Release Flash

Price History and Market Pattern Script Libraries

US5797-RFL-1000 (3.72)

The basic task of most trading algorithms is to detect a specific situation in the market which can be exploited with managed orders. For this the algorithms search for patterns in the market micro-structure.

To make this easy, actant has developed a set of script libraries that allow querying for market change patterns from any script. There are three libraries:

- **Two Level Price History** makes it possible to access the previous value and the time when the value was last updated for the BBO and last trades. This is the basis for detecting change patterns in the market.
- **Market Pattern** is a library that offers pre-defined market patterns such as “last trade done on bid” or “market flipped”.
- **Timestamp** offers basic operations on timestamps with an accuracy of one microsecond.

This release flash shows how the new script libraries are used to program trading algorithms and contains the complete reference of the new script library functions.

Developing an Algo using the new Script Libraries

Lets write a Scanner that detects a support level in a market. The best start is to write down in plain English what the condition is for a support level.

```
if the ask has moved down,
    the BBO is only one tick wide,
    the bid doesn't change,
    there are twice as many contacts traded on the ask than on the bid
    and there are at least 20 contracts traded on the ask then
    we have a support at the bid
```

Now let's look at the individual conditions and how to express them using the new libraries.

the ask has moved down	`hist::ask_down()`
the BBO is only one tick wide	`pattern::bbowidthinticks() == 1`

the bid doesn't change	`hist::bid_unchangedsince(`
	`hist::ask_timestamp())`

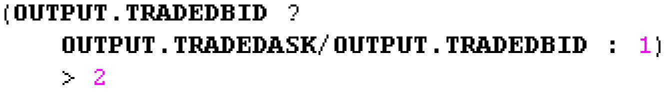
Before we can test the next expressions we have to accumulate the number of contracts that are traded on the bid and ask side of the market. This is done by testing for a tick (the term tick is used for a trade in the market, it's synonym to an entry in the time/sales sheet).

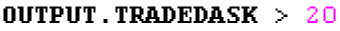
```
if hist::tick_changed() and OUTPUT.TICKTIMESTAMP < hist::tick_timestamp() then
    if LAST == BID then OUTPUT.TRADEDBID += NLAST; end
    if LAST == ASK then OUTPUT.TRADEDASK += NLAST; end
end
OUTPUT.TICKTIMESTAMP = hist::tick_timestamp();
```

We compare the LAST price to BID and ASK and increment the OUTPUT variable that is used to store the quantity. OUTPUT variables have a dual purpose. They are used to display values on the GUI, but can also be used to store state between script executions.

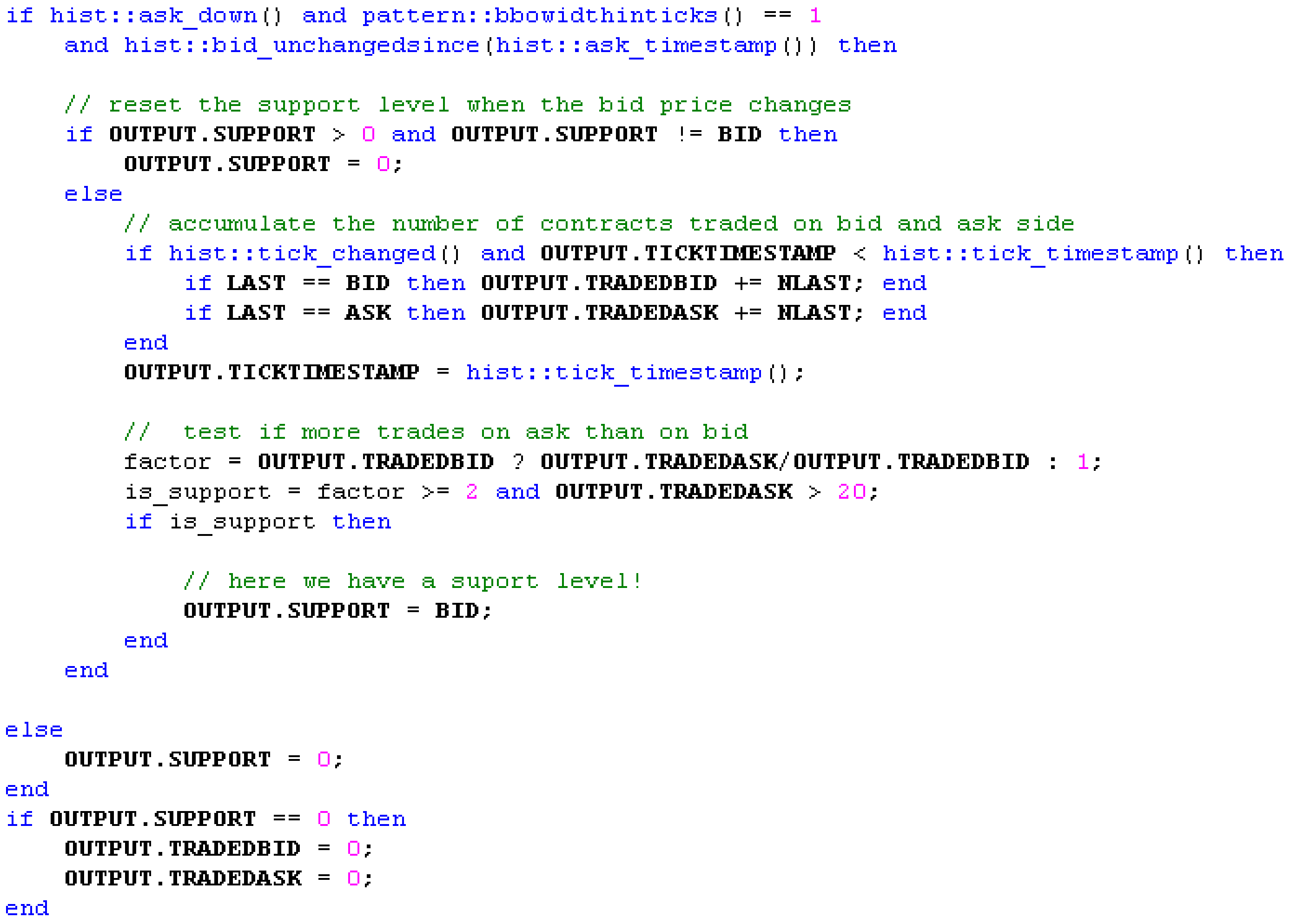
Because the script can be executed multiple times within a short time interval, we add an extra protection so that a tick is not accidentally counted twice.

The remaining conditions can now be written as:

there are twice as many trades on the ask than on the bid	
	`(OUTPUT.TRADEDBID ?`
	`    OUTPUT.TRADEDASK/OUTPUT.TRADEDBID : 1)`
	`    > 2`

there are at least 20 contracts traded on the ask	 `OUTPUT.TRADEDASK > 20`

Here’s the complete Algo. It looks surprisingly like the original English text.



```
if hist::ask_down() and pattern::bbowidthinticks() == 1
    and hist::bid_unchangedsince(hist::ask_timestamp()) then

    // reset the support level when the bid price changes
    if OUTPUT.SUPPORT > 0 and OUTPUT.SUPPORT != BID then
        OUTPUT.SUPPORT = 0;
    else
        // accumulate the number of contracts traded on bid and ask side
        if hist::tick_changed() and OUTPUT.TICKTIMESTAMP < hist::tick_timestamp() then
            if LAST == BID then OUTPUT.TRADEDBID += NLAST; end
            if LAST == ASK then OUTPUT.TRADEDASK += NLAST; end
        end
        OUTPUT.TICKTIMESTAMP = hist::tick_timestamp();

        //  test if more trades on ask than on bid
        factor = OUTPUT.TRADEDBID ? OUTPUT.TRADEDASK/OUTPUT.TRADEDBID : 1;
        is_support = factor >= 2 and OUTPUT.TRADEDASK > 20;
        if is_support then

            // here we have a suport level!
            OUTPUT.SUPPORT = BID;
        end
    end

else
    OUTPUT.SUPPORT = 0;
end
if OUTPUT.SUPPORT == 0 then
    OUTPUT.TRADEDBID = 0;
    OUTPUT.TRADEDASK = 0;
end
```

Using bespoke libraries in this way not only makes the resulting script code short and easy to read, it also makes it more reliable because it is based on fully tested components.

Script Library: Two Level Price History

Script variables always give you access to the latest value. E.g. ASK is the current value of the best offer in the market. Whenever there is a new offer coming from the exchange through the data feed, the value of ASK is overwritten.

The basic idea of the two level price history script library is that the system also saves the previous value and the time when the value has last changed. The accuracy of the timestamp is microseconds. With this information it is now possible to query how long time ago the price has last time changed and in which direction the price has moved.

All query functions are implemented for the BBO price and quantity as well as for the last price.

Functions	Description

`hist::bid_changed`
`hist::nbid_changed`
`hist::ask_changed`
`hist::nask_changed`
`hist::last_changed`
`hist::nlast_changed`
	The function returns true when the value has changed later than `ts::scriptdelay` before the script execution. We can't assume that the script is executed after every single change, so the concept of script reason doesn't work accurately. There may be multiple reasons that led to the recalculation. Querying the reason inside the script leads to more stable and more precise business logic. Instead of testing for "`if reason == "BBO" then`" the script can test for "`if hist::bid_changed() or hist::ask_changed() then`".

`hist::bid_unchangedsince`
`hist::nbid_unchangedsince`
`hist::ask_unchangedsince`
`hist::nask_unchangedsince`
`hist::last_unchangedsince`
`hist::nlast_unchangedsince`
	The function returns 1 when the value has not changed after the timestamp passed as parameter.

`hist::bid_previous`
`hist::nbid_previous`
`hist::ask_previous`
`hist::nask_previous`
`hist::last_previous`
`hist::nlast_previous`
	The function returns the previous value before the last change.

`hist::bid_timestamp`
`hist::nbid_timestamp`
`hist::ask_timestamp`
`hist::nask_timestamp`
`hist::last_timestamp`
`hist::nlast_timestamp`
	The function returns the timestamp when the value has last time changed. This allows checking if values changed simultaneously or sequentially. Because of the delay between the exchange and the application as because exchange architectures may disseminate prices through different channels, simultaneous should not be tested as equality. The ts script library provides a set of queries that can be used. In that library, equality of events is defined as not further apart as `ts::accuracy`.

`hist::bid_up`	The function returns true when the current and the

`hist::nbid_up`
`hist::ask_up`
`hist::nask_up`
`hist::last_up`
`hist::nlast_up`
	previous value are both valid and the current value is greater than the previous value.

`hist::bid_down`
`hist::nbid_down`
`hist::ask_down`
`hist::nask_down`
`hist::last_down`
`hist::nlast_down`
	The function returns true when the current and the previous value are both valid and the current value is smaller than the previous value.

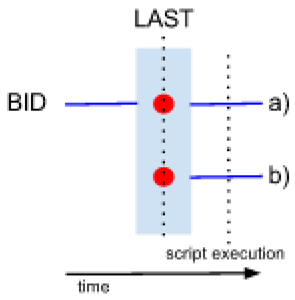
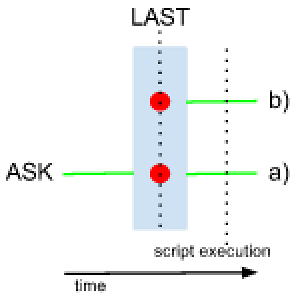
`hist::bid_appeared`
`hist::nbid_appeared`
`hist::ask_appeared`
`hist::nask_appeared`
`hist::last_appeared`
`hist::nlast_appeared`
	The function returns true when the current value is valid and the previous value was invalid. This can be used to check when the value appears the first time in the morning. When the history isn't relevant, it's simpler to test for Invalid values with isvalid() e.g. "`isvalid(BID)`".

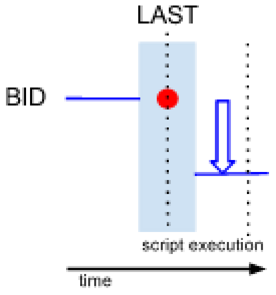
`hist::bid_gone`
`hist::nbid_gone`
`hist::ask_gone`
`hist::nask_gone`
`hist::last_gone`
`hist::nlast_gone`
	The function returns true when the current value isn't valid and the previous value was valid. When the history isn't relevant, it's simpler to test for Invalid values with isvalid() e.g. "`not isvalid(BID)`".

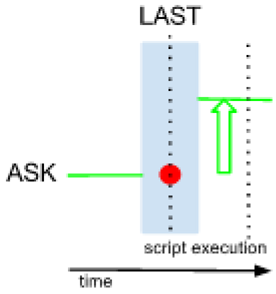
`hist::tick_changed`	The function returns true when there was a new tick in the Time/Sales sheet. It is detected by a change in the last price, quantity or total volume.
`hist::tick_timestamp`	The function returns the timestamp of the last tick int the Time/Sales sheet. It is detected by a change in the last price, quantity or total volume.

Script Library: Market Patterns

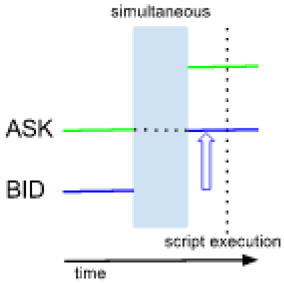
This library is based on the two level price history and the timestamp library. It supports recognizing common market price update patterns.

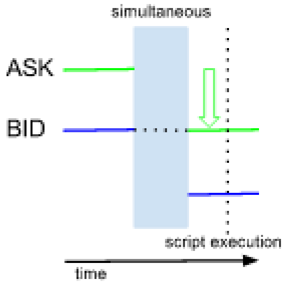
Function	Description	Picture
`pattern::tradeonbid`	The function returns 1 when the last trade was done on the bid level, 0 otherwise. Last trade means that there was a tick update recently (`hist::tick_changed`). It is possible that the bid price also changed simultaneously. The function returns true when the LAST is the same as the previous value of the bid (a) or the current value of bid (b).	
`pattern::tradeonask`	The function returns 1 when the last trade was done on the ask level, 0 otherwise. Last trade means that there was a tick update recently (`hist::tick_changed`). It is possible that the ask price also changed simultaneously. The function returns true when the LAST is the same as the previous value of the ask (a) or the current value of ask (b).	

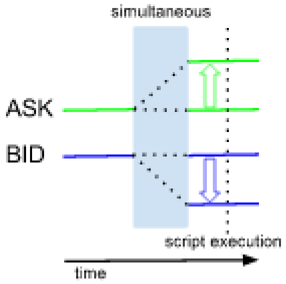
`pattern::`
`tradethroughbid`
	The function returns 1 when the last trade trades on the bid level and the bid in the market subsequently is lower or has gone, 0 otherwise.This typically happens when the complete bid quantity has traded.
	
	Last trade means that there was a tick update recently (`hist::tick_changed`). The bid price must have changed simultaneously to the tick and be lower (`hist::bid_down`) or gone (`hist::bid_gone`). The LAST must be equal or lower to the previous value of bid.
		

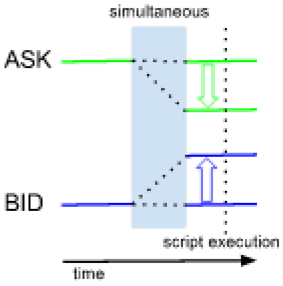
`pattern::`
`tradethroughask`
	The function returns 1 when the last trade trades on the ask level and the ask in the market subsequently is higher or has gone, 0 otherwise.This typically happens when the complete ask quantity has traded.
	
	Last trade means that there was a tick update recently (`hist::tick_changed`). The ask price must have changed simultaneously to the tick and be higher (`hist::ask_up`) or gone (`hist::ask_gone`). The LAST must be equal or higher to the previous value of ask.
		

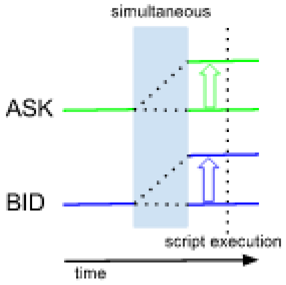
`pattern::widthinticks`	Returns the width of the spread between two prices in number of market price steps (ticks). The step size is taken from the first parameter. So on markets with non-constant price steps (e.g. US options 5 cents below	

	3 dollars, ten above) the return number of ticks indicated can be larger than in reality (US options `pattern::widthinticks(2.9,3.1)` will return 4 instead of 3). The method does not depend on the order of the parameters, so `pattern::withinticks(BID,ASK)` returns the same value as `pattern::withinticks(ASK,BID)`.	
`pattern::bbowidthinticks`	Returns the width of the BBO (ask - bid) in number of price steps. The function returns `novalue()` when either of the prices is missing. The function is the same as `pattern::withinticks(BID,ASK)`. This function was added because in markets that are just one tick wide, traders typically use different tactics than in markets that are several ticks wide.	
`pattern::bboflippedup`	Returns true when the bid and ask price both change and the new bid is equal to the previous ask price.	

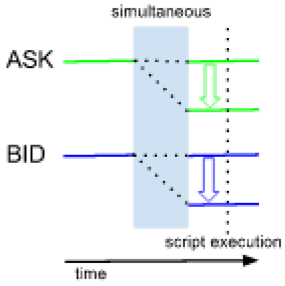
<table>
<tr>
<td>pattern::
bboflippeddown</td>
<td>Returns true when the bid and ask price both change and the new ask is equal to the previous bid price.</td>
<td>


</td>
</tr>
<tr>
<td>pattern::bbowidened</td>
<td>Returns true when either the bid moved down, the ask moved up or the bid moved down and the ask moved up. When the ask moves down or the bid moves up, the function returns zero.</td>
<td>


</td>
</tr>
<tr>
<td>pattern::bbonarrowed</td>
<td>Returns true when either the bid moved up, the ask moved down or the bid moved up and the ask moved down. When the ask moves up or the bid moves down, the function returns zero.</td>
<td>


</td>
</tr>
<tr>
<td>pattern::bbomovedup</td>
<td>Returns true when either the bid, the ask or both moved up. If any price moves down, the function returns zero.</td>
<td>


</td>
</tr>
</table>

`pattern::bbomoveddown`	Returns true when either the bid, the ask or both moved down. If any price moves down, the function returns zero.	

Script Library: Timestamps

This library supports the market micro-structure libraries with timestamp operations and measurements. The unit of all operations is seconds, the measurement accuracy is microseconds. The price updates received from the price feeds are all time stamped at the same and well defined place.

Because all updates from the exchange happen asynchronously and the delays differ between the price feeds, there are two customisable time values that can be used to fine-tune the behavior of the system:

- `ts::accuracy` is used to determine if two updates happened simultaneously. It is set to 100µs by default and can be changed with a call to `ts::setaccuracy()`. The setting is stored per instrument.
- `ts::scriptdelay` is used as the maximum time that may elapse between the price update and the script execution to still consider the change as having happened 'now'. The value is set to 1ms by default, but can be overwritten with `ts::setscriptdelay()`. The setting is stored per instrument.

Function	Description	Params	Example
`ts::accuracy`	Value that is used to define time equality between price updates in seconds. Two timestamps that don't differ more than by that level are regarded as equal. Simultaneousness can be tested with `ts::simultaneous`. The default value for accuracy is 0.000100 (100 µs), but it can be changed (per instrument) with `ts::setaccuracy`.		
`ts::scripttime`	Returns the timestamp taken at the start of the current script		

	execution.		
`ts::scriptdelay`	Value that is used to determine if a price update can be regarded as being within the same logical context as the script execution. It is used for `ts::isnow`. The default value for `scripttime` is 0.001000 (1 ms). It can be changed with `ts::setscripttime`.		
`ts::elapsed`	Returns the time elapsed between the timestamp passed and the start of the script execution in seconds. When the timestamp lies in the future or is invalid (0), the function returns 0.	timestamp	`ts::elapsed(hist::bid_timestamp())`
`ts::simultaneous`	Returns 1 when the difference between two timestamps is smaller or equal to `ts::scripttime`, 0 otherwise When any of the two timestamps isn't set (0), the function returns 0.	timestamp 1, timestamp 2	`ts::simultaneous(hist::bid_timestamp(), hist::ask_timestamp())`

`ts::isnow`	Returns 1 when the elapsed time since the timestamp passed isn't larger than `ts::scriptdelay`, 0 otherwise. This is equivalent to `ts::elapsed(timestamp) <= ts::scriptdelay()`.
	When the timestamp passed isn't set (0), the function returns 0. When the timestamp lies in the future, the function returns 1.
		timestamp	`ts::isnow(hist::bid_timestamp())`
			`// equivalent to hist::bidchanged()`

`ts::isvalid`	Returns 1 when the timestamp passed is non-zero, 0 otherwise.	timestamp	`ts::isvalid(hist::bid_timestamp())`

	This happens when the timestamp is never set (e.g. `hist::bid_timestamp` before the first bid update arrives).		

`ts::setaccuracy`	Overwrite the accuracy for the given instrument. Passing a parameter <= 0 will set the accuracy back to the default value 0.000100 (100 μs).
	This method doesn't return a value and cannot be called inside an expression.
	This method can be used to adjust to the price feed speed. If update bursts can come in a higher frequency, the accuracy can be lowered. If the price feed is slow and e.g. last and BBO updates are dissynchronized, the value may have to be increased.
	Careful: There is just one accuracy per instrument, setting different values from different scripts will result in conflicts.
		time difference in seconds	`ts::setaccuracy( 0.001)`
			`// 1 millisecond`

`ts:: setscriptdelay`	Overwrite the scriptdelay for the given instrument. Passing a parameter <= 0 will set the scriptdelay back to the default value 0.001000 (1 ms).
	This method doesn't return a value and cannot be called inside an expression.
	This time has to be increased when the system is overloaded and the typical time between the price update and the scheduling
		time difference in seconds	`ts::setscriptdelay (0.005)`
			`// 5 milliseconds`

	of the business logic transaction is growing. **Careful**: There is just one scriptdelay per instrument, setting different values from different scripts will result in conflicts.		