22 remember\_order:  buy\_price=303.8  09/29/14 12:43:22    303.72,303.88 |

Иными словами, мы ввели заявку по цене 303.8, но нижняя крайняя цена осталась 303.72, что  правильно.

Спрашивается, почему так? Видимо, логирование создает задержку.

А идея! Давайте в OnQuote просто сделаем задержку. Посмотрим, что получится. Лишнее логирование тогда тоже уберем.

Сделать задержку можно командой sleep:

|  |
| --- |
| **function** OnQuote**(**class\_code**,** sec\_code**)**  **if** class\_code**==**p\_classcode **and** sec\_code**==**p\_seccode **then**                sleep**(**100**)**  **…** |

Время задержки я поставил 0.1, этого хватило. Но вообще, его следует выбирать эмпирическим путем – если работает с меньшей задержкой – ставьте меньше, что бы робот не «тормозил» систему.

И, прежде чем я закончу сегодняшний большой урок, решим еще одну проблему: робот начинает работать только с первым изменением стакана. Это проблема решается очень просто: принудительным вызовом OnQuote в функции main:

|  |
| --- |
| **function** main**()**        p\_file**:write(os.date()..**" main\n"**)**        OnQuote**(**p\_classcode**,** p\_seccode**)**  **while** is\_run **do**              sleep**(**2000**)**  **end**  **end** |

На этом я заканчиваю урок, конечный вариант написанной на этом уроке программы см.  в приложении.

Прикрепленные файлы

**·**   [Приложение к уроку 5.rar](javascript:Community.API.Post.Attachment.Open(282))