BRINOVS Block Guide

Every Deriv Bot block and every BRINOVS block, explained. https://brinovsdigits.vercel.app/guide

Contents

1. Official blocks: Trade definition (the 1. Trade parameters block) (14 entries)
2. Official blocks: Purchase conditions (before purchase) (4 entries)
3. Official blocks: Sell conditions (during purchase) (4 entries)
4. Official blocks: Restart trading conditions (after purchase) (4 entries)
5. Official blocks: Tick and candle analysis (17 entries)
6. Official blocks: Indicators (16 entries)
7. Official blocks: Tools (the Utility menu) (16 entries)
8. Standard blocks: logic, math, text, lists, variables and functions (57 entries)
9. BRINOVS blocks: Execution and purchasing (1 to 9) (9 entries)
10. BRINOVS blocks: Stake management (10 to 13) (4 entries)
11. BRINOVS blocks: Risk management (14 to 19) (6 entries)
12. BRINOVS blocks: Entry filters and restart (20 to 24) (7 entries)
13. BRINOVS blocks: Analysis (25 to 33) (16 entries)
14. BRINOVS blocks: Sell conditions and market selection (34 to 37) (5 entries)
15. BRINOVS blocks: Control, state and statistics (38 to 44) (9 entries)
16. BRINOVS blocks: Alerts, testing and licensing (45 to 51) (7 entries)

What is this guide?

BRINOVS is a free trading page and **bot builder** for Deriv. A trading bot is a set of instructions that places trades for you. In the Bot Builder you do not write code. You snap coloured **blocks** together, like puzzle pieces. This guide explains every block you can use:

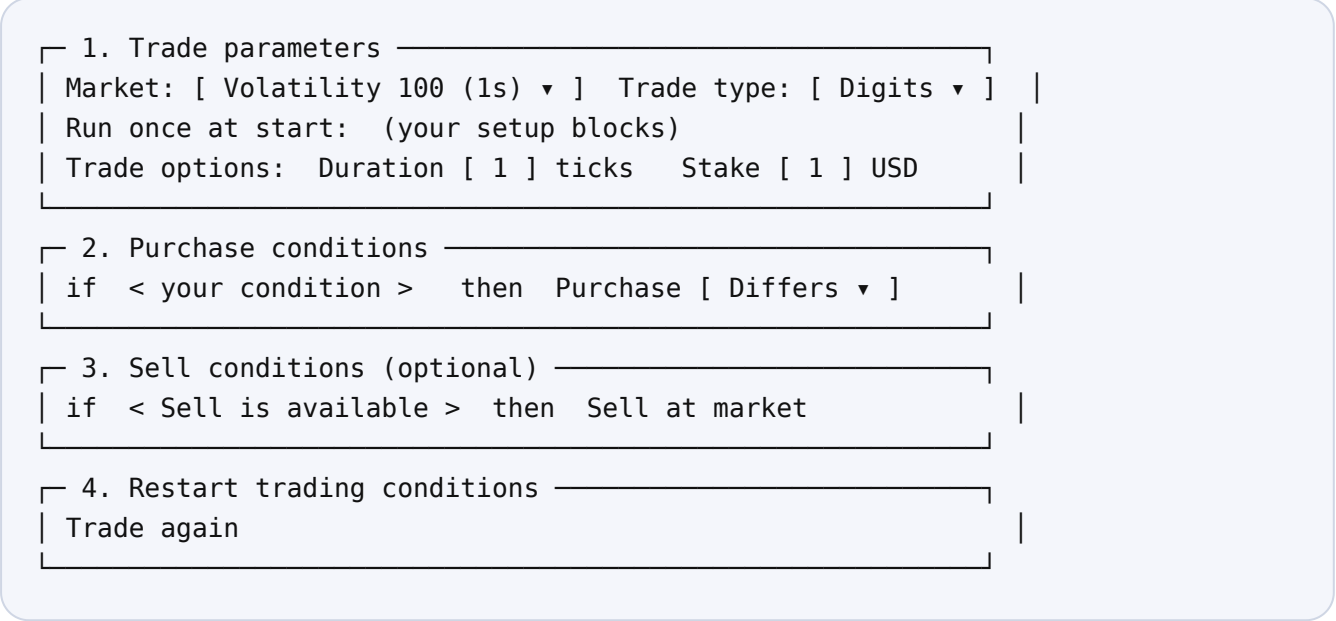
- the **official blocks** that come with Deriv's Bot Builder, and the standard logic and math blocks;
- the **51 BRINOVS blocks**, our own additions: safer buying, several contracts at once, stake plans, loss limits, entry filters, analysis, statistics, alerts and licensing.

You do not need to read it in order. Open the page for the group you care about, or use the search box on the guide home page to find a block by name.

How a bot runs

Every bot is built from four big "mother" blocks. They run in this order, round after round:

1. **Trade parameters.** Runs once when you press Run. It chooses the market, the kind of trade, the duration and the stake. Things you want to happen only once at the start (setting limits, switching on a test mode) go in its "Run once at start" part.
2. **Purchase conditions.** The bot waits here, tick after tick, until your condition is true. Then it buys the contract with the Purchase block.
3. **Sell conditions.** While the contract is open, the bot can check whether to sell it early. This part is optional.
4. **Restart trading conditions.** After the contract ends, the bot decides whether to trade again. You put the "Trade again" block here, usually behind a check such as "stop after a loss streak".



A **tick** is one new price from the market. On a one-second index a tick arrives about every second.

The three shapes of block

- **Action block.** Does something: buys, sells, sets a value, shows a message. It snaps above and below other action blocks, like a brick in a wall.
- **Value block.** Gives back a number, a piece of text or a list. It plugs into a round socket on another block.

- **True/false block.** Gives back "true" or "false". It plugs into the socket of an `if` or any block that asks a yes/no question.

In the Bot Builder, value blocks and true/false blocks both have rounded ends. Where a block goes is always told in its **Where it goes** line.

How to read an entry

Every block in this guide has the same lines:

- **Kind:** whether it is an official Deriv block or a BRINOVS block (with its number from 1 to 51).
- **Type id:** the block's internal name. You never type it; it is here so you can search for it and tell us about it.
- **Shape:** action, value or true/false.
- **Where it goes:** the mother block or the socket it belongs in.
- **What it does** and **How it works:** the idea first, then the mechanics.
- **Inputs** and **Gives back:** every box and dropdown, and what comes out.
- **Example:** a text "screenshot" of the block as it looks in the Bot Builder.
- **Good to know:** limits, safe defaults and common mistakes.

How to read the text screenshots

The examples are drawn with plain characters so they load fast and work everywhere:

You see	It means
`┌─ 2. Purchase conditions ─┐`	A mother block (a big section)
`[ 5 ]`	A box where you type a number or text
`[ Rise ▾ ]`	A dropdown: tap it and choose
`( Last digit )`	A value block plugged into a socket
`< Last digit ≥ 5 >`	A true/false block plugged into a socket
`□` and `☑`	A tick box, off and on

Where the BRINOVS blocks are

In the Bot Builder, open the **Blocks menu**. After Deriv's own groups (Trade parameters, Purchase conditions, Sell conditions, Restart trading conditions, Analysis, Utility) you will find a folder called **BRINOVS**. It has 12 sub-folders, one for each part of this guide. A few BRINOVS blocks that are useful in almost every bot are already placed on a fresh Bot Builder, some of them switched off until you enable them. Your saved bots are never changed.

Using a ready-made bot

The **Plug & Play** tab has ready-made bots. Free bots load into the Bot Builder with one tap. A premium bot opens with its blocks fixed in place: you can change its numbers (stake, take profit, stop loss) and run it, but you cannot move or delete its blocks.

Safety first

- **Always test on a demo account first.** A demo account uses practice money.
- No block, indicator or strategy can promise profit. Markets are random, and many Deriv contracts pay less than 100% on a win.
- Set a stake you can afford to lose and use a loss limit.
- Blocks that mention risk limits help you control losses; they cannot remove the risk of trading.

Short glossary

- **Market (symbol):** what you trade, for example Volatility 100 (1s) Index.
- **Contract:** one trade, for example "Rise" over 5 ticks.
- **Stake:** the money you put into one contract.
- **Payout:** what the contract pays if you win, stake included.
- **Duration:** how long a contract runs, in ticks, seconds, minutes and so on.
- **Barrier / prediction:** the price level or digit your contract is compared with.
- **Last digit:** the last number of the latest price. Digit contracts are about this digit.
- **Take profit / stop loss:** a limit to stop after a target win or a maximum loss.
- **Martingale:** raising the stake after a loss to try to win it back. It can lose money very fast.
- **Proposal:** the price and payout Deriv offers you for a contract before you buy.

- **Demo account:** a practice account with virtual money.

Official blocks: Trade definition (the 1. Trade parameters block)

Every bot you build in the Bot Builder is made of four big boxes called root blocks. The first one is **1. Trade parameters**. It is where you tell the bot what to trade (the market), how to trade it (the trade type and contract type), and for how long and for how much money (the Trade options block). In the menu on the left these blocks sit under "Trade parameters". A fresh Bot Builder already has this box filled in, so most of the time you only change the choices.

A few words used all through this guide. A **tick** is one price update from a market (they arrive about every second or two, depending on the market). A **contract** is one single trade that you buy. The **stake** is the money you put into one contract. The **payout** is the money Deriv pays you if the contract wins (it includes your stake). A **barrier** is a price level the contract is judged against. A **proposal** is Deriv's live price offer for a contract.

How a bot runs, step by step. The four root blocks run in a fixed order, and then the bot goes around again:

1. **Start.** The bot connects, then runs everything in "Run once at start:" one time only.
2. **1. Trade parameters.** At the start of every round the bot reads the Trade options block (duration, stake and so on). Because it is read again each round, you can change the stake from one round to the next with a variable.
3. **2. Purchase conditions.** The bot waits for a new tick, then runs this box. It runs again on every new tick until a Purchase block runs.
4. **Purchase.** The bot buys the contract.
5. **3. Sell conditions.** While the contract is open, the bot runs this box on every update of the contract. It is optional. A Sell at market price block inside it can close the contract early. If you leave it empty, the contract simply runs until it ends.
6. **4. Restart trading conditions.** When the contract has ended, the bot runs this box once. If it reaches a Trade again block, the bot goes back to step 2 and a new round starts. If it does not, the bot stops.

A bot needs the 1. Trade parameters box, a 2. Purchase conditions box with a Purchase block in it, and a Trade options block that matches the trade type. If one of them is missing or switched off, the bot will not start and the journal says which one. Each of the four root blocks can only be used once.

1. Trade parameters

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition`
- **Shape:** Root block (a big box with no notch above or below; other blocks go inside it)
- **Where it goes:** On its own on the workspace. Only one copy is allowed. It cannot be deleted, and it is already there in a fresh Bot Builder.
- **What it does:** Holds all the settings of your trade. It decides which market is traded, which kind of contract is bought, and what happens when something goes wrong. It also holds the stake and duration through the Trade options block.
- **Inputs:**
  - The first group (no title) is six locked blocks: Market, Trade Type, Contract Type, Default Candle Interval, Restart buy/sell on error, Restart last trade on error (described below).
  - Two slots: "Run once at start:" (blocks that run one time when the bot starts, such as setting a variable to your first stake) and "Trade options:" (the Trade options block of your trade type).
- **How it works:** When the bot starts, it reads your choices and tells the trade engine which market, contract types and candle interval to use, and the two restart switches. Then it runs the "Run once at start:" blocks. At the start of every round it runs the Trade options block, which sends the stake and duration. If you pick "Both" as the contract type, the bot prepares both sides of the trade.
- **Example:** A digits bot with a first stake set once, then 5 ticks and a stake of 1.

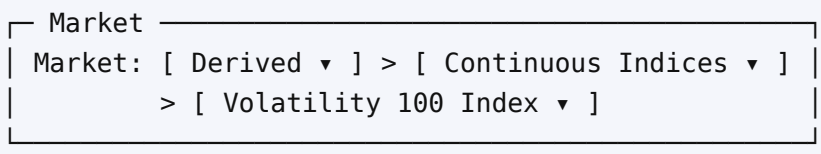
```
┌─ 1. Trade parameters ──────────────────────────────┐
│ Market: [ Derived ▾ ] > [ Continuous Indices ▾ ] │
│         > [ Volatility 100 Index ▾ ]              │
│ Trade Type: [ Digits ▾ ] > [ Even/Odd ▾ ]         │
│ Contract Type: [ Both ▾ ]                         │
│ Default Candle Interval: [ 1 minute ▾ ]           │
│ Restart buy/sell on error (disable for better     │
│   performance): ☐                                 │
│ Restart last trade on error (...): ☑              │
│ Run once at start:                                │
│   set [ stake ▾ ] to ( 1 )                        │
│ Trade options:                                    │
│   Duration: [ Ticks ▾ ] ( 5 )                     │
│   Stake: USD ( 1 )                                │
└────────────────────────────────────────────────────┘
```

- **Good to know:**
  - You must be logged in. If not, the bot stops with "Please log in".

- Only blocks whose type starts with trade_definition_ can sit in the locked group. Anything else is pushed out. If a mandatory block is missing or disabled, the journal says so, for example "The Trade parameters block is mandatory and cannot be deleted/disabled."

Market

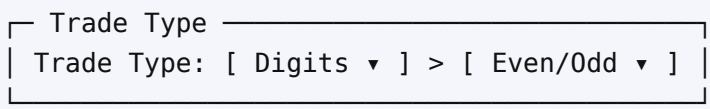
- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_market`
- **Shape:** Action block (a locked helper block)
- **Where it goes:** The first slot inside 1. Trade parameters. It is locked: you cannot move it, delete it or disable it. If you drag it out, it jumps back.
- **What it does:** Chooses the market the bot watches and trades. You pick in three steps: the market, then the submarket, then the symbol. The symbol is the exact price feed, such as Volatility 100 Index.
- **Inputs:**
  - Market dropdown (for example Derived, Forex). The choices come from Deriv's live list.
  - Submarket dropdown (for example Continuous Indices). It refills when you change the market.
  - Symbol dropdown (for example Volatility 100 Index). It refills when you change the submarket.
- **How it works:** The block makes no code of its own. The symbol you pick is handed to the trade engine when the bot starts, and the bot then follows the ticks of that symbol. When you change the symbol, the chart switches to it, and the Trade Type list is refreshed to what Deriv offers on that symbol.
- **Example:** Picking a Volatility index.



- **Good to know:**
  - A few Deriv markets are hidden in this builder (the energy group and four forex pairs).
  - The lists load from Deriv. If you are offline, the dropdowns may be empty until you are connected.
  - If the bot starts with no symbol, it stops with "Underlying market is not selected".

Trade Type

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_tradetype`
- **Shape:** Action block (a locked helper block)
- **Where it goes:** Second slot inside 1. Trade parameters. Locked like Market.
- **What it does:** Chooses the kind of trade, in two steps: a category (such as Up/Down, Digits, Multipliers, Accumulators) and then the trade type inside that category (such as Rise/Fall, Even/Odd, Over/Under, Matches/Differs). A trade type is a pair of opposite contracts.
- **Inputs:**
  - Category dropdown. The categories Deriv can name here are Up/Down, Touch/No Touch, In/Out, Asians, Digits, Reset Call/Reset Put, High/Low Ticks, Only Ups/Only Downs, Call Spread/Put Spread, Multipliers and Accumulators. You only see the ones offered on your market.
  - Trade type dropdown. It refills when you change the category.
- **How it works:** The trade type decides which contracts exist, so it also decides what the Contract Type list, the Purchase dropdown and the Trade options block show. For example, Digits trades get a Prediction box in Trade options, and Multipliers and Accumulators swap the Trade options block for their own block.
- **Example:** Even/Odd digits.

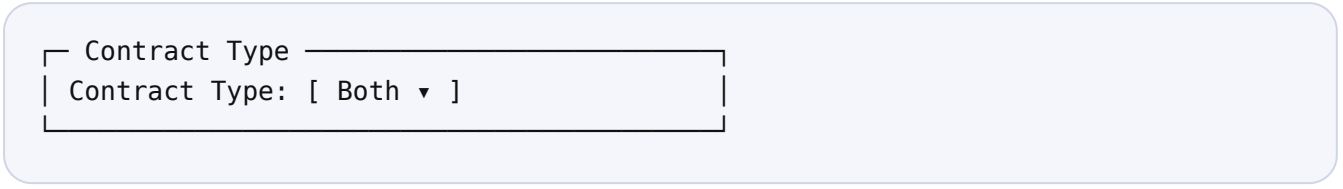


- **Good to know:**
  - After you change the trade type, check the Trade options block again. Its boxes change to match.
  - A trade type that Deriv has no durations for on your market is left out of the list.

Contract Type

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_contracttype`
- **Shape:** Action block (a locked helper block)

- **Where it goes:** Third slot inside 1. Trade parameters. Locked like Market.
- **What it does:** Chooses which side of the trade the bot can buy. A trade type is a pair, such as Even and Odd. You can allow one side only, or "Both".
- **Inputs:**
  - Contract type dropdown. When the trade type has two sides, "Both" comes first, then each side (for example Even, Odd). Multipliers offer Up and Down. Accumulators offer Buy. If nothing is available it shows "Not available".
- **How it works:** If you pick one side, the bot prepares only that contract and the Purchase block can only offer that side. If you pick "Both", the bot prepares both contracts, and the Purchase block can buy either.
- **Example:** Both sides allowed.

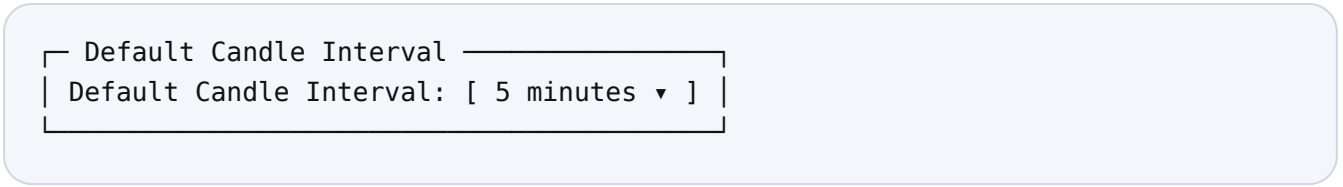


- **Good to know:**
  - Deriv's own hint: if the contract type is "Both", the Purchase conditions should include both sides, using an if block.
  - Changing the trade type resets this list.

Default Candle Interval

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_candleinterval`
- **Shape:** Action block (a locked helper block)
- **Where it goes:** Fourth slot inside 1. Trade parameters. Locked like Market.
- **What it does:** Sets the candle size the bot uses when a candle block is left on "Default". A candle sums up the price over a period of time with four numbers: open, high, low and close.
- **Inputs:**
  - Interval dropdown: 1 minute, 2 minutes, 3 minutes, 5 minutes, 10 minutes, 15 minutes, 30 minutes, 1 hour, 2 hours, 4 hours, 8 hours, 1 day. A fresh builder starts on 1 minute.
- **How it works:** The chosen interval is passed to the trade engine at start. Candle blocks that are set to "Default" then ask for candles of this size. If you pick a size on the candle block itself, that choice wins.

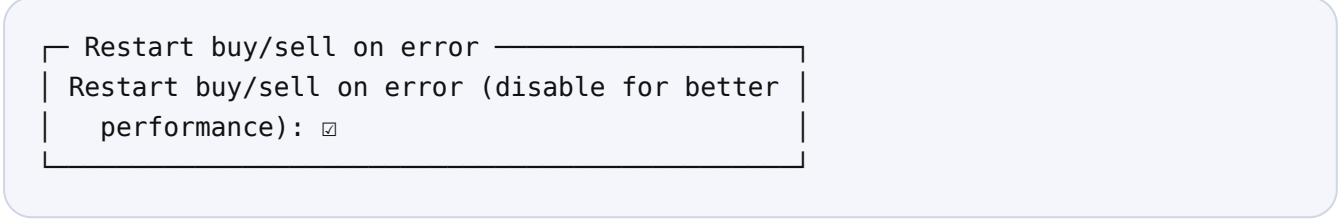
- **Example:**



- **Good to know:**
  - It does not change how long your trade lasts. That is the Duration in Trade options.
  - It only matters if your bot uses candle blocks.

Restart buy/sell on error (disable for better performance)

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_restartbuysell`
- **Shape:** Action block (a locked helper block)
- **Where it goes:** Fifth slot inside 1. Trade parameters. Locked like Market.
- **What it does:** A tick box. When it is on, the bot tries to recover if buying or selling fails because of a price problem, instead of stopping.
- **Inputs:**
  - Check box. It starts unticked (off).
- **How it works:** When it is on, the bot keeps a saved copy of where it was when it reached Purchase conditions and Sell conditions. If the buy fails with "price moved" or "invalid contract proposal" (Deriv's price changed before the order was accepted), or if a sell fails, it waits a few seconds (longer after each try, up to 15 seconds), goes back to that saved point and tries again with fresh prices. When it is off, those problems are not retried: the bot stops. Connection drops, rate limits and closed markets are always waited out and retried.
- **Example:**



- **Good to know:**
  - The label itself warns that the saved copies make the bot slower. Leave it off unless you need it.
  - It is not a guarantee that a trade will be bought.

Restart last trade on error (bot ignores the unsuccessful trade)

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_restartonerror`
- **Shape:** Action block (a locked helper block; it is the last one, with nothing below it)
- **Where it goes:** Sixth and last slot inside 1. Trade parameters. Locked like Market.
- **What it does:** A tick box. When it is on, a normal error does not stop the bot. The bot drops the failed trade and starts the round again.
- **Inputs:**
  - Check box. It starts ticked (on).
- **How it works:** When an error happens and the box is ticked, the bot shows the error in the journal, closes its connections, reconnects, and restarts from the Trade options block using the state saved at the start of the round. The unsuccessful trade is ignored. If the box is not ticked, the error stops the bot.
- **Example:**

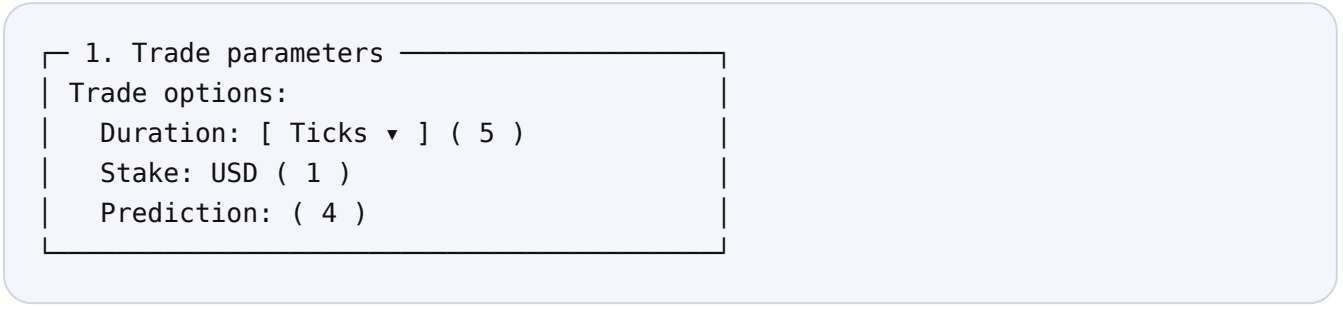
```
┌─ Restart last trade on error ──────────────┐
│ Restart last trade on error (bot ignores the │
│   unsuccessful trade): ☑                    │
└─────────────────────────────────────────────┘
```

- **Good to know:**
  - Some errors always stop the bot, even with the box ticked. Examples: not enough balance, invalid token, a daily loss limit reached, a stake or payout in the wrong format, and "price moved".
  - Restarting after an error does not repair a bad strategy. Read the journal.

Trade options

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_tradeoptions`
- **Shape:** Action block
- **Where it goes:** In the "Trade options:" slot of 1. Trade parameters. One only, with nothing below it. It is used for every trade type except Multipliers and Accumulators.

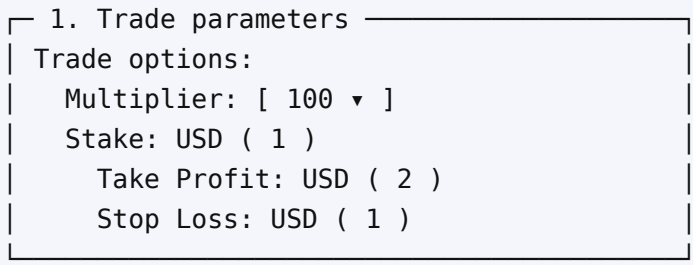
- **What it does:** Sets how long each contract lasts and how much you stake. For some trade types it also shows a Prediction box or Barrier boxes.
- **Inputs:**
  - Duration unit dropdown: the units Deriv allows for your market and trade type (Ticks, Seconds, Minutes, Hours, Days). If Deriv's list does not arrive, the builder falls back to: Ticks 1 to 10, Seconds 15 to 3600, Minutes 1 to 1440, Hours 1 to 24, Days 1 to 365.
  - Duration number (a value socket). A fresh builder has 5.
  - Stake number (a value socket), with the currency of your account written before it. A fresh builder has 1. The allowed range is written next to it as "(min: ... - max: ...)".
  - Prediction (Digits trades only): the digit you pick, in a value socket. The allowed digits come from Deriv.
  - Barrier, or High barrier and Low barrier (trade types that need a price level): a dropdown (Offset +, Offset -, or Absolute where allowed) and a number. An offset is a distance from the entry price, an absolute is a fixed price.
- **How it works:** At the start of each round the bot reads the boxes and sends the order details to the trade engine. The stake is rounded to the decimals of your currency (for example 2 for USD), so a stake like 12.232323 becomes 12.23. The bot buys on a stake basis.
- **Example:** An Over/Under bot with a prediction.



- **Good to know:**
  - The bot refuses to run with a bad stake or duration and tells you why, for example "Please enter a stake amount that's at least ...", "Maximum payout allowed is ...", "Amount must be a positive number." or "Duration value is not allowed. To run the bot, please enter a value between ... to ...".
  - If you choose Multipliers or Accumulators as the trade type, this block swaps itself for the matching block. Going back swaps it again.
  - A number with a leading zero (like 05) is tidied to 5.
  - Older saved bots may hold a hidden twin of this block, shown next as "Trade options (Payout version)".

Multiplier trade options

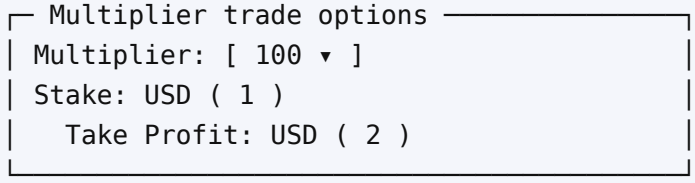
- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_multiplier`
- **Shape:** Action block
- **Where it goes:** In the "Trade options:" slot of 1. Trade parameters, only when the trade type is Multipliers. One only, with nothing below it.
- **What it does:** Sets the multiplier and the stake for a Multipliers trade. A multiplier contract makes your profit or loss move faster than the price, by the number you pick. It has a slot for Take Profit and Stop loss blocks.
- **Inputs:**
  - Multiplier dropdown: the values Deriv offers for your market. If none arrive, the list shows 100, 200, 300, 500 and 1000.
  - Stake number, with the account currency written before it.
  - A slot inside for the Take Profit (Multiplier) and Stop loss (Multiplier) blocks. Each can be used once.
- **How it works:** At the start of each round the bot sends the multiplier, the stake (rounded to the currency's decimals), and the Take Profit and Stop loss limits (rounded to 2 decimals) to the trade engine. A limit whose number is 0 or less is switched off and is not sent.
- **Example:**



- **Good to know:**
  - Choosing another trade type swaps this block for Trade options.
  - Take Profit and Stop loss are limits you accept on one contract. They are not a promise of profit.
  - The same stake checks as in Trade options apply (positive number, minimum stake).

Take Profit (Multiplier)

- **Kind:** Official (Deriv)
- **Type id:** `multiplier_take_profit`
- **Shape:** Action block
- **Where it goes:** Inside Multiplier trade options, in its inner slot. Used once. Outside that block it is switched off automatically.
- **What it does:** Closes the contract automatically when your profit reaches this amount or more. It works with the Multipliers trade type only.
- **Inputs:**
  - Amount number (a value socket) with your account currency written before it. A new block starts at 0, which means "off".
- **How it works:** The amount is rounded to 2 decimals and sent with the order. If the number is 0 or less, the block is greyed out and nothing is sent.
- **Example:**

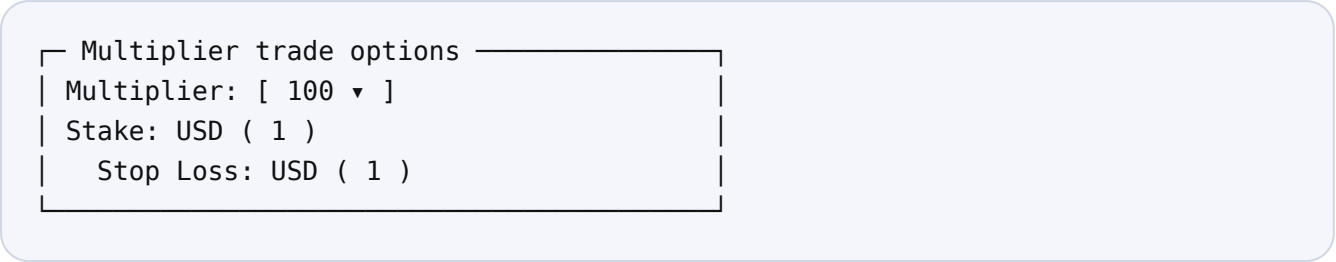


- **Good to know:**
  - The number must be positive, or the journal says "Take profit must be a positive number."
  - Take Profit only closes a contract that is winning. It is not a safety net against losses.

Stop loss (Multiplier)

- **Kind:** Official (Deriv)
- **Type id:** `multiplier_stop_loss`
- **Shape:** Action block
- **Where it goes:** Inside Multiplier trade options, in its inner slot. Used once. Outside that block it is switched off automatically.
- **What it does:** Closes the contract automatically when your loss reaches this amount or more. It works with the Multipliers trade type only. On the block it reads "Stop Loss".

- **Inputs:**
  - Amount number (a value socket) with your account currency written before it. A new block starts at 0, which means "off".
- **How it works:** The amount is rounded to 2 decimals and sent with the order. Deriv watches the contract and closes it when the loss reaches the amount. If the number is 0 or less, the block is greyed out and nothing is sent.
- **Example:**



- **Good to know:**
  - The number must be positive, or the journal says "Stop loss must be a positive number."
  - It limits the loss you accept on one contract. It cannot stop other losses, such as many contracts in a row.
  - "Not confirmed in the code": whether Deriv can close a contract at a slightly worse price than the amount you set.

Accumulator trade options

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_accumulator`
- **Shape:** Action block
- **Where it goes:** In the "Trade options:" slot of 1. Trade parameters, only when the trade type is Accumulators. One only, with nothing below it.
- **What it does:** Sets the growth rate and the stake for an Accumulators trade. An accumulator contract grows by the growth rate you pick, and Deriv decides how it ends. It has a slot for one Take Profit block.
- **Inputs:**
  - Growth Rate dropdown: the rates Deriv offers, shown as percentages (for example 1%).
  - Stake number, with the account currency written before it.
  - A slot inside for the Take Profit (Accumulator) block. It can be used once.

- **How it works:** At the start of each round the bot sends the growth rate, the stake (rounded to the currency's decimals) and the Take Profit limit (rounded to 2 decimals) to the trade engine.
- **Example:**

```
┌─ 1. Trade parameters ──────────────────────┐
│ Trade options:                             │
│   Growth Rate: [ 1% ▾ ]                    │
│   Stake: USD ( 1 )                         │
│     Take Profit: USD ( 5 )                 │
└────────────────────────────────────────────┘
```

- **Good to know:**
  - Choosing another trade type swaps this block for Trade options.
  - Only the Take Profit (Accumulator) block can sit in the inner slot.
  - A Take Profit of 0 means off.

Take Profit (Accumulator)

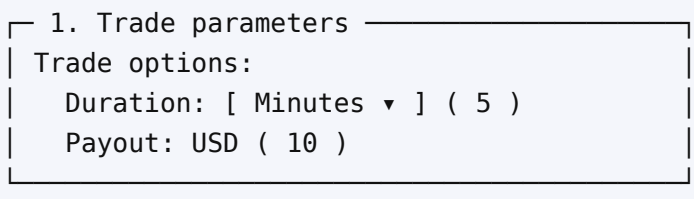
- **Kind:** Official (Deriv)
- **Type id:** `accumulator_take_profit`
- **Shape:** Action block
- **Where it goes:** Inside Accumulator trade options, in its inner slot. Used once. Outside that block it is switched off automatically.
- **What it does:** Closes the contract automatically when your profit reaches this amount or more. It works with the Accumulators trade type only.
- **Inputs:**
  - Amount number (a value socket) with your account currency written before it. A new block starts at 0, which means "off".
- **How it works:** The amount is rounded to 2 decimals and sent with the order. If the number is 0 or less, the block is greyed out and nothing is sent.
- **Example:**

```
┌─ Accumulator trade options ────────────────┐
│ Growth Rate: [ 1% ▾ ]                      │
│ Stake: USD ( 1 )                           │
│   Take Profit: USD ( 5 )                   │
└────────────────────────────────────────────┘
```

- **Good to know:**
  - The number must be positive, or the journal says "Take profit must be a positive number."
  - It closes a winning contract at the profit you chose. It does not limit losses.

Trade options (Payout version)

- **Kind:** Official (Deriv)
- **Type id:** `trade_definition_tradeoptions_payout`
- **Shape:** Action block
- **Where it goes:** In the "Trade options:" slot of 1. Trade parameters, like Trade options. It is not in the menu. It exists so that older saved bots open correctly.
- **What it does:** The same as Trade options, but the amount box is labelled "Payout" and the amount is treated as the payout you want, not the stake.
- **Inputs:**
  - The same boxes as Trade options: duration unit, duration number, the amount, and a Prediction or Barrier box when the trade type needs one.
- **How it works:** It uses the same code as Trade options. The only difference is that the order is sent on a payout basis instead of a stake basis.
- **Example:**



- **Good to know:**
  - You will not find it in the menu. New bots use Trade options.
  - "Not confirmed in the code": whether the builder treats this block as the required Trade options block when it checks your bot before running.

Official blocks: Purchase conditions (before purchase)

2. Purchase conditions is the second root block. It is where you tell the bot when to buy. In the Bot Builder menu these blocks sit under "Purchase conditions" (the Ask Price and Payout blocks sit in the Analysis group, under "Contract"). The bot runs this box after it has read the 1. Trade parameters block, and it runs it again on every new tick (a tick is one price update from the market) until a Purchase block buys a contract.

What can run inside it? The box accepts any block that makes a decision or reads a value: if and else blocks, comparisons, maths, variables, lists, tick and candle readings, indicators, the Ask Price and Payout blocks below, and the extra BRINOVS condition blocks. The one block that actually buys is **Purchase**. If your bot uses a "Tick analysis" block, that block runs on every new tick just before this box does.

A few words used here. A **contract** is one single trade. The **stake** is the money you put into it. The **payout** is the money Deriv pays you if it wins (it includes your stake). A **proposal** is Deriv's live price offer for a contract. A bot needs this box and at least one Purchase block inside it. If either is missing or switched off, the bot will not start.

2. Purchase conditions

- **Kind:** Official (Deriv)
- **Type id:** `before_purchase`
- **Shape:** Root block (a big box with no notch above or below; other blocks go inside it)
- **Where it goes:** On its own on the workspace. It is mandatory, only one copy is allowed, and it cannot be deleted. A fresh Bot Builder already has it, with one Purchase block inside.
- **What it does:** Holds the rules for buying. When a Purchase block inside it runs, the bot buys that contract. If the rules say "not now", nothing is bought and the box runs again on the next tick.
- **Inputs:**
  - One slot for a stack of blocks. Put Purchase blocks here, usually inside if blocks, so the bot only buys when your rule is true.
- **How it works:** The bot waits for a new tick, and also for live prices if your bot reads them (see Payout below). Then it runs the blocks in the box from top to bottom. If a Purchase block runs, the bot buys and moves on to the Sell conditions. If none runs, the bot waits for the next tick and runs the box again. The loop goes on until a purchase is made or you press Stop.

- **Example:** An example rule only: buy Odd when the last digit is 5 or more.

```
┌─ 2. Purchase conditions ──────────────────┐
│ if  < ( Last Digit ) ≥ ( 5 ) >  then      │
│   Purchase [ Odd ▾ ]                      │
└───────────────────────────────────────────┘
```

- **Good to know:**
  - You can use several Purchase blocks in different branches, but only one buy happens per round.
  - If the box has no Purchase block, the bot stops before it starts and the journal says "The Purchase block is mandatory and cannot be deleted/disabled."
  - Stakes, durations and markets are not set here. They live in 1. Trade parameters.

Purchase

- **Kind:** Official (Deriv)
- **Type id:** `purchase`
- **Shape:** Action block (it ends a stack, so it has a notch on top only; nothing can go below it)
- **Where it goes:** Inside 2. Purchase conditions, on its own or under an if block. Outside that box it switches itself off. You can use it more than once, in different branches.
- **What it does:** Buys one contract of the type you pick, with the stake and duration from the Trade options block. This is the block that spends money.
- **Inputs:**
  - Contract dropdown: the contracts allowed by the Trade Type and Contract Type chosen in 1. Trade parameters (for example Rise or Fall, Even or Odd, Over or Under). With "Both" you see both sides; otherwise only the one side.
- **How it works:** The bot sends the buy order to Deriv and waits for the answer. When the order is accepted, it writes the purchase in the journal and moves on to the Sell conditions. A second Purchase in the same round is ignored. If the connection drops or Deriv asks the bot to slow down, the bot waits and tries again. If the price moved, what happens depends on the "Restart buy/sell on error" tick box.
- **Example:** Buy Rise on every round.

```
┌─ 2. Purchase conditions ──────────────────┐
│ Purchase [ Rise ▾ ]                       │
└───────────────────────────────────────────┘
```

- **Good to know:**
  - Use a demo account while you test. Every Purchase spends your stake.
  - If a BRINOVS switch or safety block is in your bot, it is checked before the order is sent and can change or refuse the purchase. For example, with the Hedge switch on, the bot also buys the opposite contract.
  - The dropdown refills when you change the trade type or contract type. Check it after any change.
  - Buying is not a promise of profit. Each contract can lose the whole stake.

Ask Price (Purchase price)

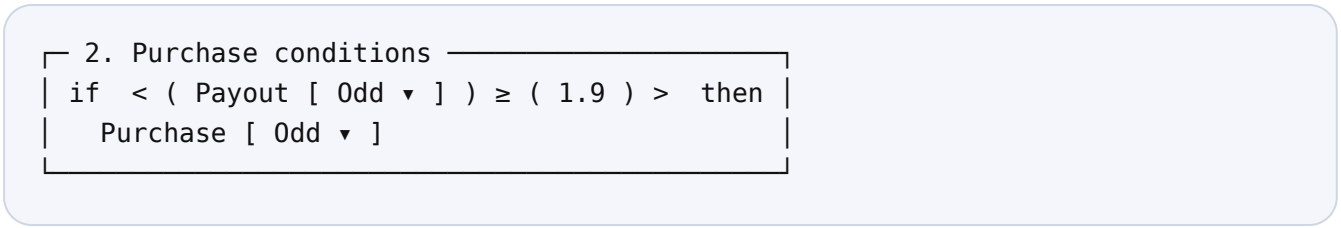
- **Kind:** Official (Deriv)
- **Type id:** `ask_price`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted inside 2. Purchase conditions (for example in a comparison). Deriv says it should be used only in that box. It can be used as often as you like.
- **What it does:** Gives the price Deriv is asking right now for the contract you pick. You can use it to buy only when the price is acceptable.
- **Inputs:**
  - Contract dropdown (the same list as in the Purchase block). On the block it reads "Ask Price" followed by the contract.
- **Gives back:** A number: the current ask price (the cost to buy) for that contract, in your account currency.
- **How it works:** The bot looks up the live proposal for the contract you picked and returns its ask price. The proposal must be for the same trade options as the current round, so a change of stake or duration gives a fresh price.
- **Example:** Buy only if the price is 1 or less.

```
┌─ 2. Purchase conditions ──────────────────────────┐
│ if  < ( Ask Price [ Rise ▾ ] ) ≤ ( 1 ) >  then │
│   Purchase [ Rise ▾ ]                             │
└───────────────────────────────────────────────────┘
```

- **Good to know:**
  - In the code, only the Payout block switches on live prices for the bot. Ask Price on its own may find no price, so use it together with a Payout block.
  - If no live price is ready for the contract, the block cannot give a number and the bot reports an error.
  - It shows the price at the moment the box runs. The price can change on the next tick.

Payout (Potential payout)

- **Kind:** Official (Deriv)
- **Type id:** `payout`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted inside 2. Purchase conditions. Deriv says it should be used only in that box. It can be used as often as you like.
- **What it does:** Gives the payout Deriv offers right now for the contract you pick, which is the money you would get back if the contract wins. You can use it to skip rounds where the offer is too small.
- **Inputs:**
  - Contract dropdown (the same list as in the Purchase block). On the block it reads "Payout" followed by the contract.
- **Gives back:** A number: the potential payout, in your account currency. It includes the stake.
- **How it works:** When this block is in your bot, the bot asks Deriv for live proposals (price offers) for your contracts and keeps them up to date. The block looks up the proposal for the contract you picked and returns its payout. The bot also waits until the first offers have arrived before it runs Purchase conditions.
- **Example:** Buy only when a win would pay at least 1.9 for a stake of 1.



- **Good to know:**
  - This is the block that switches on live prices for the whole bot. It also makes the Purchase block buy at the offered price.
  - If there is no offer for the contract, the block cannot give a number and the bot reports an error.
  - A high payout does not make a win more likely. It only tells you what a win would pay.

Official blocks: Sell conditions (during purchase)

3. Sell conditions is the third root block. It is where you tell the bot when to close a contract early, before it ends by itself. In the Bot Builder menu the root block and Sell at market price sit under "Sell conditions (optional)". The two helper blocks, Sell is available and Sell profit/loss, sit in the Analysis group, under "Contract". This box is optional. If you leave it empty, every contract simply runs until its end time.

The bot runs this box only while a contract is open: after the Purchase block has bought it, and until it ends. It runs on every update of the open contract, which is about once per tick (a tick is one price update from the market). Then, when the contract is over, the bot goes on to the Restart trading conditions box.

A word used here: the **market price** of an open contract is the price Deriv would pay you for it right now if you sold it. It moves with the market, so it can be more or less than your stake. Not every contract can be sold early. Deriv decides, contract by contract and moment by moment, and the Sell is available block tells you the answer.

3. Sell conditions

- **Kind:** Official (Deriv)
- **Type id:** `during_purchase`
- **Shape:** Root block (a big box with no notch above or below; other blocks go inside it)
- **Where it goes:** On its own on the workspace. Only one copy is allowed. It is optional. A fresh Bot Builder has one, with an if block that checks "Sell is available".
- **What it does:** Holds the rules for selling an open contract early. Put an if block in it with your rule, and a Sell at market price block inside the if.
- **Inputs:**
  - One slot for a stack of blocks. In practice you use if blocks, comparisons, the two helper blocks below, variables, and the Sell at market price block.
- **How it works:** While a contract is open, the bot runs this box each time the contract updates. If a Sell at market price block runs, the bot asks Deriv to sell, and when the sale is done the bot moves on to Restart trading conditions. If no sell happens, the contract goes on until it ends by itself. Any "Tick analysis" block you have runs just before this box on every new tick.
- **Example:** Sell early when selling now would give a profit of at least 0.5.

```
┌─ 3. Sell conditions ──────────────────────────────┐
│ if  < Sell is available >  then                   │
│   if  < ( Sell profit/loss ) ≥ ( 0.5 ) >  then    │
│     Sell at market price                          │
└───────────────────────────────────────────────────┘
```

- **Good to know:**
  - The box is optional. Some bots never sell early.
  - It only does something for contracts that can be sold. Deriv decides which ones can.
  - BRINOVS adds more blocks made for this box (see the BRINOVS part of this guide).
  - A sell rule limits a loss only if the contract can still be sold when the rule fires. It does not guarantee a result.

Sell at market price

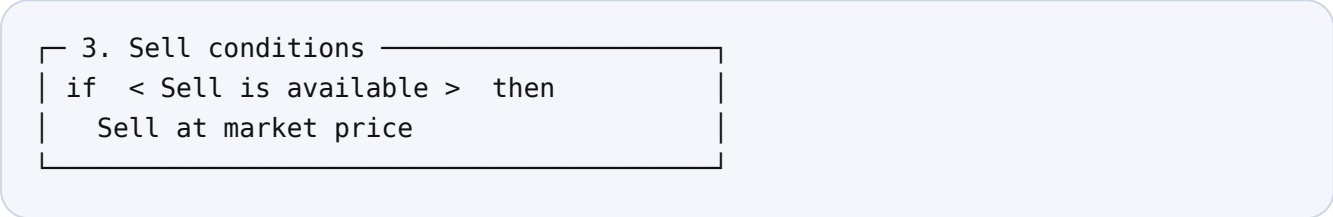
- **Kind:** Official (Deriv)
- **Type id:** `sell_at_market`
- **Shape:** Action block (it has a notch on top and a notch below)
- **Where it goes:** Inside 3. Sell conditions, usually inside an if block. Outside that box it switches itself off. You can use it more than once.
- **What it does:** Sells your open contract right now at the current market price. You get whatever Deriv pays at that moment.
- **Inputs:**
  - None. It has no boxes to fill.
- **How it works:** The bot first checks that a contract is open, not already sold, not expired, and that Deriv allows a sale. If any of these is not true, nothing is sold and the journal shows "Resale of this contract is not offered." If all are true, the bot sends the sell order and writes the sold-for amount in the journal. If Deriv says the contract is too close to its end to sell, the bot does not stop; it lets the contract finish. Selling the same contract twice is not possible.
- **Example:** Sell as soon as a sale is possible.

```
┌─ 3. Sell conditions ──────────────────────┐
│ if  < Sell is available >  then           │
│   Sell at market price                    │
└───────────────────────────────────────────┘
```

- **Good to know:**
  - The price is the market price at that moment. It can be below your stake.
  - Put a Sell is available check in front of it. This keeps the journal free of "not offered" lines.
  - If a sell fails for another reason, what happens depends on the "Restart buy/sell on error" tick box in 1. Trade parameters.

Sell is available (Can contract be sold?)

- **Kind:** Official (Deriv)
- **Type id:** `check_sell`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false answer is wanted inside 3. Sell conditions, most often in an if. Outside that box it switches itself off. It can be used as often as you like.
- **What it does:** Tells you whether your open contract can be sold right now at the market price.
- **Inputs:**
  - None. On the block it reads "Sell is available".
- **Gives back:** True when a sale is possible now. Otherwise it gives nothing (an empty answer), which an if block treats as false.
- **How it works:** The block checks the latest update of the open contract. It is true only when the contract is open, not already sold, not expired, and Deriv marks it as sellable at this moment.
- **Example:** Do something only when selling is possible.

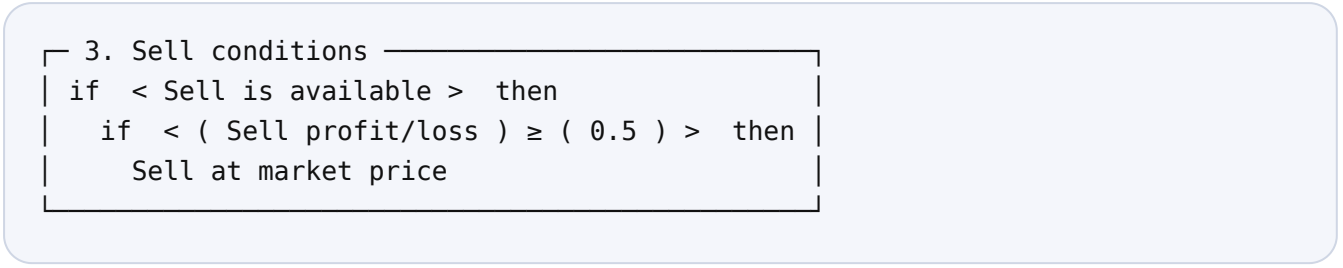


- **Good to know:**
  - The answer can change from one tick to the next, so the bot asks again each time the box runs.
  - It says nothing about whether a sale would be a good idea. For that, look at Sell profit/loss.

Sell profit/loss (Profit/loss from selling)

- **Kind:** Official (Deriv)
- **Type id:** `sell_price`

- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted inside 3. Sell conditions. Outside that box it switches itself off. It can be used as often as you like.
- **What it does:** Gives the profit or loss you would have if you sold the open contract right now.
- **Inputs:**
  - None. On the block it reads "Sell profit/loss".
- **Gives back:** A number, in your account currency. A positive number is a profit. A negative number is a loss.
- **How it works:** The bot takes the price Deriv would pay now (the bid price) and subtracts what you paid for the contract (the buy price). The answer is rounded to the decimals of your currency.
- **Example:** Sell early only when the sale would give at least 0.5 profit.



- **Good to know:**
  - It is a number for this moment only. A moment later it can be different.
  - Check Sell is available first, as the example does.

Official blocks: Restart trading conditions (after purchase)

4. Restart trading conditions is the fourth and last root block. It is where you tell the bot what to do when a contract is finished: look at how it went, change your settings if you want, and decide whether to trade again. In the Bot Builder menu the root block and Trade again sit under "Restart trading conditions". The two helper blocks, Result is and Contract Details, sit in the Analysis group, under "Contract".

The bot runs this box once, right after a contract has ended (it either ran to its end or you sold it early in Sell conditions). Inside, the bot reads your blocks from top to bottom. If it reaches a **Trade again** block, a new round starts: the bot reads the Trade options block again, then goes back to Purchase conditions. If it never reaches Trade again, the bot stops. So a bot without a Trade again block trades only once.

Two words used here. A **contract** is one single trade. The **stake** is the money you put into it. Because the Trade options block is read again at the start of every round, any variable you change in this box (for example your stake) takes effect on the next round.

4. Restart trading conditions

- **Kind:** Official (Deriv)
- **Type id:** `after_purchase`
- **Shape:** Root block (a big box with no notch above or below; other blocks go inside it)
- **Where it goes:** On its own on the workspace. Only one copy is allowed. A fresh Bot Builder has one, with a Trade again block inside.
- **What it does:** Holds the rules that run after each finished contract. You can look at the result, update variables, check your own limits, and then let the bot trade again or stop.
- **Inputs:**
  - One slot for a stack of blocks. Typical content: if blocks with Result is or Contract Details, blocks that set variables, and the Trade again block at the end.
- **How it works:** When a contract has ended, the bot runs the blocks in this box once. If a Trade again block runs, the bot goes straight to the next round. If the end of the box is reached without a Trade again block, the bot stops itself. A Trade again inside an if only runs when that if is true.
- **Example:** Trade again after every contract, with the stake back to 1 after a win.

```
┌─ 4. Restart trading conditions ──────────┐
│ if  < Result is [ Win ▾ ] >  then        │
│   set [ stake ▾ ] to ( 1 )               │
│ Trade again                              │
└──────────────────────────────────────────┘
```

- **Good to know:**
  - This box is the place for your stop rules. If you leave out Trade again when a limit is reached, the bot ends the run.
  - It runs once per finished contract, not on every tick.
  - BRINOVS adds more blocks made for this box (see the BRINOVS part of this guide).

Trade again

- **Kind:** Official (Deriv)
- **Type id:** `trade_again`
- **Shape:** Action block (it ends a stack, so it has a notch on top only; nothing can go below it)
- **Where it goes:** Inside 4. Restart trading conditions, at the end of a stack or at the end of an if. Outside that box it switches itself off. You can use it more than once, in different branches.
- **What it does:** Tells the bot to start another round. Without it, the bot stops after the contract.
- **Inputs:**
  - None. It has no boxes to fill.
- **How it works:** When the bot reaches this block, it marks "trade again" and leaves the Restart trading conditions box at once. The blocks after it, if any, are skipped. Then the next round begins: the bot reads the Trade options block and goes back to Purchase conditions.
- **Example:** Trade again only after a win.

```
┌─ 4. Restart trading conditions ──────────┐
│ if  < Result is [ Win ▾ ] >  then        │
│   Trade again                            │
└──────────────────────────────────────────┘
```

- **Good to know:**
  - In the example above the bot stops after a loss, because Trade again is not reached.
  - Deriv's wording: it transfers control back to the Purchase conditions block.
  - It works at any depth, so you can put it inside nested if blocks.

Result is (Last trade result)

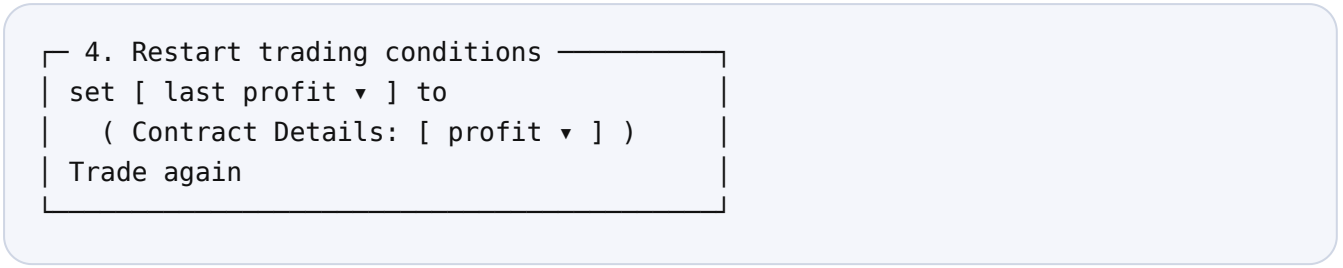
- **Kind:** Official (Deriv)
- **Type id:** `contract_check_result`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false answer is wanted inside 4. Restart trading conditions, most often in an if. If you drop it in another root block (for example Purchase conditions), it switches itself off. It can be used as often as you like.
- **What it does:** Checks how your last contract ended: with a win or with a loss.
- **Inputs:**
  - Result dropdown: Win or Loss. On the block it reads "Result is" followed by the choice.
- **Gives back:** True when the last contract's result matches your choice. Otherwise false.
- **How it works:** The bot looks at the finished contract. It works out the profit (what the contract paid out minus what you paid). A profit below 0 is a "loss". Anything else, including exactly 0, counts as a "win". Then it compares with your choice.
- **Example:** Do something only after a loss.

- **Good to know:**
  - It always talks about the last finished contract.
  - A contract sold early counts by its profit too.
  - If you use it before any contract has finished, there is nothing to check yet.

Contract Details

- **Kind:** Official (Deriv)
- **Type id:** `read_details`
- **Shape:** Value block (gives a number, text or other value, depending on your choice)
- **Where it goes:** Anywhere a value is wanted inside 4. Restart trading conditions. Outside that box it switches itself off. It can be used as often as you like.
- **What it does:** Reads one fact about your last finished contract, such as the profit or the exit price.

- **Inputs:**
  - Detail dropdown: deal reference id, purchase price, payout, profit, contract type, entry spot time, entry spot price, exit spot time, exit spot price, barrier, result. On the block it reads "Contract Details:" followed by the choice. A spot is the market price at a moment.
- **Gives back:** Numbers for reference id, purchase price, payout, profit, entry and exit spot price, and barrier (0 if the contract had no barrier). Text for contract type, result ("win" or "loss"), and the two spot times (shown as hours:minutes:seconds). "payout" is the amount paid back to you when the contract ended.
- **How it works:** The bot takes the finished contract's record from Deriv and picks the item you chose. The profit is the payout minus the purchase price, rounded to your currency's decimals.
- **Example:** Keep the profit of the last contract in a variable.



- **Good to know:**
  - It always talks about the last finished contract.
  - Use it for records and rules. It does not change anything by itself.

Official blocks: Tick and candle analysis

This group of blocks lets your bot look at the market itself, before and between trades. A **tick** is one price update of a market. A **candle** is a summary of the price over a fixed slice of time, such as one minute. Each candle holds four prices: **Open** (the first price), **High** (the highest), **Low** (the lowest) and **Close** (the last), plus the **Open Time** (the moment the candle began, given as a count of seconds).

In the Bot Builder you find these blocks under **Analysis**, in the sub-menu called **Tick and candle analysis**. Most of them are value blocks: they give you a number, a list of numbers or a candle, and you plug them into other blocks (for example into an `if` test, or into the Input List of an indicator). One block, **Run on every tick**, is a section of its own that holds your action blocks.

All of these blocks use the market and the candle size you chose in **Trade parameters**. The bot reads up to the latest 1000 ticks (or 1000 candles) from Deriv, oldest first and newest last. Where a block says "Default" for the candle interval, it means the candle size chosen in Trade parameters (60 seconds, one minute, if none is chosen).

These blocks only **describe the past**. They do not know what the market will do next, and nothing here protects your stake.

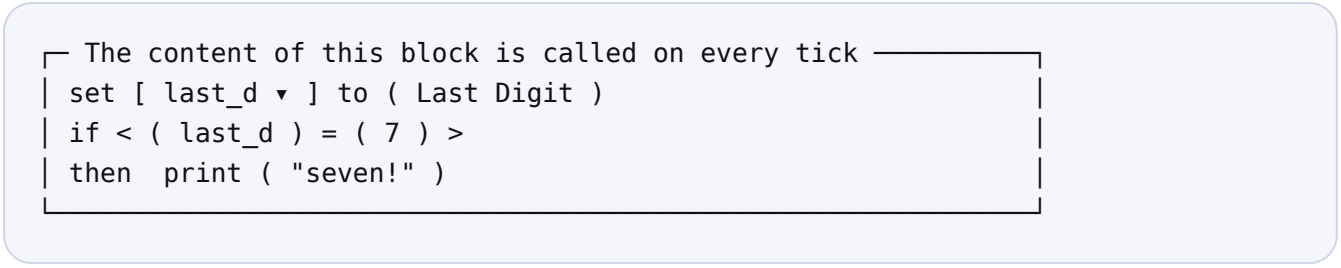
In a running bot, **Run on every tick** runs at the start of each round of the bot's loop (before the purchase conditions are checked, while a contract is open, and after it ends). The value blocks run at the moment the block they sit in runs.

A tip for lists: the "Make a list of..." blocks and the "Candles List" block give you lists. The indicator blocks (see the Indicators section of this guide) take such a list as their Input List.

Run on every tick

- **Kind:** Official (Deriv)
- **Type id:** `tick_analysis`
- **Shape:** Action block (a section that holds action blocks)
- **Where it goes:** On the empty workspace, outside every other root block (Deriv says "Place this block outside of any root block"). You can use more than one.
- **What it does:** Runs the blocks you put inside it once for every new tick, whatever the trade is doing. Use it to watch the market, keep a counter, or update an indicator, even when no contract is open.

- **Inputs:**
  - The label reads "The content of this block is called on every tick". Below it is an empty space: drop your action blocks there. There are no fields to type.
- **How it works:** The bot keeps a list of all the Run on every tick sections. In its main loop (while it waits to buy, while a contract runs, and after a trade) it checks the time of the newest tick. If that tick is new, it runs every section in the list once. If the tick is the same as last time, it does nothing, so your blocks never run twice for the same tick.
- **Example:**



On each new tick, the bot saves the last digit in the variable `last_d` and prints a note when it is 7.

- **Good to know:**
  - It does not buy or sell by itself. It only runs the blocks you place inside it.
  - If the market is closed, the bot waits and tries again every few seconds.
  - Everything inside runs on every new tick, so keep it light.

Last Tick

- **Kind:** Official (Deriv)
- **Type id:** `tick`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. Needs a market chosen in Trade parameters.
- **What it does:** Gives the price of the newest tick of your market, as a number.
- **Inputs:**
  - No fields. It is just the label "Last Tick".
- **Gives back:** A number, for example 1234.56. Its help name is "Last tick".
- **How it works:** The bot asks for the recent ticks of the chosen market and takes the newest one. If this block sits directly inside a Notify or Print block, it gives the price as text with the right number of decimals (so 1234.50 does not lose its last zero).
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if < ( Last Tick ) > ( 1234.5 ) >                         │
│ then  Purchase [ Rise ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys Rise when the newest price is above 1234.5.

- **Good to know:**
  - It is the latest price only. It says nothing about the next price.
  - If the market is closed, the block gives the text `MarketIsClosed` instead of a number.

Last Tick String

- **Kind:** Official (Deriv)
- **Type id:** `tick_string`
- **Shape:** Value block (gives text)
- **Where it goes:** Anywhere text is wanted. It is not in the Bot Builder menu; it only appears if a saved bot file contains it.
- **What it does:** Gives the price of the newest tick as text, written with the market's usual number of decimals.
- **Inputs:**
  - No fields. It is just the label "Last Tick String".
- **Gives back:** Text, for example "1234.50". Its help name is "Tick value".
- **How it works:** It does the same as Last Tick, but always turns the price into text with the right number of decimals. This keeps trailing zeros.
- **Example:**

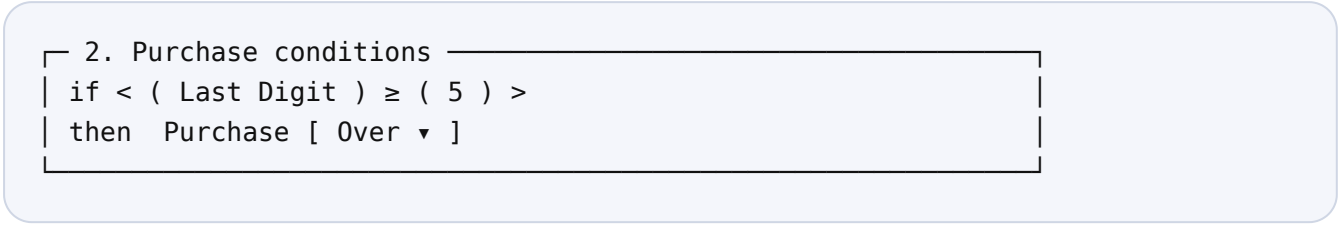
```
┌─ Run on every tick ───────────────────────────────────────┐
│ print ( Last Tick String )                                │
└───────────────────────────────────────────────────────────┘
```

The bot prints the newest price with all its decimals.

- **Good to know:**
  - Use Last Tick when you need to do maths or compare sizes.
  - If the market is closed, the block gives the text `MarketIsClosed`.

Last Digit

- **Kind:** Official (Deriv)
- **Type id:** `last_digit`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. Needs a market chosen in Trade parameters.
- **What it does:** Gives the last digit (0 to 9) of the newest tick price. This is the digit that Digits contracts (Matches, Differs, Even, Odd, Over, Under) are based on.
- **Inputs:**
  - No fields. It is just the label "Last Digit".
- **Gives back:** A whole number from 0 to 9. Its help name is "Last Digit".
- **How it works:** The bot takes the newest price as text, with the market's own number of decimals, and reads its final character. So the price 1234.50 gives 0, not 5.
- **Example:**



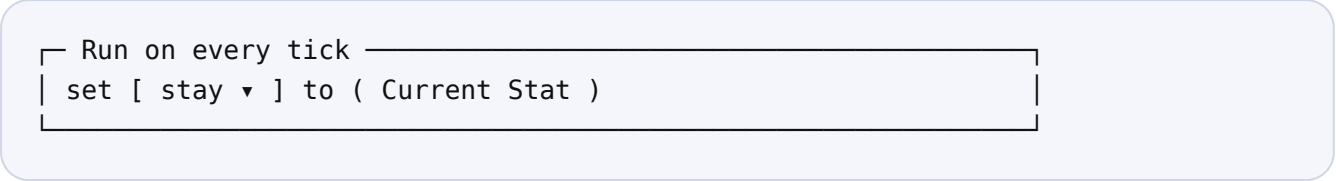
The bot buys when the newest digit is 5 or more.

- **Good to know:**
  - Each digit is random from tick to tick. A digit that just appeared does not make the next digit more or less likely.
  - Zeros at the end count, because the price keeps its fixed decimals.

Current Stat

- **Kind:** Official (Deriv)
- **Type id:** `stat`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. Meant for bots that trade Accumulators.
- **What it does:** Gives the newest number from the "ticks stayed in" statistics that Deriv sends for Accumulators. In plain words: how many ticks in a row the price stayed inside the Accumulators range, in the latest count.

- **Inputs:**
  - No fields. It is just the label "Current Stat".
- **Gives back:** A number (the first entry of the stat list). It can be empty if Deriv does not answer.
- **How it works:** The bot sends Deriv a price request for an Accumulators contract, using the stake, currency, growth rate and market from your Trade parameters. It waits for the answer, reads the "ticks stayed in" list, and gives back its first entry. Then it stops that request so a new call gets fresh numbers.
- **Example:**



On every tick the variable `stay` holds the latest Accumulators stat.

- **Good to know:**
  - Needs a live connection to Deriv.
  - Without Accumulators set up in Trade parameters, the result is not reliable. Not confirmed in the code what you get then.
  - It makes a new request each time, so using it on every tick adds extra traffic.

Current stat list

- **Kind:** Official (Deriv)
- **Type id:** `stat_list`
- **Shape:** Value block (gives a list)
- **Where it goes:** Anywhere a list is wanted. Meant for bots that trade Accumulators.
- **What it does:** Gives a list of the "ticks stayed in" numbers that Deriv sends for Accumulators. Each number says how long the price stayed in range before it broke out.
- **Inputs:**
  - No fields. It is just the label "Current stat list".
- **Gives back:** A list of numbers, starting with the newest, with at most 100 entries (the code keeps the latest 100, although the help text says 1000).
- **How it works:** It works like Current Stat: it asks Deriv for an Accumulators price offer, waits for the answer, then cuts the "ticks stayed in" list to the first 100 entries. It then stops the request.

- **Example:**

```
┌─ Run on every tick ──────────────────────────────────────┐
│ set [ stats ▾ ] to ( Current stat list )                 │
└──────────────────────────────────────────────────────────┘
```

The variable `stats` now holds up to 100 recent stats.

- **Good to know:**
  - Needs a live connection to Deriv.
  - Use it to see how long runs usually last. It does not tell you when the next break-out will happen.

Ticks list

- **Kind:** Official (Deriv)
- **Type id:** `ticks`
- **Shape:** Value block (gives a list)
- **Where it goes:** Anywhere a list is wanted, for example in the Input List of an indicator. Needs a market chosen in Trade parameters.
- **What it does:** Gives a list with the last 1000 tick prices of your market, oldest first and newest last.
- **Inputs:**
  - No fields. It is just the label "Ticks list".
- **Gives back:** A list of numbers. Its help name is "Tick list".
- **How it works:** The bot asks for the recent ticks of the chosen market and keeps only the price of each one. If this block sits directly inside a Notify or Print block, the prices come out as text with the right decimals.
- **Example:**

```
┌─ Run on every tick ──────────────────────────────────────┐
│ set [ sma ▾ ] to Simple Moving Average                   │
│   Input List ( Ticks list )                              │
│   Period ( 10 )                                          │
└──────────────────────────────────────────────────────────┘
```

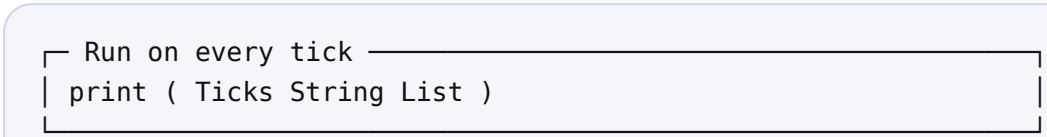
The indicator averages the latest 10 tick prices.

- **Good to know:**

- If the market has fewer than 1000 ticks of history, you get a shorter list.
- The list is only as long as the market history that Deriv sends.

Ticks String List

- **Kind:** Official (Deriv)
- **Type id:** `ticks_string`
- **Shape:** Value block (gives a list)
- **Where it goes:** Anywhere a list is wanted. It is not in the Bot Builder menu; it only appears if a saved bot file contains it.
- **What it does:** Gives the last 1000 tick prices as a list of text items, each written with the market's usual number of decimals.
- **Inputs:**
  - No fields. It is just the label "Ticks String List".
- **Gives back:** A list of text items. Its help name is "Tick List String".
- **How it works:** It works like Ticks list, but every price is turned into text with the right decimals.
- **Example:**



The bot prints all the recent prices as text.

- **Good to know:**
  - Indicators need numbers. Use Ticks list for them, not this block.

Last digits list

- **Kind:** Official (Deriv)
- **Type id:** `lastDigitList`
- **Shape:** Value block (gives a list)
- **Where it goes:** Anywhere a list is wanted. Needs a market chosen in Trade parameters.
- **What it does:** Gives a list of the last digit of each of the last 1000 tick prices, oldest first and newest last. Use it to count how often each digit showed up.

- **Inputs:**
  - No fields. It is just the label "Last digits list".
- **Gives back:** A list of whole numbers from 0 to 9. Its help name is "Last Digits List".
- **How it works:** The bot gets the last 1000 ticks, writes each price with the market's fixed number of decimals, and takes the final digit of each one.
- **Example:**

The variable `digits` holds up to 1000 digits you can count or look through.

- **Good to know:**
  - Past digits do not change the odds of the next digit. Counting them shows what happened, not what will happen.

Direction is

- **Kind:** Official (Deriv)
- **Type id:** `check_direction`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false test is wanted, for example after an `if`. Needs a market chosen in Trade parameters.
- **What it does:** Tests whether the last price move was a rise, a fall, or no change. Its help name is "Market direction".
- **Inputs:**
  - **Direction** dropdown: `Rise`, `Fall` or `No Change`. The first choice in the list is Rise.
- **Gives back:** True if the newest tick price compared with the one before matches your choice, otherwise false.
- **How it works:** The bot looks at the last two ticks. If the newest is higher the direction is "rise", if lower it is "fall", otherwise it is "no change". It then checks that against your dropdown.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if < Direction is [ Rise ▾ ] >                            │
│ then  Purchase [ Rise ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

If the last tick moved up, the bot buys Rise.

- **Good to know:**
  - It looks at only the last two ticks, so it flips often.
  - "No Change" is also the answer when there are fewer than two ticks.
  - A rise just now does not mean another rise is coming.

Is candle black?

- **Kind:** Official (Deriv)
- **Type id:** `is_candle_black`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false test is wanted, except inside Trade parameters. The candle socket must be filled.
- **What it does:** Tests whether a candle is "black". A black candle is one that closed lower than it opened.
- **Inputs:**
  - **Candle** socket: plug in a candle, for example the **Get candle** block. Only candle blocks fit.
- **Gives back:** True if the candle's Close is lower than its Open, otherwise false.
- **How it works:** The bot checks that the given item is a real candle, then compares its Close with its Open. If it is not a candle, the bot stops with the error "Given candle is not valid".
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if < Is candle (in candle list get # from end ( 1 )       │
│      with interval: [ 1 minute ▾ ]) black? >              │
│ then  Purchase [ Fall ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

If the newest 1 minute candle closed below its open, the bot buys Fall.

- **Good to know:**

- A candle that closed at exactly its open price is not black.
- The help text mentions "the last candle", but the code checks the candle you plug in.

In candles list read

- **Kind:** Official (Deriv)
- **Type id:** `read_ohlc`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. Needs a market chosen in Trade parameters. The index socket must be filled.
- **What it does:** Reads one price of one candle in a single step: you say which price (Open, High, Low, Close or Open Time), which candle (counted from the newest), and the candle size. Its help name is "Read candle value (1)".
- **Inputs:**
  - **Candle property** dropdown: `Open`, `High`, `Low`, `Close`, `Open Time`. The menu starts with Open.
  - **# from end** socket: a whole number. 1 is the newest candle, 2 the one before. The menu fills it with 1.
  - **with interval** dropdown: `Default`, `1 minute`, `2 minutes`, `3 minutes`, `5 minutes`, `10 minutes`, `15 minutes`, `30 minutes`, `1 hour`, `2 hours`, `4 hours`, `8 hours`, `1 day`. The menu starts with Default.
- **Gives back:** A number: the chosen price of that candle (for Open Time, the time as a count of seconds).
- **How it works:** The bot gets the candle list for that size, takes the candle at your position from the end, and returns the chosen price. If you give a number that is not a positive whole number, the bot stops with "Index must be a positive integer".
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if < In candles list read [ Close ▾ ] # from end ( 1 )  │
│      with interval: [ 5 minutes ▾ ] > ( 1234.5 ) >      │
│ then  Purchase [ Rise ▾ ]                               │
└─────────────────────────────────────────────────────────┘
```

If the newest 5 minute candle has a Close above 1234.5, the bot buys Rise.

- **Good to know:**
  - "Default" means the candle size set in Trade parameters (60 seconds if none).
  - The newest candle may still be forming, so its Close can change.

Read ... value in candle

- **Kind:** Official (Deriv)
- **Type id:** `read_ohlc_obj`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. The candle socket must be filled.
- **What it does:** Reads one price from a candle that you already have, for example one given by the **Get candle** block. Its help name is "Read candle value (2)".
- **Inputs:**
  - **Candle property** dropdown: `Open`, `High`, `Low`, `Close`, `Open Time`. The menu starts with Open.
  - **Candle** socket: a candle block, for example Get candle.
- **Gives back:** A number: the chosen price of that candle.
- **How it works:** The bot checks that the item is a real candle (with Open, High, Low, Close and Open Time) and returns the field you picked. If it is not a candle, the bot stops with "Given candle is not valid".
- **Example:**

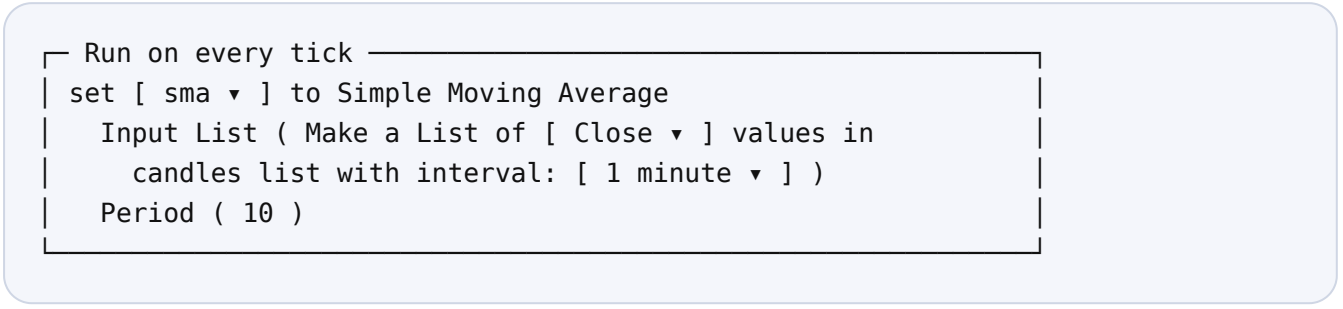
```
┌─ Run on every tick ──────────────────────────────────────┐
│ set [ top ▾ ] to Read [ High ▾ ] value in candle         │
│   ( in candle list get # from end ( 2 )                  │
│      with interval: [ Default ▾ ] )                      │
└──────────────────────────────────────────────────────────┘
```

The variable `top` holds the High of the candle before the newest one.

- **Good to know:**
  - Use it when you want several prices from one candle: get the candle once, store it in a variable, then read from the variable.

Make a List of ... values in candles list with interval:

- **Kind:** Official (Deriv)
- **Type id:** `ohlc_values`
- **Shape:** Value block (gives a list)
- **Where it goes:** Anywhere a list is wanted, for example in the Input List of an indicator. Needs a market chosen in Trade parameters.
- **What it does:** Gives one list that holds the chosen price (Open, High, Low, Close or Open Time) of every candle at the chosen size. Its help name is "Create a list of candle values (1)". The Bot Builder label reads "Make a List of {property} values in candles list with interval: {size}".
- **Inputs:**
  - **Candle property** dropdown: `Open`, `High`, `Low`, `Close`, `Open Time`. The menu starts with Open.
  - **with interval** dropdown: `Default`, `1 minute`, `2 minutes`, `3 minutes`, `5 minutes`, `10 minutes`, `15 minutes`, `30 minutes`, `1 hour`, `2 hours`, `4 hours`, `8 hours`, `1 day`. The menu starts with Default.
- **Gives back:** A list of numbers, oldest candle first.
- **How it works:** The bot gets the candle list for that size (up to 1000 candles) and keeps only the chosen field of each candle.
- **Example:**



The indicator averages the Close of the latest 10 one minute candles.

- **Good to know:**
  - Pick Close for most indicators. Open Time is a time stamp, not a price.

Make a list of ... values from candles list

- **Kind:** Official (Deriv)
- **Type id:** `ohlc_values_in_list`
- **Shape:** Value block (gives a list)

- **Where it goes:** Anywhere a list is wanted. The candle list socket must be filled.
- **What it does:** Takes a candle list you already have (for example from the **Candles List** block) and gives one list with the chosen price of each candle. Its help name is "Create a list of candle values (2)".
- **Inputs:**
  - **Candle property** dropdown: `Open`, `High`, `Low`, `Close`, `Open Time`. If empty, the code uses Open. The menu starts with Open.
  - **Candles list** socket: a list of candles, for example the Candles List block.
- **Gives back:** A list of numbers, in the same order as the candle list.
- **How it works:** The bot checks that the socket holds a list in which every item is a real candle, then picks the chosen field from each one. If not, the bot stops with "Given candle list is not valid".
- **Example:**

```
┌─ Run on every tick ──────────────────────────────────────┐
│ set [ highs ▾ ] to Make a list of [ High ▾ ] values      │
│   from candles list ( Candles List with interval here 2: │
│                       [ 5 minutes ▾ ] )                  │
└──────────────────────────────────────────────────────────┘
```

The variable `highs` holds the High of every 5 minute candle.

- **Good to know:**
  - Handy if you load the candle list once and then want several kinds of price from it.

In candle list get # from end

- **Kind:** Official (Deriv)
- **Type id:** `get_ohlc`
- **Shape:** Value block (gives a candle)
- **Where it goes:** Anywhere a candle is wanted, for example in Is candle black? or Read value in candle. The index socket must be filled.
- **What it does:** Gives you one whole candle (with its Open, High, Low, Close and Open Time) from the list of candles at the chosen size. Its help name is "Get candle".
- **Inputs:**

  - **# from end** socket: a whole number. 1 is the newest candle, 2 the one before. The menu fills it with 1.
  - **with interval** dropdown: `Default`, `1 minute`, `2 minutes`, `3 minutes`, `5 minutes`, `10 minutes`, `15 minutes`, `30 minutes`, `1 hour`, `2 hours`, `4 hours`, `8 hours`, `1 day`. The menu starts with Default.
- **Gives back:** One candle (not a plain number). Only blocks that want a candle accept it.
- **How it works:** The bot gets the candle list for that size and picks the candle at your position from the end. If you give a number that is not a positive whole number, the bot stops with "Index must be a positive integer".
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if < Is candle (in candle list get # from end ( 1 )       │
│      with interval: [ 15 minutes ▾ ]) black? >            │
│ then  Purchase [ Fall ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys Fall when the newest 15 minute candle is black.

- **Good to know:**
  - A candle is not a number. Put it into a candle socket, or read a price from it with Read value in candle.

Candles List with interval here 2:

- **Kind:** Official (Deriv)
- **Type id:** `ohlc`
- **Shape:** Value block (gives a list)
- **Where it goes:** Anywhere a list is wanted. Best used with Make a list of ... values from candles list. Needs a market chosen in Trade parameters.
- **What it does:** Gives the whole list of candles at the chosen size, up to 1000 of them, oldest first. Each item is a full candle. Its help name is "Get candle list".
- **Inputs:**
  - **Candle interval** dropdown: `Default`, `1 minute`, `2 minutes`, `3 minutes`, `5 minutes`, `10 minutes`, `15 minutes`, `30 minutes`, `1 hour`, `2 hours`, `4 hours`, `8 hours`, `1 day`. The menu starts with Default.

- **Gives back:** A list of candles.
- **How it works:** The bot asks Deriv for the candles of the chosen market and size and returns them all. The label text "here 2" is part of the block's name in the Bot Builder.
- **Example:**

```
┌─ Run on every tick ──────────────────────────────────────────┐
│ set [ highs ▾ ] to Make a list of [ High ▾ ] values          │
│   from candles list ( Candles List with interval here 2:     │
│                       [ 5 minutes ▾ ] )                      │
└──────────────────────────────────────────────────────────────┘
```

The variable `highs` holds the High of every 5 minute candle.

- **Good to know:**
  - A list of candles is not a list of numbers. Indicators will not accept it directly; turn it into numbers first.
  - The newest candle may still be forming.

Official blocks: Indicators

What is an indicator?

An **indicator** is a small piece of maths that looks at a list of past prices and boils it down to one number, or to a new list of numbers. The idea is to make a messy list of prices easier to read. For example, an average smooths out the little ups and downs so you can see which way the price has been drifting.

An indicator only **describes the past**. It does not know what the market will do next, and **none of them predicts the future**. A reading that looked "strong" yesterday can be wrong today. Treat every indicator as one more piece of information, never as a promise. Using one does not protect your stake or make a trade safer.

Words you will meet

- **List** (the blocks call it an "Array"): a row of numbers, for example the last 1000 tick prices.
- **Period**: how many of the latest numbers the indicator looks at. A bigger period looks further back, so the result moves more slowly.
- **Variable**: a named box that holds a value. Each indicator block stores its result in a variable you choose (the box next to "set").
- **Pip size**: the number of decimals a market uses. Results of the list versions, of EMA and of Bollinger Bands are rounded to the market's decimals.

How the indicator blocks work

In the Bot Builder you find them under **Analysis**, in the sub-menu **Indicators**. Each indicator is an **action block**: it has a notch on top and below, so it snaps into a stack like any other action block. Put it inside a section that holds action blocks, for example **Purchase conditions** or **Run on every tick**. When the bot runs it, the result goes into the variable, and you use that variable later (for example in an `if` test).

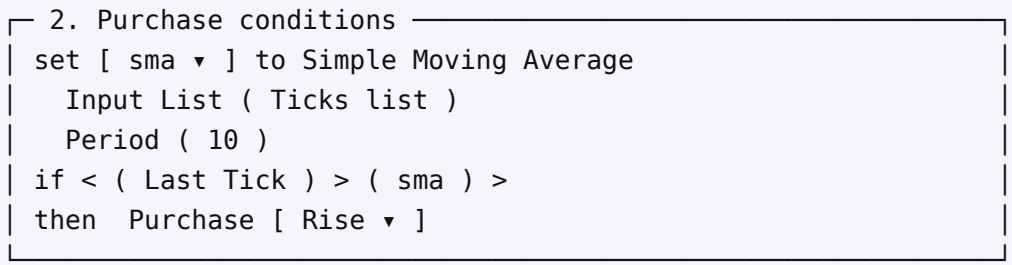
Each indicator comes with small, locked blocks inside it: **Input List** and **Period** (and a few more for Bollinger Bands and MACD). You cannot delete or move them, and you cannot drag in other blocks; you only fill in their sockets. The Input List must be a list of plain numbers: use **Ticks list**, or **Make a**

List of Close values in candles list (see the Tick and candle analysis section). Give the list at least as many numbers as the period, or the bot stops with the error "Periods longer than data length".

Most indicators come in two versions. The plain version gives **one number** (the newest reading). The **Array** version (the name ends in "Array") gives a **list** with a reading for every step along the way, so you can see how the indicator moved. The last item of that list is the newest reading.

set ... to Simple Moving Average

- **Kind:** Official (Deriv)
- **Type id:** `sma_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Works out the plain average of the latest numbers in a list and stores it in a variable. SMA stands for Simple Moving Average. It measures where the price has been "on average" lately, which smooths out small jumps.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `sma`; pick or create any name.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
- **How it works:** The bot takes the last `Period` numbers of the list, adds them up and divides by `Period`. The single result is not rounded. If the list has fewer numbers than the period, the bot stops with "Periods longer than data length".
- **Example:**



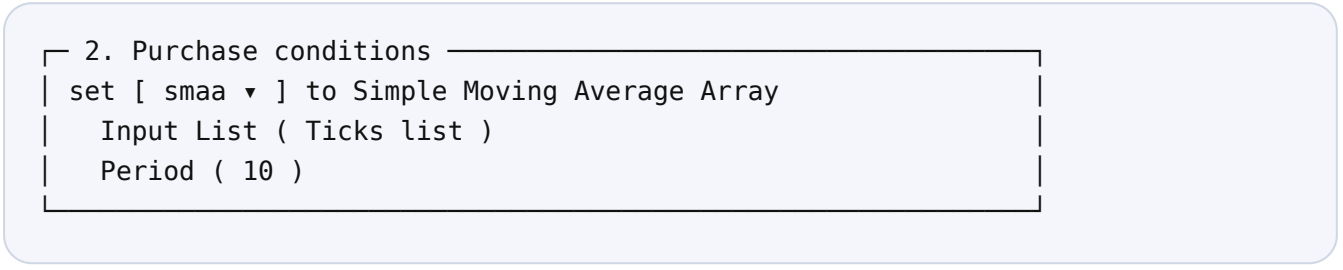
The bot averages the last 10 ticks and buys Rise when the newest price is above that average.

- **Good to know:**

- Deriv describes SMA as a trend helper: if the average keeps moving up, the market has been trending up. That is a description of the past.
- A bigger period gives a smoother but slower line.

set ... to Simple Moving Average Array

- **Kind:** Official (Deriv)
- **Type id:** `smaa_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Gives the whole SMA line as a list instead of one number. Each item is the average of one window of `Period` numbers, sliding along the input list.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `smaa`; pick or create any name.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
- **How it works:** The bot slides a window of `Period` numbers along the input list, one step at a time, and works out the average for each window. Each result is rounded to the market's decimals. The list has (list length - period + 1) items, and the last one is the newest average.
- **Example:**



The variable `smaa` becomes a list of moving averages, oldest first. Use list blocks to pick the item you need.

- **Good to know:**
  - The first item needs a full window, so the list is shorter than the input by (period - 1).
  - Same error as SMA if the input list is shorter than the period.

set ... to Bollinger Bands

- **Kind:** Official (Deriv)
- **Type id:** `bb_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Works out one line of the Bollinger Bands for the latest numbers and stores it as a number. The bands wrap around a moving average: an upper band above it and a lower band below it. They show how widely the price has been swinging.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `bb`; pick or create any name.
  - **Band** dropdown: `upper`, `middle` or `lower`. The menu starts with `middle`.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
  - **Standard Deviation Up Multiplier** (locked inner block): how far above the middle the upper band sits. The menu fills it with 5.
  - **Standard Deviation Down Multiplier** (locked inner block): how far below the middle the lower band sits. The menu fills it with 5.
- **How it works:** The bot takes the last `Period` numbers. The middle line is their average. The **standard deviation** (a measure of how far the numbers usually sit from their average) is worked out. The upper band is the middle plus (standard deviation x up multiplier). The lower band is the middle minus (standard deviation x down multiplier). Your dropdown picks which of the three goes into the variable, rounded to the market's decimals.
- **Example:**

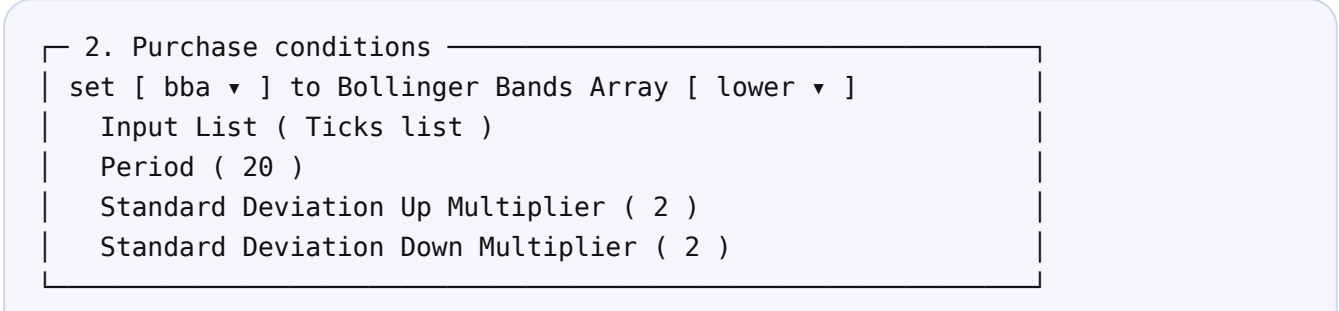
```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ set [ bb ▾ ] to Bollinger Bands [ upper ▾ ]               │
│   Input List ( Ticks list )                               │
│   Period ( 20 )                                          │
│   Standard Deviation Up Multiplier ( 2 )                  │
│   Standard Deviation Down Multiplier ( 2 )                │
└──────────────────────────────────────────────────────────┘
```

The variable `bb` holds the upper band of the last 20 ticks.

- **Good to know:**
  - Deriv says that when the price touches a band, a turn-around is possible. It is only a possibility, not a signal to count on.
  - Bigger multipliers make wider bands. Fill both multipliers in; they are required.

set ... to Bollinger Bands Array

- **Kind:** Official (Deriv)
- **Type id:** `bba_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Gives one whole line of the Bollinger Bands as a list. The line is the lower band, the middle line (a moving average) or the upper band, as you choose.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `bba`; pick or create any name.
  - **Band** dropdown: `upper`, `middle` or `lower`. The menu starts with `middle`.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
  - **Standard Deviation Up Multiplier** (locked inner block): the menu fills it with 5.
  - **Standard Deviation Down Multiplier** (locked inner block): the menu fills it with 5.
- **How it works:** The bot slides a window of `Period` numbers along the list. For each window it works out the middle, upper and lower bands exactly as Bollinger Bands does, and keeps the one you chose. The list has (list length - period + 1) items, rounded to the market's decimals, and the last item is the newest.
- **Example:**



The variable `bba` is a list of lower-band values, one per window.

- **Good to know:**
  - Use it when you want to compare how the band moved over time.
  - Same error as the other indicators if the list is shorter than the period.

set ... to Exponential Moving Average

- **Kind:** Official (Deriv)
- **Type id:** `ema_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Works out the Exponential Moving Average (EMA) of a list and stores it in a variable. It is an average that gives more weight to the newest numbers, so it reacts faster to recent price changes than SMA does.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `ema`; pick or create any name.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
- **How it works:** The bot starts from the average of the first `Period` numbers of the list. Then it goes through the remaining numbers one by one, each time moving the result part of the way toward the new number. The step size is 2 / (Period + 1). The final result is rounded to the market's decimals.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ ema ▾ ] to Exponential Moving Average           │
│   Input List ( Ticks list )                           │
│   Period ( 10 )                                       │
└───────────────────────────────────────────────────────┘
```

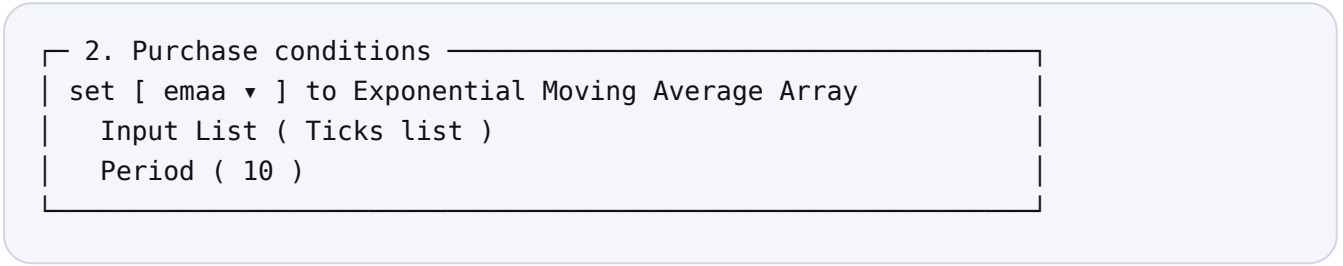
The variable `ema` holds a fast-reacting average of the tick prices.

- **Good to know:**

- Same list and period rules as SMA. If the list is shorter than the period, the bot stops with "Periods longer than data length".
- The same list can give slightly different EMA values if you cut it differently, because the calculation starts at the first number you give it.

set ... to Exponential Moving Average Array

- **Kind:** Official (Deriv)
- **Type id:** `emaa_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Gives the whole EMA line as a list. There is one EMA value for each step from the first full period to the newest number.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `emaa`; pick or create any name.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
- **How it works:** The first item is the average of the first `Period` numbers. Each next item moves part of the way (2 / (Period + 1)) toward the next number. Values are rounded to the market's decimals. The list has (list length - period + 1) items, and the last item is the newest EMA.
- **Example:**



The variable `emaa` is a list of EMA values, oldest first.

- **Good to know:**
  - Handy for comparing two EMA lines with different periods.
  - Needs at least as many numbers as the period.

set ... to Relative Strength Index

- **Kind:** Official (Deriv)
- **Type id:** `rsi_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Works out the Relative Strength Index (RSI) and stores it in a variable. RSI compares how much the price has gone up with how much it has gone down lately, and gives a number from 0 to 100.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `rsi`; pick or create any name.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
- **How it works:** The bot looks at the change from one number to the next. It averages the rises and the falls (with a smoothing step that keeps older changes in the mix), then works out 100 - 100 / (1 + average rise / average fall). If there were no falls the result is 100; if there were no rises it is 0. If the list is exactly as long as the period, the result is 0. The single result is not rounded.
- **Example:**

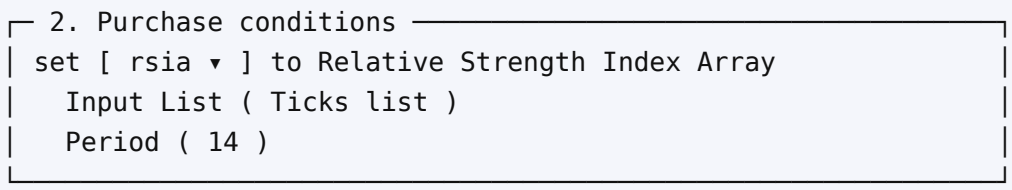
```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ set [ rsi ▾ ] to Relative Strength Index                  │
│   Input List ( Ticks list )                               │
│   Period ( 14 )                                           │
│ if < ( rsi ) ≥ ( 70 ) >                                   │
│ then  Purchase [ Fall ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys Fall when RSI is 70 or more.

- **Good to know:**
  - Deriv's help says 70 and above means "overbought" and 30 and below means "oversold", and that the trend may then reverse. It may not: prices can stay at those levels for a long time.
  - Give it more numbers than the period (at least period + 1) for a real reading.

set ... to Relative Strength Index Array

- **Kind:** Official (Deriv)
- **Type id:** `rsia_statement`
- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Gives the RSI as a list: one RSI reading for each step along the input list. Each value is from 0 to 100.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `rsia`; pick or create any name.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Period** (locked inner block): how many numbers to look at. The menu fills it with 10.
- **How it works:** It works like Relative Strength Index, but repeats the calculation for each point from the first full period onward, keeping the smoothing going from step to step. Each value is rounded to the market's decimals. The list has (list length - period) items, and the last item is the newest RSI.
- **Example:**



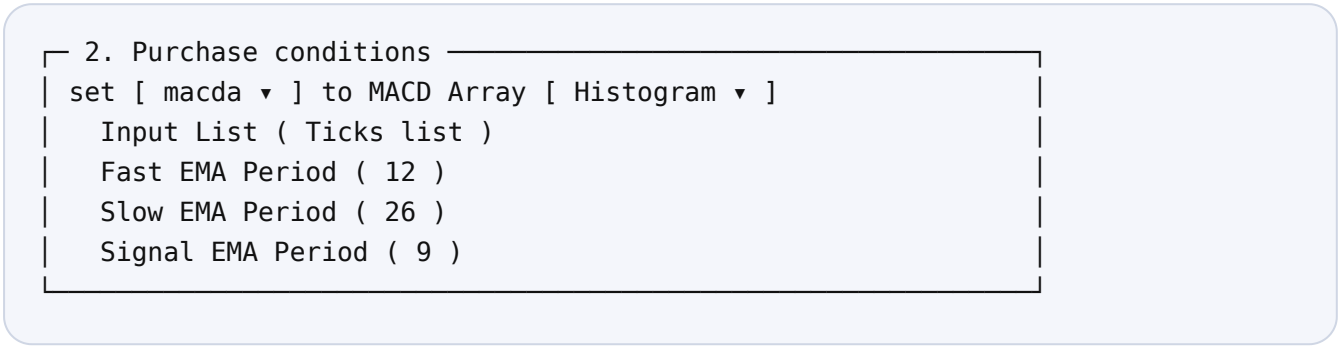
The variable `rsia` is a list of RSI values, oldest first.

- **Good to know:**
  - Use it to see how RSI moved, not just its latest value.
  - It is a description of the past, like every indicator.

set ... to MACD Array

- **Kind:** Official (Deriv)
- **Type id:** `macda_statement`

- **Shape:** Action block (it stores a result in a variable)
- **Where it goes:** Inside any section that holds action blocks, for example Purchase conditions or Run on every tick. You can use it more than once, each with its own variable.
- **What it does:** Works out MACD (Moving Average Convergence Divergence) and gives it as a list. MACD compares a fast EMA with a slow EMA. When the two pull apart or come together, it shows that the price has recently been speeding up or slowing down.
- **Inputs:**
  - **Variable** dropdown (the box after "set"): the variable that receives the result. It starts as `macda`; pick or create any name.
  - **MACD part** dropdown: `Histogram`, `MACD` or `Signal`. The menu starts with `MACD`.
  - **Input List** (locked inner block, see below): a list of numbers, such as Ticks list.
  - **Fast EMA Period** (locked inner block): the short EMA. The menu fills it with 12.
  - **Slow EMA Period** (locked inner block): the long EMA. The menu fills it with 26.
  - **Signal EMA Period** (locked inner block): the smoothing period of the signal line. The menu fills it with 9.
- **How it works:** The bot works out the fast EMA and the slow EMA of the list. The **MACD** line is fast minus slow. The **Signal** line is an EMA of the MACD line. The **Histogram** is MACD minus Signal. Your dropdown picks which of the three becomes the list. The first items, where there is not yet enough data, are left out, so the result is shorter than the input (by Slow + Signal - 2 items). Values are rounded to the market's decimals.
- **Example:**

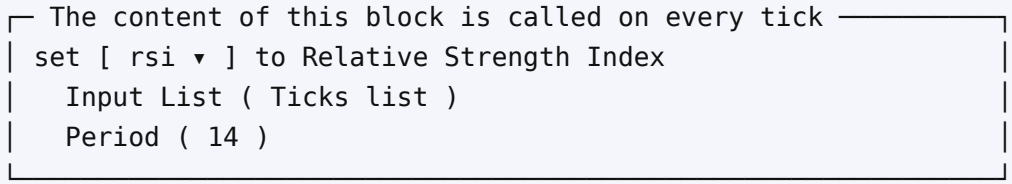


The variable `macda` is a list of histogram values, oldest first.

- **Good to know:**
  - There is no single-number MACD block; it always gives a list. The newest value is the last item.
  - Give it a long list, such as the 1000-tick list. Deriv says a turn is possible when the fast EMA crosses the slow one; that is only a possibility.

Input List

- **Kind:** Official (Deriv)
- **Type id:** `input_list`
- **Shape:** Action block (a locked part of its indicator block)
- **Where it goes:** Only inside any of the nine indicator blocks (SMA, SMAA, BB, BBA, EMA, EMAA, RSI, RSIA, MACD Array). It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Tells the indicator which list of numbers to work on. It is the first inner block of every indicator.
- **Inputs:**
  - **Input List** socket: a block that gives a list. Only list blocks fit (for example Ticks list, or Make a List of Close values in candles list).
- **How it works:** It holds the list for the indicator. It makes no code of its own; the indicator block reads the socket when it runs. If you drag the block out of its indicator, it jumps back in. It is a required socket: leave it empty and the block shows a warning.
- **Example:**



The indicator reads Ticks list, the last 1000 tick prices.

- **Good to know:**
  - The list must hold plain numbers. A list of whole candles will not work; turn it into numbers first.
  - A list of text items from Ticks String List is not suitable for maths.

Period

- **Kind:** Official (Deriv)
- **Type id:** `period`
- **Shape:** Action block (a locked part of its indicator block)

- **Where it goes:** Only inside SMA, SMAA, BB, BBA, EMA, EMAA, RSI and RSIA. It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Says how many of the latest numbers the indicator should look at. A larger period means a smoother result that reacts more slowly.
- **Inputs:**
  - **Period** socket: a number block. The menu fills it with 10. Any number block can go there, including a variable.
- **How it works:** The indicator block reads this socket when it runs. The socket is required. MACD has its own three period blocks instead of this one.
- **Example:**

```
┌─ The content of this block is called on every tick ──────────┐
│ set [ ema ▾ ] to Exponential Moving Average                  │
│   Input List ( Ticks list )                                  │
│   Period ( 10 )                                              │
└──────────────────────────────────────────────────────────────┘
```

The EMA looks at a period of 10.

- **Good to know:**
  - Use a whole number of at least 1. The code does not check this for you.
  - The input list must have at least this many numbers.

Standard Deviation Up Multiplier

- **Kind:** Official (Deriv)
- **Type id:** `std_dev_multiplier_up`
- **Shape:** Action block (a locked part of its indicator block)
- **Where it goes:** Only inside Bollinger Bands and Bollinger Bands Array. It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Sets how far above the middle line the upper Bollinger band sits, counted in standard deviations (a measure of how widely the numbers swing around their average).
- **Inputs:**
  - **Multiplier** socket: a number block. The menu fills it with 5.
- **How it works:** The upper band is the middle line plus (standard deviation x this number). A bigger number moves the upper band further away. The socket is required.

- **Example:**

```
┌─ The content of this block is called on every tick ──────────┐
│ set [ bb ▾ ] to Bollinger Bands [ upper ▾ ]                  │
│   Input List ( Ticks list )                                  │
│   Period ( 20 )                                              │
│   Standard Deviation Up Multiplier ( 2 )                     │
│   Standard Deviation Down Multiplier ( 2 )                   │
└──────────────────────────────────────────────────────────────┘
```

The upper band sits 2 standard deviations above the average.

- **Good to know:**
  - It only changes the upper band. The lower band uses the Down Multiplier.
  - A wide band will be touched less often; a narrow one more often.

Standard Deviation Down Multiplier

- **Kind:** Official (Deriv)
- **Type id:** `std_dev_multiplier_down`
- **Shape:** Action block (a locked part of its indicator block)
- **Where it goes:** Only inside Bollinger Bands and Bollinger Bands Array. It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Sets how far below the middle line the lower Bollinger band sits, counted in standard deviations.
- **Inputs:**
  - **Multiplier** socket: a number block. The menu fills it with 5.
- **How it works:** The lower band is the middle line minus (standard deviation x this number). The socket is required.
- **Example:**

```
┌─ The content of this block is called on every tick ──────────┐
│ set [ bb ▾ ] to Bollinger Bands [ lower ▾ ]                  │
│   Input List ( Ticks list )                                  │
│   Period ( 20 )                                              │
│   Standard Deviation Up Multiplier ( 2 )                     │
│   Standard Deviation Down Multiplier ( 2 )                   │
└──────────────────────────────────────────────────────────────┘
```

The lower band sits 2 standard deviations below the average.

- **Good to know:**
  - It only changes the lower band. The upper band uses the Up Multiplier.
  - The two multipliers can be different numbers.

Fast EMA Period

- **Kind:** Official (Deriv)
- **Type id:** `fast_ema_period`
- **Shape:** Action block (a locked part of its indicator block)
- **Where it goes:** Only inside MACD Array. It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Sets the period of the fast (short) EMA that MACD uses.
- **Inputs:**
  - **Fast EMA Period** socket: a number block. The menu fills it with 12.
- **How it works:** MACD subtracts the slow EMA from this fast EMA to get the MACD line. The socket is required.
- **Example:**

```
┌─ The content of this block is called on every tick ──────────┐
│ set [ macda ▾ ] to MACD Array [ MACD ▾ ]                     │
│   Input List ( Ticks list )                                  │
│   Fast EMA Period ( 12 )                                     │
│   Slow EMA Period ( 26 )                                     │
│   Signal EMA Period ( 9 )                                    │
└──────────────────────────────────────────────────────────────┘
```

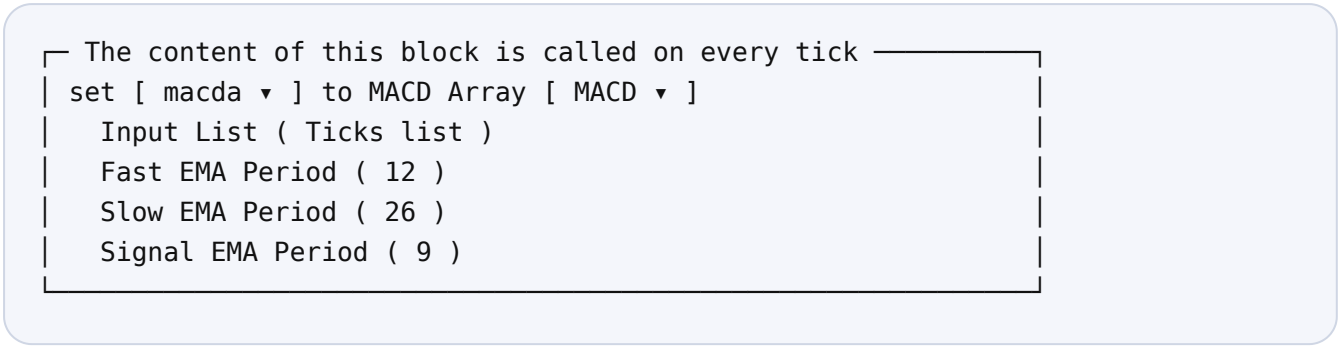
The fast EMA looks at 12 numbers.

- **Good to know:**
  - Normally smaller than the Slow EMA Period. The code does not check this.

Slow EMA Period

- **Kind:** Official (Deriv)
- **Type id:** `slow_ema_period`

- **Shape:** Action block (a locked part of its indicator block)
- **Where it goes:** Only inside MACD Array. It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Sets the period of the slow (long) EMA that MACD uses.
- **Inputs:**
  - **Slow EMA Period** socket: a number block. The menu fills it with 26.
- **How it works:** MACD subtracts this slow EMA from the fast EMA. The input list needs at least this many numbers, and the result list is shorter than the input by (Slow + Signal - 2) items. The socket is required.
- **Example:**



The slow EMA looks at 26 numbers.

- **Good to know:**
  - A longer slow period needs a longer input list.

Signal EMA Period

- **Kind:** Official (Deriv)
- **Type id:** `signal_ema_period`
- **Shape:** Action block (a locked part of its indicator block)
- **Where it goes:** Only inside MACD Array. It is already placed for you when you drag the indicator out, and it cannot be moved out or deleted.
- **What it does:** Sets the period used to smooth the MACD line into the Signal line.
- **Inputs:**
  - **Signal EMA Period** socket: a number block. The menu fills it with 9.
- **How it works:** The Signal line is an EMA of the MACD line with this period. The Histogram is MACD minus Signal, so this setting changes the Signal and Histogram parts. The socket is required.

- **Example:**

```
┌─ The content of this block is called on every tick ──────────┐
│ set [ macda ▾ ] to MACD Array [ Signal ▾ ]                   │
│   Input List ( Ticks list )                                  │
│   Fast EMA Period ( 12 )                                     │
│   Slow EMA Period ( 26 )                                     │
│   Signal EMA Period ( 9 )                                    │
└──────────────────────────────────────────────────────────────┘
```

The Signal line is smoothed over 9 steps.

- **Good to know:**
  - It does not change the MACD part of the dropdown.

Official blocks: Tools (the Utility menu)

The Tools blocks are the helpers of the Bot Builder. They show messages, wait for a while, turn dates into numbers, read candles and tell you your balance and profit. The Notifications, Time and Miscellaneous blocks sit in the **Utility** menu, the candle blocks sit in the **Analysis** menu under Tick and candle analysis, and Account balance, Total profit/loss and Number of runs sit in **Analysis** under Stats. Action blocks go inside a root block such as Purchase conditions and run in the order the bot reaches them, while value blocks plug into sockets. Words used below: a **tick** is one new price from the market, a **candle** is a summary of one slot of time (the open, high, low and close price), the **Journal** is the list of messages in the run panel, and **epoch** time is the count of seconds since 1 January 1970. None of these blocks buys or sells, and none of them protects your money by itself.

Notifications

Notify

- **Kind:** Official (Deriv)
- **Type id:** `notify`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function or a Run on every tick block). You can use it as many times as you like.
- **What it does:** Shows a coloured message in the Journal (the list of messages in the run panel) and can play a sound. Use it to tell yourself what your bot is doing. It does not stop the bot and does not change any trade.
- **Inputs:**
  - Colour dropdown: green, blue, yellow or red. The block from the menu starts on green.
  - Sound dropdown: Silent, Announcement, Earned money, Job done, Error or Severe error. Default: Silent.
  - Message socket: the text or value to show. It must be filled.
- **How it works:** When the bot reaches this block it sends the colour, the message and the sound to the Journal, then carries on with the next block. The message can be a text, a number or a variable.

- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────────┐
│ Notify [ green ▾ ] with sound: [ Earned money ▾ ]            │
│        ( "Round finished" )                                  │
│ Trade again                                                  │
└──────────────────────────────────────────────────────────────┘
```

After every round the bot writes a green "Round finished" line in the Journal, plays the Earned money sound, then trades again.

- **Good to know:**
  - An empty Message socket turns the block red and the bot will not start.
  - The colour is only a label for you. A red message does not mean a loss and a green one does not mean a win.
  - Sounds depend on your device. If your phone is on silent you may hear nothing.

Notify Telegram

- **Kind:** Official (Deriv)
- **Type id:** `notify_telegram`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function or a Run on every tick block). You can use it as many times as you like.
- **What it does:** Sends a message to a Telegram chat, so you get your bot's news on your phone while you are away from the screen. The page must stay open for the bot to keep running.
- **Inputs:**
  - Access Token socket: the secret token of your Telegram bot. It must be filled.
  - Chat ID socket: the number of the chat that should get the message. It must be filled.
  - Message socket: the text to send. It must be filled.
- **How it works:** The bot sends one request over the internet to Telegram's web address, using your token, your chat id and your text. It does not wait for the answer. If Telegram refuses or the request fails, a yellow note "The Telegram notification could not be sent" appears in the Journal and the bot carries on.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ Notify Telegram                                          │
│   Access Token: ( "123456:ABC..." )                      │
│   Chat ID: ( "987654321" )                               │
│   Message: ( "Round finished" )                          │
└──────────────────────────────────────────────────────────┘
```

After a round, the bot sends "Round finished" to your Telegram chat.

- **Good to know:**
  - Treat the token like a password. It is saved as plain text inside your bot file, so do not share or post a bot file that holds a real token.
  - If any of the three sockets is empty, the block is marked red and the bot will not start.
  - A message that is sent is not a promise it arrives. Check your chat the first time.

Time

Seconds Since Epoch

- **Kind:** Official (Deriv)
- **Type id:** `epoch`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket in an if, a Compare, a Set variable and so on). You can use it as many times as you like.
- **What it does:** Gives the current time as one whole number: the seconds that have passed since 1 January 1970 (computers call this the Unix epoch). It is the easy way to measure how long something took.
- **Gives back:** A whole number of seconds, for example 1767225600.
- **How it works:** The block reads the clock of your own device, divides the milliseconds by 1000 and drops the decimals. Run it twice and subtract to get the seconds between two moments.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ( Seconds Since Epoch ) ≥ ( start + 3600 ) >        │
│ then  Notify [ blue ▾ ] with sound: [ Silent ▾ ]          │
│          ( "1 hour" )                                     │
└───────────────────────────────────────────────────────────┘
```

Here "start" is a variable that was set earlier to the same block. When an hour (3600 seconds) has passed, the bot shows a blue note.

- **Good to know:**
  - It uses your device clock, not Deriv's clock. A wrong clock on your phone gives wrong numbers.
  - Seconds only: there are no fractions of a second.

Delayed run

- **Kind:** Official (Deriv)
- **Type id:** `timeout`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function or a Run on every tick block). You can use it as many times as you like.
- **What it does:** Waits a number of seconds and then runs the blocks you put inside it. The rest of your bot waits too.
- **Inputs:**
  - Statement slot (the open space on the block): the blocks to run after the wait.
  - Run after ... second(s) socket: how many seconds to wait. It must be filled.
- **How it works:** The bot pauses for the number of seconds you gave, then runs the blocks inside, then goes on to the next block under this one. Nothing else in your strategy moves during the wait.
- **Example:**

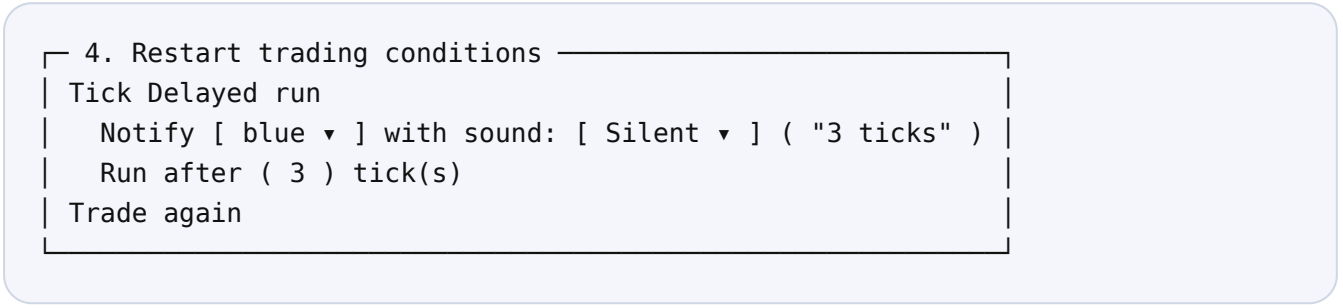
```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ Delayed run                                              │
│   Notify [ blue ▾ ] with sound: [ Silent ▾ ] ( "Back" )  │
│   Run after ( 5 ) second(s)                              │
│ Trade again                                              │
└──────────────────────────────────────────────────────────┘
```

The bot waits 5 seconds, writes "Back" in the Journal, and then trades again.

- **Good to know:**
  - Because the whole bot waits, a long delay means you skip the market in the meantime.
  - To wait for price updates instead of seconds, use Tick Delayed run.

Tick Delayed run

- **Kind:** Official (Deriv)
- **Type id:** `tick_delay`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function or a Run on every tick block). You can use it as many times as you like.
- **What it does:** Waits for a number of ticks (a tick is one new price from the market) and then runs the blocks inside it. The rest of your bot waits too.
- **Inputs:**
  - Statement slot: the blocks to run after the wait.
  - Run after ... tick(s) socket: how many ticks to wait. If it is left out the code uses 1.
- **How it works:** The bot starts watching the prices of the market set in Trade parameters, counts the new ticks as they arrive, and when it has counted your number it runs the blocks inside. Then it continues with the block under this one.
- **Example:**



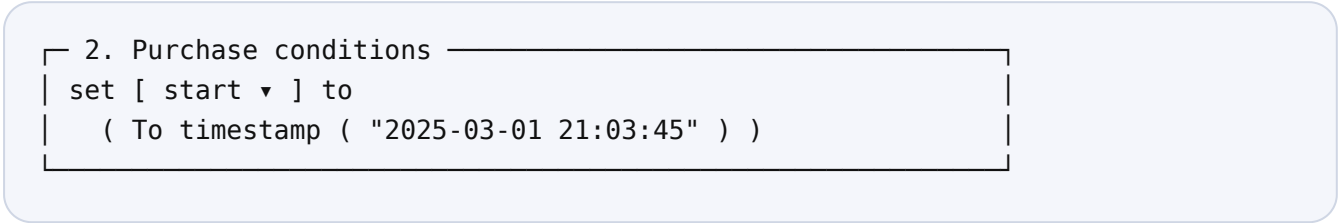
The bot lets 3 new prices go by, writes "3 ticks", and then trades again.

- **Good to know:**
  - A negative number is refused. The block turns red with the message "Values cannot be negative" and the bot will not start.

- A value the block cannot work out (not a number and not a variable) is refused with "Invalid Input".
- How long a tick takes depends on the market, so this is not a fixed number of seconds.

Convert to timestamp

- **Kind:** Official (Deriv)
- **Type id:** `totimestamp`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket in an if, a Compare, a Set variable and so on). You can use it as many times as you like.
- **What it does:** Turns a date and time written as text into one number: the seconds since 1 January 1970 (the Unix epoch).
- **Inputs:**
  - Text socket: a date like 2025-03-01 or a date and time like 2025-03-01 21:03:45. The menu block starts with the text yyyy-mm-dd hh:mm:ss. It must be filled.
- **Gives back:** A number of seconds. If the text cannot be read, it gives the words "Invalid date/time:" followed by your text.
- **How it works:** The block keeps only digits and the signs - . : and spaces. It accepts a year, month and day (YYYY-MM-DD) and, after a space, hours and minutes with optional seconds (hh:mm or hh:mm:ss). Any time-zone words such as GMT+0800 are thrown away, so the time is read as time on your own device.
- **Example:**



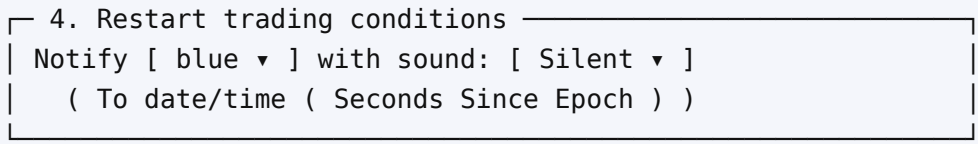
The variable "start" now holds the number for that moment, ready to compare with Seconds Since Epoch.

- **Good to know:**
  - The time zone you type is ignored. Two people in different time zones get different numbers from the same text.
  - Write the date exactly as year-month-day with two digits for month and day.

- Check the result for the words "Invalid date/time" before you do maths with it.

Convert to date/time

- **Kind:** Official (Deriv)
- **Type id:** `todatetime`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket in an if, a Compare, a Set variable and so on). You can use it as many times as you like.
- **What it does:** Turns a number of seconds since 1 January 1970 (the Unix epoch) into a readable date and time.
- **Inputs:**
  - Number socket: the seconds to convert. It must be filled. The menu block starts with 0.
- **Gives back:** Text such as 2019-01-01 13:03:45 GMT+0000. If the number is not a real time it gives "Invalid timestamp:" followed by the number.
- **How it works:** The block builds the text year-month-day hours:minutes:seconds in the time zone of your own device and adds the zone at the end as GMT plus or minus hours and minutes.
- **Example:**



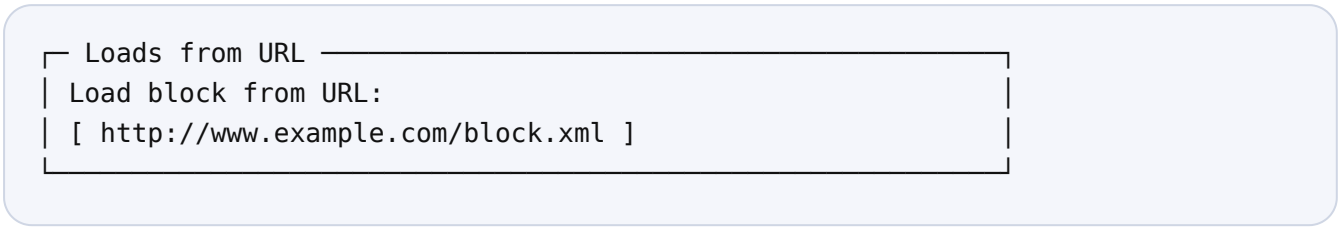
The Journal shows the time of this moment, for example 2025-03-01 21:03:45 GMT+0000.

- **Good to know:**
  - The text uses your device time zone, so the same number looks different on a device in another zone.
  - The socket takes numbers only.
  - The number 0, which is the value the menu block starts with, gives "Invalid timestamp: 0".

Miscellaneous

Loads from URL

- **Kind:** Official (Deriv)
- **Type id:** `loader`
- **Shape:** Stand-alone block (no dents; it does not plug into other blocks)
- **Where it goes:** Straight on the empty canvas, as its own stand-alone block. It does not plug into other blocks. Several are allowed, but each needs its own web address.
- **What it does:** Loads extra blocks from a block file (an .xml file) kept on a web address, and adds them to your canvas.
- **Inputs:**
  - Web address box: the address of the block file. The new block starts with http://www.example.com/block.xml.
- **How it works:** When you place the block, or change its address, or open a saved bot that contains it, the page fetches the file and adds the blocks to the canvas. If you change the address, the blocks it added before are removed first. An address with no http gets http:// put in front. An address that ends in / gets index.xml added. The block itself adds no code to your bot.
- **Example:**



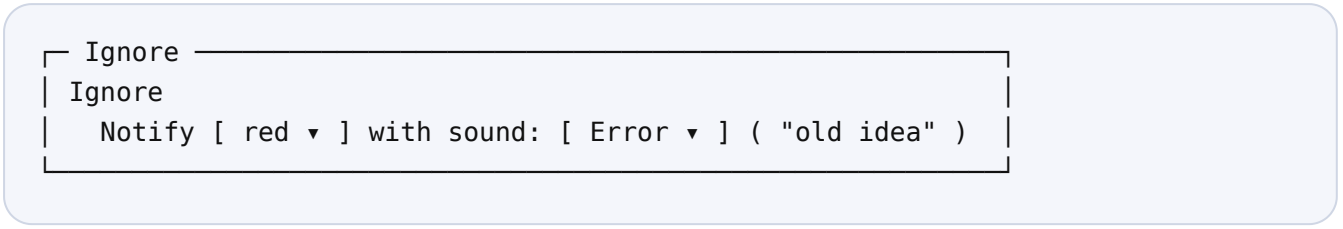
The page fetches that file and the blocks inside it appear on your canvas.

- **Good to know:**
  - Only load files you made or fully trust. Blocks from a stranger's address become part of your bot.
  - The address must end in a file name with a three-letter ending such as .xml, or end with /. Otherwise you get "Target must be an XML file".
  - A second block with an address that is already loaded is switched off ("This URL is already loaded"). A failed load shows "An error occured while trying to load the URL" and switches the block off.

Ignore

- **Kind:** Official (Deriv)
- **Type id:** `block_holder`

- **Shape:** Stand-alone block (it has an open slot but no dents; it does not plug into other blocks)
- **Where it goes:** Straight on the empty canvas, as its own stand-alone block. It does not plug into other blocks.
- **What it does:** A parking space for blocks. Everything you put inside it is kept on the canvas but is never run.
- **Inputs:**
  - Statement slot: drop any blocks here. The title on the block reads Ignore.
- **How it works:** The block creates no code, so the bot skips everything inside it. Before a run the bot switches off loose blocks that sit outside a root block, but this block is allowed to stay on its own, so what is inside is kept safe.
- **Example:**



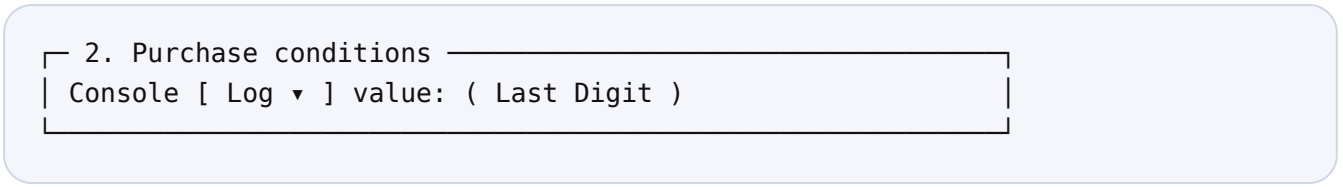
The Notify block is saved with your bot but does nothing when the bot runs.

- **Good to know:**
  - Use it to keep an old idea without deleting it.
  - Moving a block out of Ignore and into a root block makes it live again.

Console

- **Kind:** Official (Deriv)
- **Type id:** `console`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function or a Run on every tick block). You can use it as many times as you like.
- **What it does:** Writes a value to the browser's developer console, a technical window meant for programmers. It does not show in the Journal.
- **Inputs:**
  - Type dropdown: Log, Warn, Error or Table. Default: Log.
  - Value socket: the text, number, true/false or list to write. It must be filled.

- **How it works:** The bot calls the browser's own console with the type you chose and your value. Table is meant for lists and shows them as rows. If the socket is empty the code writes the words <empty message>.
- **Example:**



Each time this runs, the last digit is written to the developer console.

- **Good to know:**
  - On a phone you usually cannot open the developer console. Use Notify if you want to read it.
  - It never changes a trade.

In the Analysis menu

Account balance

- **Kind:** Official (Deriv)
- **Type id:** `balance`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a number or text is wanted. It sits in the Analysis menu under Stats.
- **What it does:** Gives the money in your account, either as a number you can calculate with or as ready-to-show text.
- **Inputs:**
  - Type dropdown: string or number. The block starts on string.
- **Gives back:** With number: your balance as a plain number (0 if no balance is known yet). With string: text with thousands commas and the currency, such as 10,000.00 USD.
- **How it works:** The block reads the last balance the page knows about for the account that is logged in. The dropdown also changes the plug of the block between text and number so it fits the right sockets. On the block it reads Balance: ...
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < Balance: [ number ▾ ] ≥ ( 20 ) >                    │
│ then  Purchase [ Rise ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot only buys while the balance is at least 20.

- **Good to know:**
  - Use number for maths and comparisons, and string for messages.
  - It only reads the balance. It does not stop the bot or protect your money by itself.
  - A balance of 0 can also mean that the balance has not arrived yet.

Total profit/loss

- **Kind:** Official (Deriv)
- **Type id:** `total_profit`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a number is wanted. It sits in the Analysis menu under Stats.
- **What it does:** Gives the total profit or loss of your bot since it started, as one number. Wins add to it and losses take away from it.
- **Gives back:** A number in your account currency. A negative number means the bot is down overall.
- **How it works:** The block reads the running total that the page keeps for the account. The total is reset when you click Clear stats in the Transaction Stats window, or when you refresh the page.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ if  < Total profit/loss ≤ ( -50 ) >                      │
│ then  Notify [ red ▾ ] with sound: [ Error ▾ ]           │
│          ( "Down 50" )                                   │
└──────────────────────────────────────────────────────────┘
```

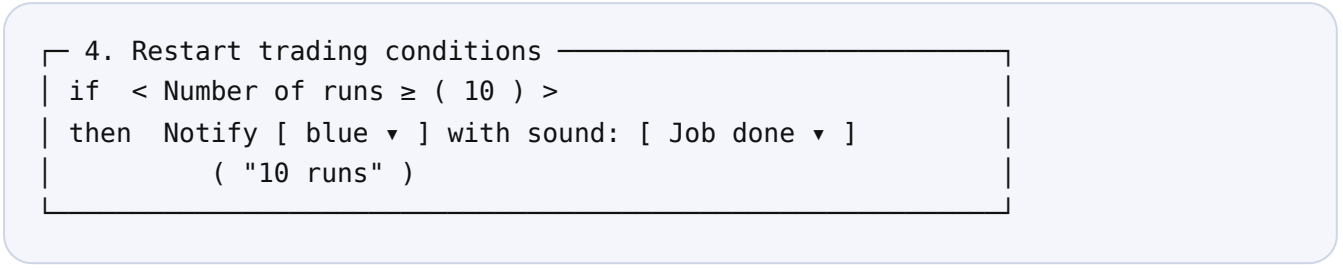
When the bot is down 50 or more, it writes a red note. It does not stop by itself; you would build the stop yourself.

- **Good to know:**
  - It counts the whole run of the bot, not only the last round.

- It reports a number. It does not protect you from losses.

Number of runs

- **Kind:** Official (Deriv)
- **Type id:** `total_runs`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a number is wanted. It sits in the Analysis menu under Stats.
- **What it does:** Gives how many times your bot has run a trade since the stats were last cleared.
- **Gives back:** A whole number, starting at 0.
- **How it works:** The block reads the running count that the page keeps for the account. The count is reset when you click Clear stats in the Transaction Stats window, or when you refresh the page.
- **Example:**



After 10 runs, the bot writes a blue note and plays the Job done sound.

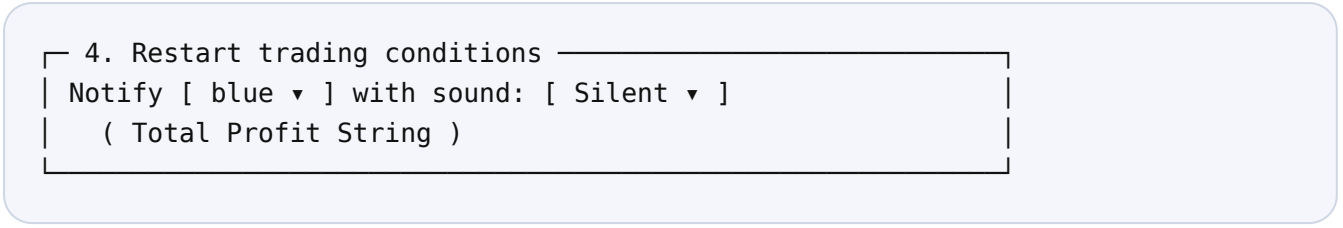
- **Good to know:**
  - Useful to stop or change something after a set number of trades by comparing it with a number.
  - It counts trades, not wins.

Not in the menu

Total Profit String

- **Kind:** Official (Deriv)
- **Type id:** `total_profit_string`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted. It is not in the Bot Builder menu; it only exists for bots that already use it.

- **What it does:** Gives the total profit or loss of your bot, rounded to the number of decimals of your account currency. On the block it reads Total Profit String.
- **Gives back:** The total profit or loss, rounded. The block is typed as text. Not confirmed in the code: whether the value is text or a number.
- **How it works:** It reads the same running total as Total profit/loss and rounds it to the decimals of the account currency. The total is reset by Clear stats or by refreshing the page.
- **Example:**



The Journal shows the total so far, for example 12.5.

- **Good to know:**
  - You will not find it in the menu. For new bots use Total profit/loss.
  - A total of exactly 0 is given as 0.

Barrier offset (old)

- **Kind:** Official (Deriv)
- **Type id:** `barrier_offset`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Nowhere new. It is not in the Bot Builder menu. It stays in the program only so that very old bots still open.
- **What it does:** A leftover from a very old Binary Bot version. A barrier is the price line a contract is measured against, and this block once set it as a plus or minus offset.
- **Inputs:**
  - Dropdown: Offset + or Offset -.
- **Gives back:** Nothing. It creates no code and gives no value.
- **How it works:** The block is kept for backward compatibility. When the bot is built, this block produces nothing.
- **Example:**

The block sits there but has no effect on the bot.

- **Good to know:**
  - Do not use it in new bots.
  - For a barrier offset, use the Offset + / Offset - choice in the Trade options block of Trade parameters.

Conversion Helper Block

- **Kind:** Official (Deriv)
- **Type id:** `useless_block`
- **Shape:** Action block
- **Where it goes:** Nowhere new. It is not in the Bot Builder menu. It appears when the page converts a very old strategy.
- **What it does:** A filler block that the page adds when it converts an old strategy so that the converted blocks fit together.
- **How it works:** The block creates no code. The bot skips it and carries on with the next block.
- **Example:**

The block sits in the stack and has no effect when the bot runs.

- **Good to know:**
  - The tooltip says: "This block was required to correctly convert your old strategy."
  - You can leave it where it is.

Standard blocks: logic, math, text, lists, variables and functions

These are the everyday building blocks of every bot. They test conditions, do maths, handle text and lists, remember values, repeat steps and let you build your own functions. They sit in the **Utility** menu, in the groups Custom functions, Variables, Notifications (Print and Request an input), Math, Text, Logic, Lists and Loops. Action blocks go inside a root block such as Purchase conditions and run from top to bottom, while value and true/false blocks plug into sockets (the open spaces on blocks) and are worked out first, from the inside out. Words used below: a **variable** is a named box that holds a value for later, and a **list** is an ordered row of values. Counting places in lists and text starts at 1, not 0. None of these blocks trade by themselves, protect your money or predict the market.

Logic

Conditional block

- **Kind:** Official (Deriv)
- **Type id:** `controls_if`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Runs the blocks inside only when a test is true. With the plus icon you can add an else part for when the test is false, and more else if tests.
- **Inputs:**
  - if socket: a true/false block. It must be filled.
  - then slot: the blocks to run when the test is true.
  - Plus icon: the first click adds else, the next clicks add else if (each with its own test and slot). The minus icon removes one.
- **How it works:** The bot checks the first test. If it is true it runs the first slot and skips the rest. If not it checks each else if in order, and at the end it runs the else slot if there is one.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ( Last Digit ) ≥ ( 5 ) >                            │
│ then  Purchase [ Rise ▾ ]                                 │
│ else                                                      │
│       Notify [ blue ▾ ] with sound: [ Silent ▾ ]          │
│          ( "wait" )                                       │
└───────────────────────────────────────────────────────────┘
```

If the last digit is 5 or more the bot buys; otherwise it writes "wait" in the Journal.

- **Good to know:**
  - Only a true/false block fits in the if socket. Build the test with Compare, Logic operation and the like.
  - Without an else part, a false test simply does nothing and the bot moves on.

Compare

- **Kind:** Official (Deriv)
- **Type id:** `logic_compare`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** Compares two values and tells you if the comparison is true.
- **Inputs:**
  - Left socket: the first value. The block marks itself red and the bot will not start if this is empty.
  - Operator dropdown: =, ≠, <, ≤, >, ≥. The first choice, =, is the default.
  - Right socket: the second value. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** True or false.
- **How it works:** The bot works out both sides and tests them with the chosen sign. The = test is relaxed: the number 5 and the text "5" count as equal.
- **Example:**

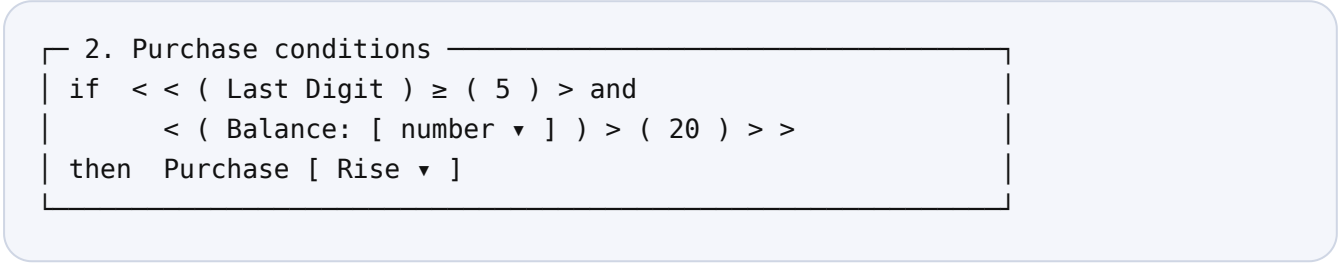
```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ( Last Digit ) = ( 7 ) >                            │
│ then  Purchase [ Rise ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys only when the last digit is exactly 7.

- **Good to know:**
  - ≠ means "is not equal to". ≤ means "less than or equal to".
  - Compare numbers with numbers. Comparing a word with a number is allowed but rarely what you want.

Logic operation

- **Kind:** Official (Deriv)
- **Type id:** `logic_operation`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** Joins two true/false tests into one. With and, both must be true. With or, one is enough.
- **Inputs:**
  - Left socket: a true/false block. The block marks itself red and the bot will not start if this is empty.
  - Dropdown: and, or. Default: and.
  - Right socket: a true/false block. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** True or false.
- **How it works:** The bot works out both sides and combines them. and gives true only if both are true. or gives true if at least one is true.
- **Example:**



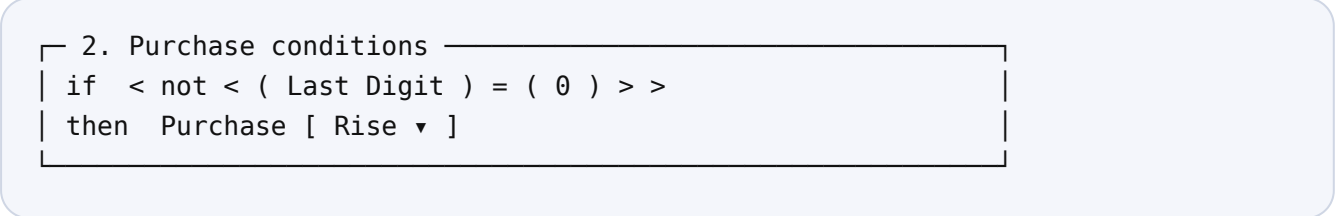
The bot buys only if the last digit is 5 or more AND the balance is above 20.

- **Good to know:**
  - For three tests, put one Logic operation inside another.

- Use it to say several things must all be true before you buy.

Logic negation

- **Kind:** Official (Deriv)
- **Type id:** `logic_negate`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** Turns true into false and false into true.
- **Inputs:**
  - Socket (after the word not): a true/false block. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** The opposite of what is plugged in.
- **How it works:** The bot works out the block inside and flips the answer.
- **Example:**



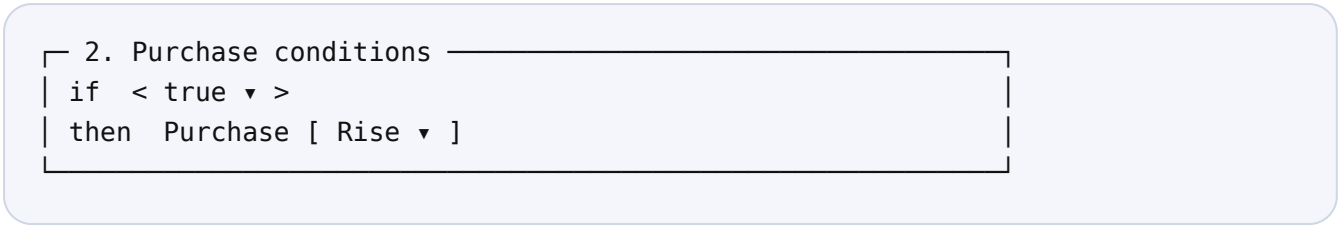
The bot buys when the last digit is anything except 0.

- **Good to know:**
  - Often a Compare with ≠ says the same thing in a shorter way.

True-False

- **Kind:** Official (Deriv)
- **Type id:** `logic_boolean`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** A fixed answer: always true or always false. Handy as a switch while you test.

- **Inputs:**
  - Dropdown: true, false. Default: true.
- **Gives back:** True or false, as chosen.
- **How it works:** The block just hands its chosen answer to the block it is plugged into.
- **Example:**

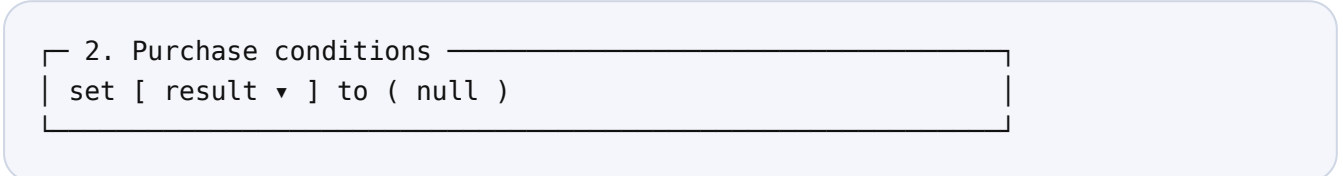


The test is always true, so the bot always buys. Switch it to false to turn the step off.

- **Good to know:**
  - Changing it to false is a quick way to switch a step off without deleting blocks.

Null

- **Kind:** Official (Deriv)
- **Type id:** `logic_null`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted, for example in a Set variable.
- **What it does:** Means "nothing": an empty value. Use it to say a variable has no value yet.
- **Gives back:** The empty value null.
- **How it works:** The block hands the word null to the block it is plugged into.
- **Example:**

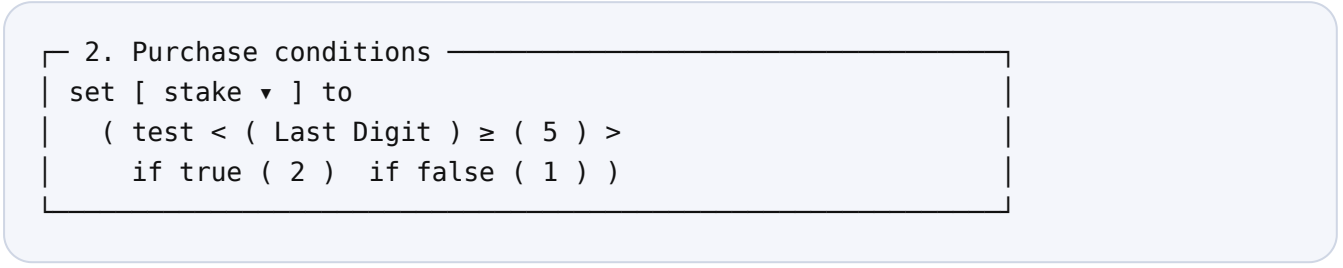


The variable "result" is set to nothing.

- **Good to know:**
  - Null is not the same as 0 or an empty text. It means that there is nothing at all.

Test value

- **Kind:** Official (Deriv)
- **Type id:** `logic_ternary`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Picks one of two values, depending on a test. It is a one-line if.
- **Inputs:**
  - test socket: a true/false block. The block marks itself red and the bot will not start if this is empty.
  - if true socket: the value to give when the test is true. The block marks itself red and the bot will not start if this is empty.
  - if false socket: the value to give when the test is false. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** The value from if true or the value from if false.
- **How it works:** The bot works out the test. If it is true it gives the if true value, otherwise the if false value. Only the chosen side is used.
- **Example:**



The variable "stake" becomes 2 when the last digit is 5 or more, and 1 otherwise.

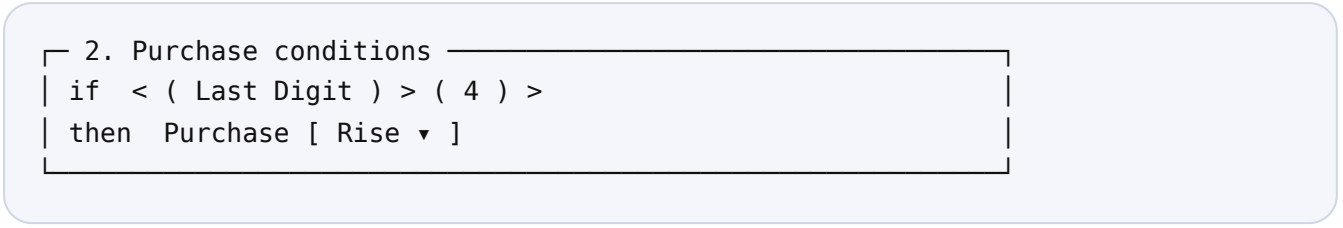
- **Good to know:**
  - The answer can be any value: a number, text or list.

Math

Number

- **Kind:** Official (Deriv)

- **Type id:** `math_number`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a number is wanted. The menu also puts one in many sockets by default. You can use it as many times as you like.
- **What it does:** A number you type in. Whole numbers and numbers with decimals both work.
- **Inputs:**
  - Number box. Default: 0. Use a dot for decimals, for example 0.5.
- **Gives back:** The number you typed.
- **How it works:** The block hands its number to the block it is plugged into. The box refuses text that is not a plain number (a minus sign, digits and one dot are fine) and refuses numbers beyond 9,007,199,254,740,991.
- **Example:**



The 4 is a Number block.

- **Good to know:**
  - Use . (a dot) for decimals, never a comma.
  - If a typed value is refused, the box keeps the old number.

Arithmetical operations

- **Kind:** Official (Deriv)
- **Type id:** `math_arithmetic`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Adds, subtracts, multiplies, divides or raises one number to the power of another.
- **Inputs:**
  - Left socket: a number. If it is empty the code uses 0.
  - Dropdown: +, -, *, /, ^ (power). The menu block starts on +.

  - Right socket: a number. If it is empty the code uses 0.
- **Gives back:** A number.
- **How it works:** The bot works out both sides and applies the sign. ^ means "to the power of": 2 ^ 3 is 2 times 2 times 2, which is 8.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ stake ▾ ] to ( ( stake ) * ( 2 ) )              │
└───────────────────────────────────────────────────────┘
```

The stake is doubled.

- **Good to know:**
  - An empty socket counts as 0, which can give a surprise. Fill both sockets.
  - The sockets accept blocks that give numbers, and variables.

Operations on a given number

- **Kind:** Official (Deriv)
- **Type id:** `math_single`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Does one operation on a single number: square root, absolute value, minus sign, natural log, log base 10, e to the power, or 10 to the power.
- **Inputs:**
  - Dropdown: square root, absolute, -, ln, log10, e^, 10^. The menu block starts on square root.
  - Number socket. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A number.
- **How it works:** The bot works out the number and applies the chosen operation. absolute drops the minus sign (-5 becomes 5). - flips the sign. ln and log10 are logarithms.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ size ▾ ] to ( absolute ( profit ) )             │
└───────────────────────────────────────────────────────┘
```

The variable "size" holds the profit without its minus sign.

- **Good to know:**
  - The square root of a negative number is not a real number. Check before you use it.
  - ln and log10 are for advanced maths and are rarely needed in a bot.

Trigonometric functions

- **Kind:** Official (Deriv)
- **Type id:** `math_trig`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Works out sin, cos, tan, asin, acos or atan of a number. Angles are in degrees.
- **Inputs:**
  - Dropdown: sin, cos, tan, asin, acos, atan. The menu block starts on sin.
  - Number socket. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A number.
- **How it works:** For sin, cos and tan the bot first turns your degrees into the angle the computer uses. For asin, acos and atan it turns the answer back into degrees.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
| set [ wave ▾ ] to ( sin ( 90 ) )                      |
└───────────────────────────────────────────────────────┘
```

The variable "wave" becomes 1.

- **Good to know:**
  - You rarely need these in a trading bot. They are here for people who build their own formulas.

Mathematical constants

- **Kind:** Official (Deriv)
- **Type id:** `math_constant`
- **Shape:** Value block (gives a number, text, list...)

- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Gives a famous fixed number such as pi.
- **Inputs:**
  - Dropdown: π, e, ϕ (golden ratio), sqrt(2), sqrt(½), ∞ (infinity). The menu block starts on π.
- **Gives back:** A number: about 3.14159 for π, 2.71828 for e, 1.61803 for ϕ, 1.41421 for sqrt(2), 0.70711 for sqrt(½), or infinity.
- **How it works:** The block hands the chosen constant to the block it is plugged into.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
| set [ half_turn ▾ ] to ( π )                          |
└───────────────────────────────────────────────────────┘
```

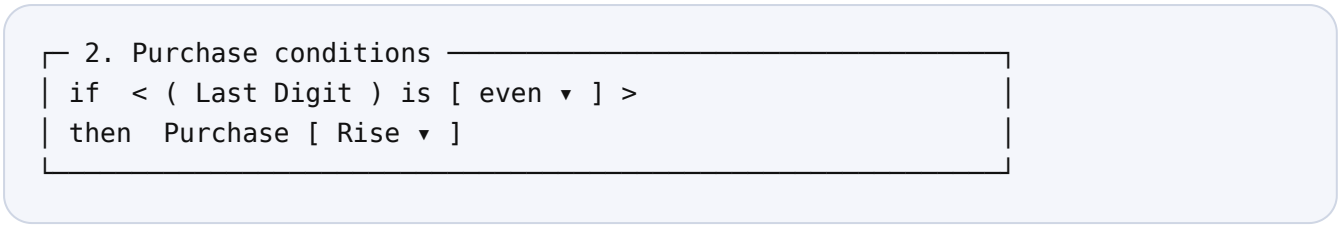
The variable "half_turn" holds about 3.14159.

- **Good to know:**
  - Infinity is bigger than any number, so it is handy as a starting value for a smallest-value search.

Test a number

- **Kind:** Official (Deriv)
- **Type id:** `math_number_property`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** Checks a property of a number: even, odd, prime, whole, positive, negative, or divisible by another number.
- **Inputs:**
  - Number socket (left): the number to check. The block marks itself red and the bot will not start if this is empty.
  - Dropdown: even, odd, prime, whole, positive, negative, divisible by. The menu block starts on even.
  - Extra socket: only for divisible by, it appears after the word and holds the divisor.
- **Gives back:** True or false.

- **How it works:** The bot checks the property. even and odd look at the remainder after dividing by 2. prime means divisible only by 1 and itself. whole means no decimals. positive means above 0 and negative means below 0.
- **Example:**



The bot buys when the last digit is 0, 2, 4, 6 or 8.

- **Good to know:**
  - odd is checked as "the remainder is 1", so a negative number such as -3 is not reported as odd.
  - A prime number is a whole number above 1 that has no divisor except 1 and itself, such as 2, 3, 5, 7.

Change variable

- **Kind:** Official (Deriv)
- **Type id:** `math_change`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Adds a number to a variable. Use a negative number to subtract.
- **Inputs:**
  - Variable dropdown: the variable to change. The default name is item.
  - by socket: the number to add. The block marks itself red and the bot will not start if this is empty.
- **How it works:** The bot adds the number to the variable and stores the result back. If the variable does not hold a number yet, it starts counting from 0.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ change [ losses ▾ ] by ( 1 )                            │
│ Trade again                                              │
└──────────────────────────────────────────────────────────┘
```

The variable "losses" goes up by one and the bot trades again.

- **Good to know:**
  - Make the variable first with Create in the Variables menu, and give it a start value with Set variable.
  - It is the quickest way to count things, such as the number of losses in a row.

Aggregate operations

- **Kind:** Official (Deriv)
- **Type id:** `math_on_list`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Works out one number from a whole list of numbers: sum, min, max, average, median, mode, antimode, standard deviation or a random item.
- **Inputs:**
  - Dropdown: sum, min, max, average, median, mode, antimode, standard deviation, random item. The menu block starts on sum.
  - List socket: a block that gives a list, such as a list variable or Last digits list. It must be a list-giving block.
- **Gives back:** A number. For mode and antimode it gives a list of the most (or least) common values.
- **How it works:** The bot runs through the list and calculates the chosen result. min is the smallest item and max the largest. average is the sum divided by the number of items. median is the middle item once sorted. mode is the most common item and antimode the least common. standard deviation tells how spread out the items are around their average.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ set [ avg ▾ ] to ( [ average ▾ ] of list ( closes ) )     │
└───────────────────────────────────────────────────────────┘
```

The variable "avg" holds the average of the numbers in "closes".

- **Good to know:**
  - Plugging a block that does not give a list into the socket turns the block red.
  - An empty list can give a surprising answer. Check the length first.

Rounding operation

- **Kind:** Official (Deriv)
- **Type id:** `math_round`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Rounds a number to a whole number: round, round up or round down.
- **Inputs:**
  - Dropdown: round, round up, round down. The menu block starts on round.
  - Number socket. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A whole number.
- **How it works:** round goes to the nearest whole number. round up always goes up (3.1 becomes 4). round down always goes down (3.9 becomes 3).
- **Example:**

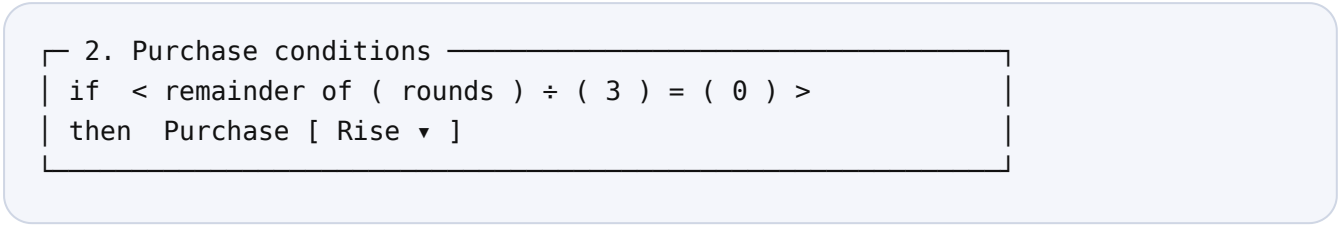
```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ set [ digits ▾ ] to ( round down ( 3.9 ) )                │
└───────────────────────────────────────────────────────────┘
```

The variable "digits" becomes 3.

- **Good to know:**
  - Round only the end result. Rounding in the middle of a calculation can add small errors.

Remainder after division

- **Kind:** Official (Deriv)
- **Type id:** `math_modulo`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Gives what is left over after dividing one number by another.
- **Inputs:**
  - Dividend socket (after "remainder of"): the number to divide. The block marks itself red and the bot will not start if this is empty.
  - Divisor socket (after the ÷ sign): the number to divide by. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A number. For 64 ÷ 10 it gives 4.
- **How it works:** The bot divides the first number by the second and gives the leftover part. 64 ÷ 10 is 6 with 4 left over, so the answer is 4.
- **Example:**



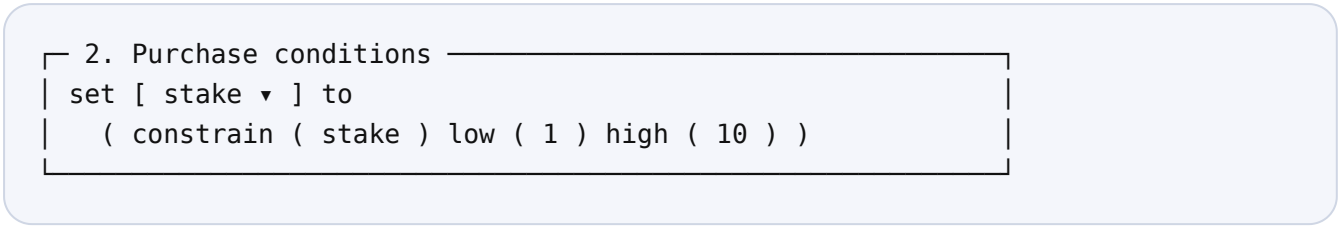
With a counter called "rounds", the bot buys on every third round.

- **Good to know:**
  - Dividing by 0 is not a sensible question. Do not plug in 0 as the divisor.
  - A common use: the remainder of a counter divided by 3 is 0 on every third round.

Constrain within a range

- **Kind:** Official (Deriv)
- **Type id:** `math_constrain`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.

- **What it does:** Keeps a number between a lowest and a highest value. If the number is too small you get the low value. If it is too big you get the high value.
- **Inputs:**
  - constrain socket: the number. The block marks itself red and the bot will not start if this is empty.
  - low socket: the lowest allowed value. The block marks itself red and the bot will not start if this is empty.
  - high socket: the highest allowed value. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A number between low and high.
- **How it works:** The bot first lifts the number up to low if it is smaller, then pushes it down to high if it is bigger.
- **Example:**



If the stake was 14, it becomes 10. If it was 0.5, it becomes 1.

- **Good to know:**
  - A handy way to stop a growing stake from going above a ceiling you chose.
  - It does not check that low is smaller than high. Keep them in the right order.

Random integer

- **Kind:** Official (Deriv)
- **Type id:** `math_random_int`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Gives a random whole number between two numbers. Both ends can be chosen.
- **Inputs:**
  - from socket: the lowest number. The block marks itself red and the bot will not start if this is empty.

  - to socket: the highest number. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A whole number from the first to the second number, both included.
- **How it works:** The bot picks at random. If you put the bigger number first, the code swaps them for you.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ pick ▾ ] to ( random integer from ( 0 ) to ( 9 ) )   │
└──────────────────────────────────────────────────────┘
```

The variable "pick" holds a random digit from 0 to 9.

- **Good to know:**
  - The numbers are random like a dice roll. They do not predict the market.

Random fraction number

- **Kind:** Official (Deriv)
- **Type id:** `math_random_float`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Gives a random number with decimals between 0.0 and 1.0.
- **Gives back:** A number from 0 up to (but not including) 1.
- **How it works:** The bot picks at random each time the block is used.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < ( random fraction ) < ( 0.5 ) >                 │
│ then  Purchase [ Rise ▾ ]                             │
└──────────────────────────────────────────────────────┘
```

About half of the time the bot buys.

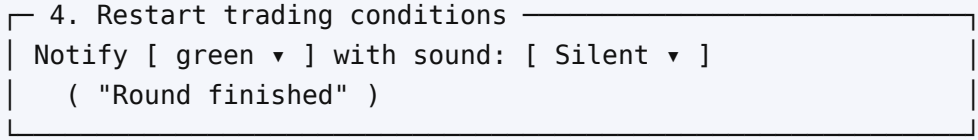
- **Good to know:**
  - Multiply it by a number to get a bigger range, for example times 100 for 0 to 100.

- Random does not mean smart. It does not give you an edge.

Text

Text

- **Kind:** Official (Deriv)
- **Type id:** `text`
- **Shape:** Value block (gives text)
- **Where it goes:** Anywhere a text is wanted. The menu also puts one in many sockets by default. You can use it as many times as you like.
- **What it does:** A piece of text that you type in, such as a word or a sentence.
- **Inputs:**
  - Text box. In the menu it starts as abc.
- **Gives back:** The text you typed.
- **How it works:** The block hands its text to the block it is plugged into. It is shown between quote marks.
- **Example:**



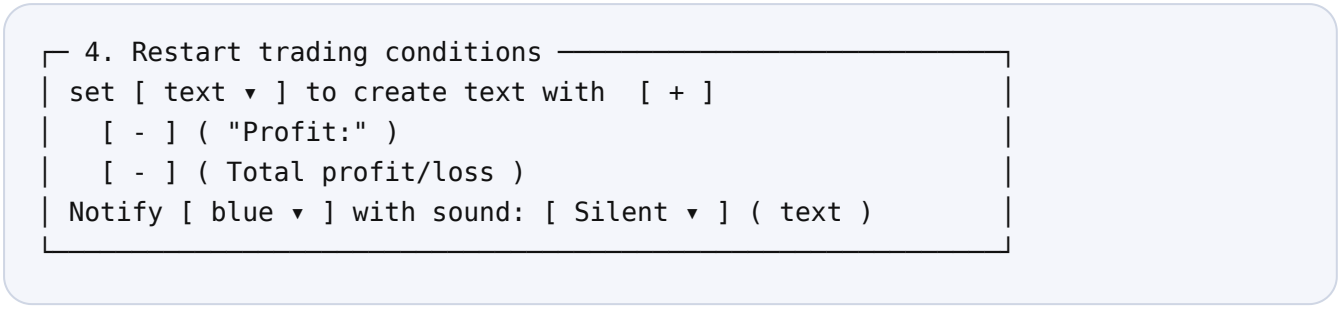
The Text block holds the message.

- **Good to know:**
  - Numbers typed in a Text block are text, not numbers. Use the Number block for maths.

Text join

- **Kind:** Official (Deriv)
- **Type id:** `text_join`
- **Shape:** Action block

- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Builds one text out of several pieces and stores it in a variable.
- **Inputs:**
  - Variable dropdown: where to store the result. The default name is text.
  - Items: a list of slots, each holding a text or any value. The plus icon adds a slot, the minus icon on a slot removes it. The menu block starts with one.
- **How it works:** The bot turns each item into text and joins them into one text with a single space between the items, then stores it in the variable.
- **Example:**



The variable "text" becomes something like Profit: 12.5 and the Journal shows it.

- **Good to know:**
  - Items are always joined with one space between them.
  - Only the item blocks that belong to this block can sit in its slot.

Text Append

- **Kind:** Official (Deriv)
- **Type id:** `text_append`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Adds text to the end of a text variable.
- **Inputs:**
  - Variable dropdown: the variable to add to. The default name is text.

  - text socket: what to add. The block marks itself red and the bot will not start if this is empty.
- **How it works:** The bot turns what you plugged in into text (numbers too) and puts it at the end of the variable.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ to [ log ▾ ] append text ( "W " )                        │
└──────────────────────────────────────────────────────────┘
```

Each time this runs, a "W " is added to the variable "log".

- **Good to know:**
  - Set the variable to a text first, for example an empty Text block, or the result can start with the word undefined.
  - Handy for building a short record of results.

Text String Length

- **Kind:** Official (Deriv)
- **Type id:** `text_length`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Counts the characters in a text, including spaces, digits and signs.
- **Inputs:**
  - Text socket (after "length of"). The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A number: the number of characters. "abc" gives 3.
- **How it works:** The bot counts every character in the text.
- **Example:**

```
┌─ 2. Purchase conditions ─────────────────────────────────┐
│ if  < length of ( "abc" ) = ( 3 ) >                      │
│ then  Purchase [ Rise ▾ ]                                │
└──────────────────────────────────────────────────────────┘
```

The text has 3 characters, so the test is true.

- **Good to know:**
  - To count items in a list, use List Length instead.

Text Is empty

- **Kind:** Official (Deriv)
- **Type id:** `text_isEmpty`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** Checks whether a text has nothing in it.
- **Inputs:**
  - Text socket (between "text" and "is empty"). The block marks itself red and the bot will not start if this is empty.
- **Gives back:** True if the text has no characters, otherwise false.
- **How it works:** The bot checks whether the length of the text is zero.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
| if  < text ( note ) is empty >                           |
| then  Trade again                                        |
└──────────────────────────────────────────────────────────┘
```

If "note" has no text, the bot trades again.

- **Good to know:**
  - A space is a character, so a text with only a space is not empty.
  - The socket takes text and variables.

Search for string

- **Kind:** Official (Deriv)
- **Type id:** `text_indexOf`
- **Shape:** Value block (gives a number)

- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Looks for a piece of text inside a longer text and tells you where it starts.
- **Inputs:**
  - in text socket: the text to search in. The block marks itself red and the bot will not start if this is empty.
  - Dropdown: first, last. Default: first. Which place to report if the piece appears more than once.
  - of text socket: the piece to look for. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** The position of the first letter of the match. The first letter of the text is position 1. If there is no match, it gives 0.
- **How it works:** The bot searches the text from the start (first) or the end (last) and gives the position found, counted from 1.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < in text ( note ) find [ first ▾ ] occurence     │
│       of text ( "win" ) > ( 0 ) >                     │
│ then  Purchase [ Rise ▾ ]                             │
└───────────────────────────────────────────────────────┘
```

The test is true when the word "win" appears somewhere in "note".

- **Good to know:**
  - Test for "greater than 0" to know if the text was found.
  - The plug of this block is marked as text, so a number-only socket may refuse it. A Compare block or a Set variable accepts it.

Get character

- **Kind:** Official (Deriv)
- **Type id:** `text_charAt`
- **Shape:** Value block (gives text)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Picks one letter out of a text.

- **Inputs:**
  - in text socket: the text. The block marks itself red and the bot will not start if this is empty.
  - Dropdown: letter #, letter # from end, first letter, last letter, random letter. Default: letter #.
  - Number socket: appears only for letter # and letter # from end. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** One character as text.
- **How it works:** The bot counts the letters from 1. letter # 1 is the first letter and letter # from end 1 is the last letter. random letter picks any letter by chance.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ first ▾ ] to                                    │
│   ( in text ( "945" ) get [ letter # ▾ ] ( 1 ) )      │
└───────────────────────────────────────────────────────┘
```

The variable "first" becomes the text "9".

- **Good to know:**
  - The result is text. "9" is not the number 9, although a Compare with = treats them as equal.

Get substring

- **Kind:** Official (Deriv)
- **Type id:** `text_getSubstring`
- **Shape:** Value block (gives text)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Cuts a piece out of a text, from one position to another.
- **Inputs:**
  - in text socket: the text. The block marks itself red and the bot will not start if this is empty.
  - First dropdown: letter #, letter # from end, first. Default: letter #. The two # choices show a number socket.
  - Second dropdown: letter #, letter # from end, last. Default: letter #. The two # choices show a number socket.
- **Gives back:** The piece of text, with both end letters included.

- **How it works:** The bot counts the letters from 1 and cuts from the first position to the second, including both. With first and last it gives the whole text.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ part ▾ ] to ( in text ( "ABCDEF" ) get substring     │
│   from [ letter # ▾ ] ( 2 ) to [ letter # ▾ ] ( 4 ) )      │
└──────────────────────────────────────────────────────┘
```

The variable "part" becomes "BCD".

- **Good to know:**
  - The menu block starts with the numbers 0 and 2. Counting starts at 1, so change the first number to 1 or more.

Change text case

- **Kind:** Official (Deriv)
- **Type id:** `text_changeCase`
- **Shape:** Value block (gives text)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Changes the capital letters of a text: ALL CAPS, all small, or Title Case.
- **Inputs:**
  - Text socket (after "Transform"). The block marks itself red and the bot will not start if this is empty.
  - Dropdown: UPPER CASE, lower case, Title Case. The menu block starts on UPPER CASE.
- **Gives back:** The changed text.
- **How it works:** The bot turns every letter big or small. Title Case makes the first letter of each word big and the rest small.
- **Example:**

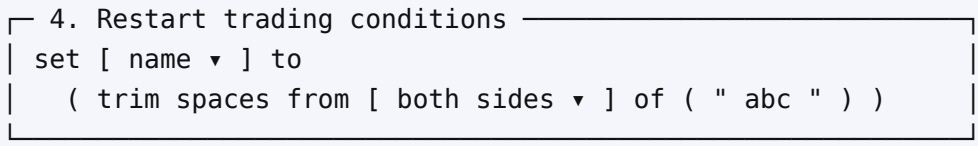
```
┌─ 4. Restart trading conditions ──────────────────────┐
│ Notify [ blue ▾ ] with sound: [ Silent ▾ ]                │
│   ( Transform ( "round done" ) to [ UPPER CASE ▾ ] )      │
└──────────────────────────────────────────────────────┘
```

The Journal shows ROUND DONE.

- **Good to know:**
  - Handy before you compare two texts, so that small and big letters do not matter.

Trim spaces

- **Kind:** Official (Deriv)
- **Type id:** `text_trim`
- **Shape:** Value block (gives text)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Cuts away the spaces at the start, the end, or both ends of a text.
- **Inputs:**
  - Dropdown: both sides, left side, right side. The menu block starts on both sides.
  - Text socket. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** The text without those spaces.
- **How it works:** The bot removes the blank characters from the chosen side or sides. Spaces in the middle stay.
- **Example:**



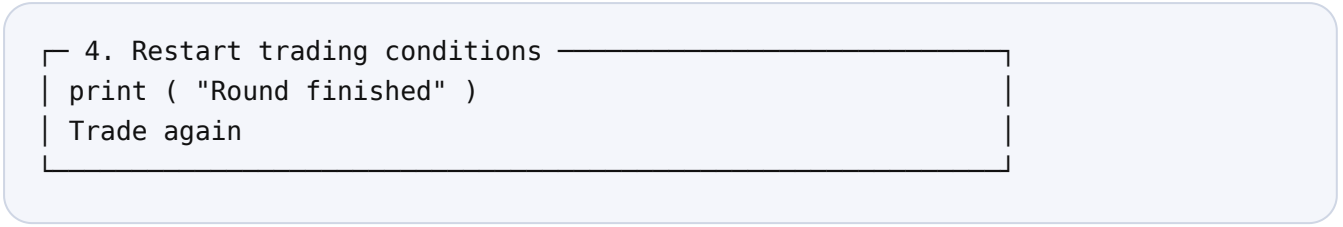
The variable "name" becomes "abc".

- **Good to know:**
  - Useful when a text comes from outside and may have extra spaces.

Print

- **Kind:** Official (Deriv)
- **Type id:** `text_print`
- **Shape:** Action block

- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like. In the menu it sits in Utility under Notifications.
- **What it does:** Shows a pop-up box with a message. The bot stops and waits until you tap OK.
- **Inputs:**
  - Message socket (after "print"). The block marks itself red and the bot will not start if this is empty.
- **How it works:** The bot asks the browser to show a pop-up with your message and does nothing more until you close it.
- **Example:**



A pop-up shows "Round finished". When you tap OK, the bot trades again.

- **Good to know:**
  - The pop-up freezes the bot. If you are away, the bot waits for you.
  - For a message that does not stop the bot, use Notify.

Request an input

- **Kind:** Official (Deriv)
- **Type id:** `text_prompt_ext`
- **Shape:** Value block (gives text or a number, as chosen)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like. In the menu it sits in Utility under Notifications.
- **What it does:** Shows a pop-up box that asks you to type something. The bot waits until you answer.
- **Inputs:**
  - Type dropdown: string or number. The menu block starts on string.
  - Message socket (after "with message"): the question to show. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** What you typed: as text for string, or as a number for number.

- **How it works:** The bot asks the browser to show the question with a typing box and does nothing more until you answer. For number, the answer is turned into a number.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ stake ▾ ] to                                    │
│   ( prompt for [ number ▾ ] with message ( "Stake?" ) )  │
└───────────────────────────────────────────────────────┘
```

A box asks "Stake?". The number you type is stored in the variable "stake".

- **Good to know:**
  - The bot stops until you answer. This is not good for a bot that should run on its own.
  - If you type something that is not a number while number is chosen, the result is not a number.

Text Statement

- **Kind:** Official (Deriv)
- **Type id:** `text_statement`
- **Shape:** Action block (a helper that lives inside Text join)
- **Where it goes:** Only inside the slot of a Text join block. You add it with the plus icon of Text join, not from the menu on its own.
- **What it does:** One item of a Text join block. It holds the text or value that is joined.
- **Inputs:**
  - Value socket: the text or value for this item. The minus icon on the item removes it.
- **How it works:** Inside Text join, the bot turns the value of each item into text and joins the items together.
- **Example:**

```
┌─ Text join ──────────────────────────────────────────┐
│ set [ text ▾ ] to create text with  [ + ]            │
│   [ - ] ( "abc" )                                    │
└──────────────────────────────────────────────────────┘
```

The indented line with the minus icon is one Text Statement.

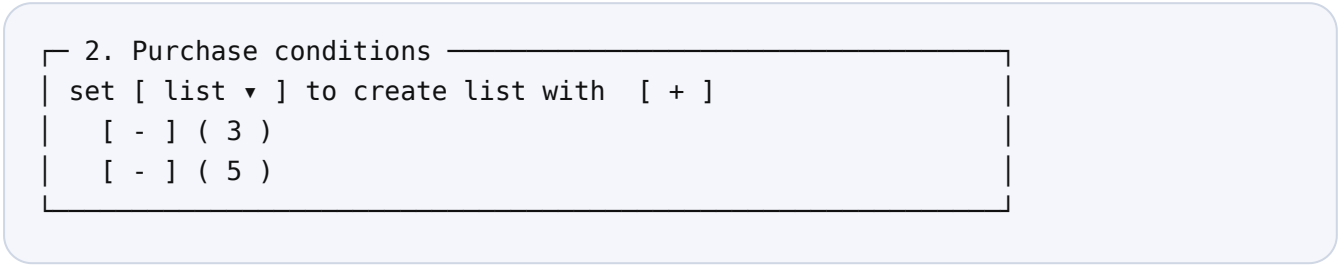
- **Good to know:**
  - If you drag it out of its Text join, it jumps back in.

- Plain Text blocks fit in its socket, but so do numbers and variables.

Lists

Create list

- **Kind:** Official (Deriv)
- **Type id:** `lists_create_with`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Makes a new list from items you list one by one and stores it in a variable. A list is an ordered row of values.
- **Inputs:**
  - Variable dropdown: where the list is stored. The default name is list.
  - Items: a stack of slots, each with a value. The plus icon adds a slot, the minus icon removes one. The menu block starts with two.
- **How it works:** The bot collects the values of the slots in order and stores them as a list in the variable. The old content of the variable is replaced.
- **Example:**



The variable "list" holds the list 3, 5.

- **Good to know:**
  - An empty slot gives the empty value null.
  - Items may be numbers, text or even other lists.

Repeat an item

- **Kind:** Official (Deriv)
- **Type id:** `lists_repeat`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a list is wanted (a socket of another block, or a Set variable). You can use it as many times as you like.
- **What it does:** Makes a list in which one item is repeated a number of times.
- **Inputs:**
  - item socket: the value to repeat. The block marks itself red and the bot will not start if this is empty.
  - times socket: how many copies. The block marks itself red and the bot will not start if this is empty. The menu block starts with 5.
- **Gives back:** A list.
- **How it works:** The bot makes a list with as many copies of the item as you asked for.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ zeros ▾ ] to                                    │
│   ( create list with item ( 0 ) repeated ( 5 ) times )  │
└───────────────────────────────────────────────────────┘
```

The variable "zeros" holds 0, 0, 0, 0, 0.

- **Good to know:**
  - Good for starting a list of counters, one for each digit.

List Length

- **Kind:** Official (Deriv)
- **Type id:** `lists_length`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Counts the items in a list.
- **Inputs:**

  - List socket (after "length of"). The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A number: how many items the list has.
- **How it works:** The bot counts the items of the list.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < length of ( digits ) ≥ ( 20 ) >                 │
│ then  Purchase [ Rise ▾ ]                             │
└───────────────────────────────────────────────────────┘
```

The bot only buys once the list "digits" has 20 items or more.

- **Good to know:**
  - For the number of characters in a text, use Text String Length.

Is list empty?

- **Kind:** Official (Deriv)
- **Type id:** `lists_isEmpty`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false is wanted, such as the test of an if. You can use it as many times as you like.
- **What it does:** Checks whether a list has no items.
- **Inputs:**
  - List socket. It accepts blocks that give a list, and variables. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** True if the list has no items, otherwise false.
- **How it works:** The bot checks if the list has length zero. A variable that holds nothing yet counts as empty.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < list ( digits ) is empty >                      │
│ then  Trade again                                     │
└───────────────────────────────────────────────────────┘
```

If "digits" has no items yet, the bot trades again.

- **Good to know:**
  - Check this before you read an item from a list, to avoid reading from nothing.

List item position

- **Kind:** Official (Deriv)
- **Type id:** `lists_indexOf`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Finds where an item is in a list.
- **Inputs:**
  - in list socket: the list. The block marks itself red and the bot will not start if this is empty.
  - Dropdown: first, last. Default: first. Which place to report if the item appears more than once.
  - item socket: the item to look for. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** The position of the item, counted from 1. If the item is not in the list, it gives 0.
- **How it works:** The bot searches from the start (first) or from the end (last) and gives the place found.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ set [ where ▾ ] to                                        │
│   ( in list ( digits ) find [ first ▾ ] occurence         │
│     of item ( 7 ) )                                       │
└───────────────────────────────────────────────────────────┘
```

The variable "where" holds the place of the first 7, or 0 if there is no 7.

- **Good to know:**
  - Test for "greater than 0" to know if the item was found.

Get list item

- **Kind:** Official (Deriv)

- **Type id:** `lists_getIndex`
- **Shape:** Value block (gives the item). With the remove choice it turns into an action block.
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like. With the remove choice it becomes an action block and goes inside a root block instead.
- **What it does:** Reads one item from a list. It can also take the item out of the list.
- **Inputs:**
  - in list socket: the list.
  - Mode dropdown: get, get and remove, remove. Default: get.
  - Place dropdown: #, # from end, first, last, random. Default: #. The two # choices show a number socket.
- **Gives back:** The item. With remove it gives nothing and is an action block.
- **How it works:** The bot counts the places from 1. # 1 is the first item. # from end 1 is the last item. get leaves the list as it was. get and remove gives the item and takes it out. remove only takes it out.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ set [ newest ▾ ] to                                       │
│   ( in list ( digits ) [ get ▾ ] [ last ▾ ] )             │
└───────────────────────────────────────────────────────────┘
```

The variable "newest" holds the last item of "digits". The list stays as it was.

- **Good to know:**
  - Taking an item from an empty list gives nothing useful. Check with Is list empty? first.
  - Switching to remove changes the block from a value block to an action block.

Set list item

- **Kind:** Official (Deriv)
- **Type id:** `lists_setIndex`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.

- **What it does:** Changes one item of a list, or inserts a new item at a place.
- **Inputs:**
  - in list socket: the list.
  - Mode dropdown: set, insert at. Default: set.
  - Place dropdown: #, # from end, first, last, random. Default: #. The two # choices show a number socket.
  - as socket: the new value.
- **How it works:** The bot counts the places from 1. set replaces the item at that place. insert at puts the new value in at that place and pushes the others along.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ in list ( digits ) [ insert at ▾ ] [ first ▾ ]        │
│   as ( Last Digit )                                   │
└───────────────────────────────────────────────────────┘
```

The newest last digit is put at the front of the list "digits".

- **Good to know:**
  - Keep a list at a fixed length by inserting at first and removing the last item.

Get sub-list

- **Kind:** Official (Deriv)
- **Type id:** `lists_getSublist`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a list is wanted (a socket of another block, or a Set variable). You can use it as many times as you like.
- **What it does:** Makes a smaller list out of a part of a list.
- **Inputs:**
  - in list socket: the list.
  - First dropdown: get sub-list from #, get sub-list from # from end, get sub-list from first. Default: from #.
  - Second dropdown: #, # from end, last. Default: #. The # choices show a number socket.
- **Gives back:** A new list with the items from the first place to the second place, both included.

- **How it works:** The bot counts places from 1, copies the items in the range, and gives them back as a new list. The original list is not changed.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ last5 ▾ ] to ( in list ( digits ) get sub-list  │
│   from [ # from end ▾ ] ( 5 ) to [ last ▾ ] )         │
└───────────────────────────────────────────────────────┘
```

The variable "last5" holds the last 5 items of "digits".

- **Good to know:**
  - Nice for taking the last few items of a long list.

Create list from text

- **Kind:** Official (Deriv)
- **Type id:** `lists_split`
- **Shape:** Value block (gives a number, text, list...) (gives a list, or text in the other mode)
- **Where it goes:** Anywhere a list is wanted (a socket of another block, or a Set variable). You can use it as many times as you like.
- **What it does:** Turns a text into a list by cutting it at a separator, or the other way round: joins a list into one text.
- **Inputs:**
  - Mode dropdown: make list from text, make text from list. Default: make list from text.
  - First socket: the text (or the list in the other mode).
  - with delimiter socket: the separator, for example a comma. The menu block starts with ,
- **Gives back:** A list (make list from text) or a text (make text from list).
- **How it works:** Make list from text cuts the text each time it sees the separator. Make text from list puts the separator between the items and joins them.
- **Example:**

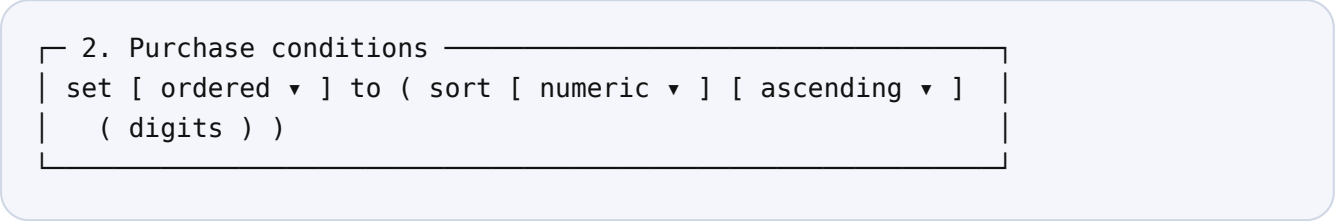
```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ nums ▾ ] to ( make list from text ( "3,5,7" )   │
│   with delimiter ( "," ) )                            │
└───────────────────────────────────────────────────────┘
```

The variable "nums" holds the list 3, 5, 7. The items are text, not numbers.

- **Good to know:**
  - Items cut from a text are text. A Compare with = still treats "5" and 5 as equal.

Sort list

- **Kind:** Official (Deriv)
- **Type id:** `lists_sort`
- **Shape:** Value block (gives a number, text, list...) (gives a list)
- **Where it goes:** Anywhere a list is wanted (a socket of another block, or a Set variable). You can use it as many times as you like.
- **What it does:** Gives the list in order, from small to big or big to small, by number or by alphabet.
- **Inputs:**
  - Type dropdown: numeric, alphabetic. Default: numeric.
  - Direction dropdown: ascending, descending. Default: ascending.
  - List socket. The block marks itself red and the bot will not start if this is empty.
- **Gives back:** A sorted list.
- **How it works:** The bot sorts the items by number value or by alphabet, upward (ascending) or downward (descending). In the code, the original list is also put in order as text as a side effect.
- **Example:**



The variable "ordered" holds the numbers of "digits", smallest first.

- **Good to know:**
  - If the order of your original list matters, make a copy of it first, because sorting can change it.

List Statement

- **Kind:** Official (Deriv)
- **Type id:** `lists_statement`

- **Shape:** Action block (a helper that lives inside Create list)
- **Where it goes:** Only inside the slot of a Create list block. You add it with the plus icon of Create list, not from the menu on its own.
- **What it does:** One item of a Create list block. It holds the value of that place in the list.
- **Inputs:**
  - Value socket: the value for this item. If it is empty the code uses null. The minus icon on the item removes it.
- **How it works:** Inside Create list, the bot takes the value of every item, in order, and puts them into the list.
- **Example:**

```
┌─ Create list ──────────────────────────────────────────┐
│ set [ list ▾ ] to create list with  [ + ]              │
│   [ - ] ( 3 )                                          │
└────────────────────────────────────────────────────────┘
```

The indented line with the minus icon is one List Statement.

- **Good to know:**
  - You cannot drag it out of its Create list block and use it alone. Dropped elsewhere it is removed.

Loops

Repeat (1)

- **Kind:** Official (Deriv)
- **Type id:** `controls_repeat`
- **Shape:** Action block
- **Where it goes:** Inside a root block or a function, like any action block. Other blocks go in its slot. Loops can sit inside other loops.
- **What it does:** Runs the blocks inside it a fixed number of times that you type in.
- **Inputs:**
  - Number box: how many times. Default: 10. Whole numbers from 0 up.
  - do slot: the blocks to repeat.

- **How it works:** The bot runs the blocks in the slot, one round after the other, until it has done as many rounds as the number.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ repeat [ 3 ] times                                        │
│   do  Notify [ blue ▾ ] with sound: [ Silent ▾ ] ( "hi" )  │
└───────────────────────────────────────────────────────────┘
```

The Journal shows "hi" three times.

- **Good to know:**
  - To use a variable or a calculation as the number, use Repeat (2).

Repeat (2)

- **Kind:** Official (Deriv)
- **Type id:** `controls_repeat_ext`
- **Shape:** Action block
- **Where it goes:** Inside a root block or a function, like any action block. Other blocks go in its slot. Loops can sit inside other loops.
- **What it does:** Runs the blocks inside it as many times as a value says, so the number can come from a variable.
- **Inputs:**
  - times socket: a number block. The block marks itself red and the bot will not start if this is empty.
  - do slot: the blocks to repeat.
- **How it works:** The bot works out the number once, then runs the blocks in the slot that many times.
- **Example:**

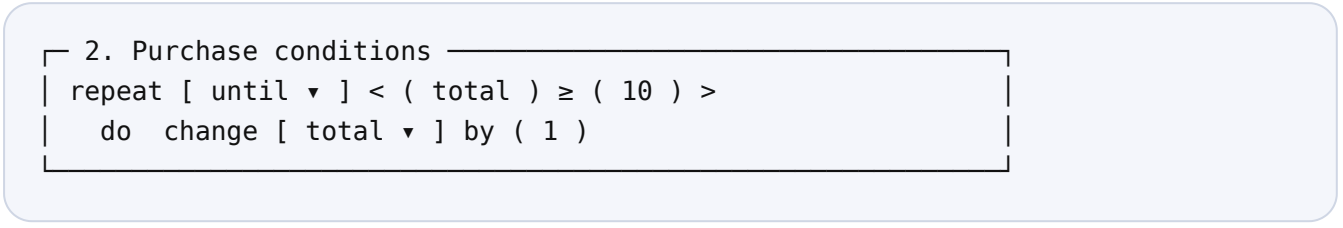
```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ repeat ( count ) times                                    │
│   do  change [ total ▾ ] by ( 2 )                         │
└───────────────────────────────────────────────────────────┘
```

If "count" is 4, the variable "total" goes up by 2 four times.

- **Good to know:**
  - Take care when the loop changes the same variable that gives the number of rounds. That can change how many rounds it runs.

Repeat While/Until

- **Kind:** Official (Deriv)
- **Type id:** `controls_whileUntil`
- **Shape:** Action block
- **Where it goes:** Inside a root block or a function, like any action block. Other blocks go in its slot. Loops can sit inside other loops.
- **What it does:** Keeps running the blocks inside while a test is true, or until a test becomes true.
- **Inputs:**
  - Dropdown: while, until. Default: while.
  - Test socket: a true/false block. The block marks itself red and the bot will not start if this is empty.
  - do slot: the blocks to repeat.
- **How it works:** Before every round the bot checks the test. With while it runs the slot as long as the test is true. With until it runs the slot as long as the test is false. The bot counts the rounds and stops with the error "Infinite loop detected" if a loop goes on for more than about 10,000 rounds.
- **Example:**



The variable "total" counts up by 1 until it reaches 10.

- **Good to know:**
  - Make sure something inside the loop can change the test, or the loop will never end.
  - The 10,000-round guard stops a runaway loop with an error. It does not give you a loop that waits for the market.

Iterate (1)

- **Kind:** Official (Deriv)
- **Type id:** `controls_for`
- **Shape:** Action block
- **Where it goes:** Inside a root block or a function, like any action block. Other blocks go in its slot. Loops can sit inside other loops.
- **What it does:** Counts with a variable from a start number to an end number in steps, and runs the blocks inside for each step.
- **Inputs:**
  - Variable dropdown: the counter (the block's help text calls it i).
  - from socket: the start number. The block marks itself red and the bot will not start if this is empty.
  - to socket: the end number. The block marks itself red and the bot will not start if this is empty.
  - by socket: the step size. The block marks itself red and the bot will not start if this is empty.
  - do slot: the blocks to run for each step.
- **How it works:** The variable starts at from and goes up (or down, if from is bigger than to) by the step each round, until it passes the end number. The end number is included. The step is always used as a positive size.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ count with [ i ▾ ] from ( 1 ) to ( 5 ) by ( 1 )       │
│   do  change [ total ▾ ] by ( i )                     │
└───────────────────────────────────────────────────────┘
```

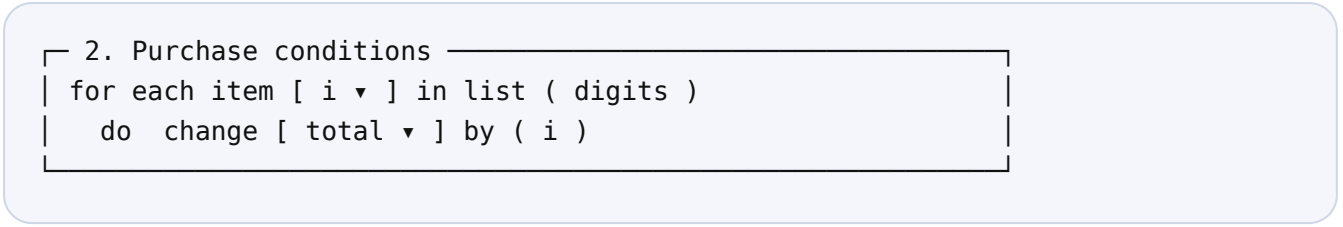
The variable "i" takes the values 1, 2, 3, 4, 5, so "total" grows by 15 in all.

- **Good to know:**
  - Use it when you know the start and end. Use a For each loop when you have a list.

Iterate (2)

- **Kind:** Official (Deriv)
- **Type id:** `controls_forEach`
- **Shape:** Action block

- **Where it goes:** Inside a root block or a function, like any action block. Other blocks go in its slot. Loops can sit inside other loops.
- **What it does:** Goes through a list one item at a time and runs the blocks inside for each item.
- **Inputs:**
  - Variable dropdown: holds the current item (the block's help text calls it i).
  - list socket: a list block. The block marks itself red and the bot will not start if this is empty.
  - do slot: the blocks to run for each item.
- **How it works:** For each item of the list, in order, the bot puts the item in the variable and runs the blocks in the slot.
- **Example:**



The variable "total" grows by every number in "digits".

- **Good to know:**
  - Do not add or remove items of the same list inside the loop. Work on a copy.

Break out/continue

- **Kind:** Official (Deriv)
- **Type id:** `controls_flow_statements`
- **Shape:** Action block
- **Where it goes:** Inside a loop block (Repeat, Repeat While/Until, Iterate), at any depth. Do not use it outside a loop.
- **What it does:** Leaves a loop early, or skips to the next round of the loop.
- **Inputs:**
  - Dropdown: break out, continue with next iteration. Default: break out.
- **How it works:** break out ends the loop at once and the bot goes on with the block after the loop. continue with next iteration skips the rest of this round and starts the next round.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ repeat [ 10 ] times                                   │
│   do  if  < ( i ) = ( 5 ) >                           │
│       then  break out of loop                         │
└───────────────────────────────────────────────────────┘
```

The loop ends when the test is true, even if it has not done all 10 rounds.

- **Good to know:**
  - Put it inside an if so that it only fires on a condition.
  - Outside of a loop it does nothing useful and the bot may stop with an error.

Variables

Set variable

- **Kind:** Official (Deriv)
- **Type id:** `variables_set`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Puts a value into a variable. A variable is a named box that holds a number, text or list for later.
- **Inputs:**
  - Variable dropdown: the variable to fill. Make a new one with the Create button in the Variables menu. The default name is item.
  - to socket: the value to store. If it is empty the code stores 0.
- **How it works:** The bot works out the value and stores it in the box. Whatever was in the box before is replaced.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ set [ stake ▾ ] to ( 1 )                              │
└───────────────────────────────────────────────────────┘
```

The variable "stake" now holds 1.

- **Good to know:**
  - Give a variable a start value before you change it or read it.
  - In the Variables menu, type a name in the New variable name box and press Create. Set and variable blocks for it then appear in that menu.

User-defined variable

- **Kind:** Official (Deriv)
- **Type id:** `variables_get`
- **Shape:** Value block (gives a number, text, list...) (can hold a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like. One is made for each variable you create, and it appears in the Variables menu.
- **What it does:** Hands over what is in a variable. Plug it in wherever you want to use the value.
- **Inputs:**
  - Variable dropdown: which variable to read. The default name is item.
- **Gives back:** The value the variable holds now.
- **How it works:** The bot looks up the variable and gives its current value. If you rename a variable, the blocks that use it follow the new name.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < ( stake ) > ( 5 ) >                             │
│ then  set [ stake ▾ ] to ( 1 )                        │
└───────────────────────────────────────────────────────┘
```

The block "stake" reads the variable. If it is above 5, it is reset to 1.

- **Good to know:**
  - A variable that was never set holds nothing, so give it a value first.
  - Pick short, clear names such as stake or losses.

Custom functions

Function

- **Kind:** Official (Deriv)
- **Type id:** `procedures_defnoreturn`
- **Shape:** Stand-alone block (a definition; it does not plug into other blocks)
- **Where it goes:** Straight on the empty canvas. You can build as many functions as you like, each with its own name.
- **What it does:** Defines your own named group of blocks that you can run from anywhere with a call block. Write the steps once and reuse them.
- **Inputs:**
  - Name box: the name of the function. The menu block starts as do something.
  - Plus icon: adds a parameter, a named value that the caller sends in. A box asks "Specify a parameter name:". The block then reads with: followed by the names.
  - Slot: the blocks to run when the function is called.
- **How it works:** Nothing runs where the function is defined. The blocks inside run each time a matching Custom function block is reached. When they finish, the bot goes on after the call block. Once you define a function, a call block with its name appears at the bottom of the Custom functions menu.
- **Example:**

```
┌─ Function ──────────────────────────────────────────────┐
│ function [ report ] with: total                         │
│   Notify [ blue ▾ ] with sound: [ Silent ▾ ] ( total )  │
└─────────────────────────────────────────────────────────┘
```

A function called "report" takes a value named "total" and writes it in the Journal.

- **Good to know:**
  - Give each function a different name.
  - The block can be collapsed with the small arrow, to keep the canvas tidy.

Function that returns a value

- **Kind:** Official (Deriv)

- **Type id:** `procedures_defreturn`
- **Shape:** Stand-alone block (a definition; it does not plug into other blocks)
- **Where it goes:** Straight on the empty canvas. You can build as many as you like, each with its own name.
- **What it does:** Like Function, but at the end it hands one value back to whoever called it, so you can use it as a value block.
- **Inputs:**
  - Name box: the name of the function. The menu block starts as do something.
  - Plus icon: adds a parameter (named value the caller sends in).
  - Slot: the blocks to run. Under the slot, return socket: the value to hand back.
- **How it works:** Nothing runs where the function is defined. When a matching call block is reached, the blocks run and the return value is given to the place where the call block was plugged in.
- **Example:**

```
┌─ Function that returns a value ──────────────────────────┐
│ function [ double ] with: n                              │
│   return ( ( n ) * ( 2 ) )                               │
└──────────────────────────────────────────────────────────┘
```

Calling "double" with 4 gives back 8.

- **Good to know:**
  - A function without a value in the return socket gives back nothing.
  - Use Conditional return to hand a value back early.

Conditional return

- **Kind:** Official (Deriv)
- **Type id:** `procedures_ifreturn`
- **Shape:** Action block
- **Where it goes:** Only inside a Function or a Function that returns a value, at any depth. Elsewhere the bot switches it off.
- **What it does:** Leaves the function early when a test is true, handing back a value.
- **Inputs:**

  - if socket: a true/false block.
  - return socket: the value to hand back. Inside a Function with no return value it shows only the word return.
- **How it works:** The bot checks the test. If it is true, the function ends at once with the return value. If it is false, the bot carries on with the next block.
- **Example:**

```
┌─ Function that returns a value ──────────────────────────┐
│ function [ safe_div ] with: a, b                         │
│   if < ( b ) = ( 0 ) > return ( 0 )                      │
│   return ( ( a ) / ( b ) )                               │
└──────────────────────────────────────────────────────────┘
```

If b is 0 the function returns 0 and never divides.

- **Good to know:**
  - Outside a function the block is switched off and does nothing.

Custom function

- **Kind:** Official (Deriv)
- **Type id:** `procedures_callnoreturn`
- **Shape:** Action block
- **Where it goes:** Inside a root block such as 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions (also inside a function, a loop or an if). You can use it as many times as you like.
- **What it does:** Runs one of your own functions (made with the Function block). It shows the function name.
- **Inputs:**
  - The name of the function, fixed by the Function block.
  - One socket for each parameter of the function. If one is empty the code sends null.
- **How it works:** The bot jumps to the function, runs its blocks with the values you plugged in, then comes back and carries on with the next block. It appears at the bottom of the Custom functions menu once the function exists.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────────┐
│ report ( Total profit/loss )                                 │
│ Trade again                                                  │
└──────────────────────────────────────────────────────────────┘
```

The function "report" runs with the total profit and then the bot trades again.

- **Good to know:**
  - Rename the function in its definition and every call block follows the new name.
  - Make the function first. The call block only appears in the menu after the function exists.

Custom function (returns a value)

- **Kind:** Official (Deriv)
- **Type id:** `procedures_callreturn`
- **Shape:** Value block (gives a number, text, list...)
- **Where it goes:** Anywhere a value is wanted (a socket of another block). You can use it as many times as you like.
- **What it does:** Runs one of your functions that hands back a value (made with the Function that returns a value block) and gives you that value.
- **Inputs:**
  - The name of the function, fixed by its definition.
  - One socket for each parameter. If one is empty the code sends null.
- **Gives back:** The value that the function returns.
- **How it works:** The bot jumps to the function, runs its blocks with your values, and the returned value takes the place of this block. It appears at the bottom of the Custom functions menu once the function exists.
- **Example:**

```
┌─ 2. Purchase conditions ─────────────────────────────────────┐
│ set [ stake ▾ ] to ( double ( stake ) )                      │
└──────────────────────────────────────────────────────────────┘
```

The function "double" is called with the stake and the answer is stored back in "stake".

- **Good to know:**

- The menu shows it under the same name "Custom function". The shape tells them apart: this one has a pill shape.

BRINOVS blocks: Execution and purchasing (1 to 9)

What are BRINOVS blocks?

BRINOVS blocks are extra blocks that sit on top of Deriv's own Bot Builder. You find them in the Blocks menu, in one folder called BRINOVS. That folder has 12 sub-folders and 51 blocks in all. Together they add safer buying, several contracts at once, stake plans, trading limits, entry filters, analysis tools and licensing to the blocks Deriv already gives you. Every BRINOVS folder has its own soft colour, so you can tell them from Deriv's pale grey blocks. On a BRINOVS block you can also use the menu (right-click on a computer, long press on a phone) to Disable Block or Enable Block.

Under the blocks there is a working part called the BRINOVS execution layer. Every purchase made by the normal Purchase block, by Multi purchase or by Hedge purchase passes through it. Before anything is sent to Deriv, the layer checks your safety rules: the connection is ready, your limits, your balance and your price ceiling. The blocks only set the numbers for these rules. The last numbers a block saved stay in force even if you later delete that block, and a rule that is set up but cannot decide says no. A pretend purchase (see Dry run mode) skips the money rules, because no money moves. No BRINOVS block makes a win more likely, and none of them promises profit.

Some words used in this guide. A **contract** is one bet on a market: you pick a type, a length of time and a stake. The **stake** is the money you put on one contract. The **payout** is the total Deriv gives back if the contract wins. A **tick** is one price update from the market. A **barrier** is the digit or price level a contract is measured against, for example the 1 in "Over 1". A **proposal** (also called an offer or a quote) is the live price Deriv shows before you buy. The **journal** is the message panel where the bot writes what it does.

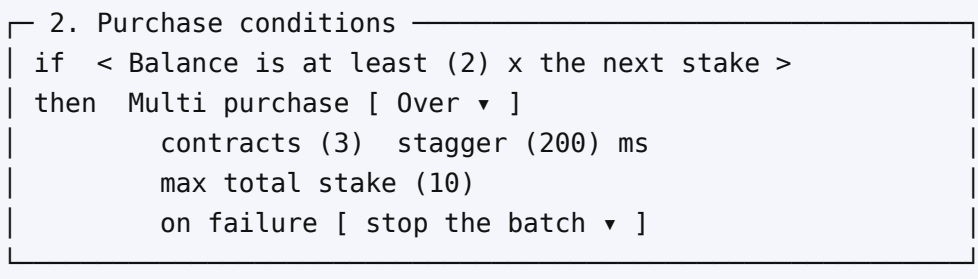
Execution and purchasing (blocks 1 to 9)

These blocks are about the moment of buying. You find them in the Blocks menu under BRINOVS, in the sub-folder "Execution and purchasing" (soft teal). In a bot they work in this order. Three switch blocks (Dry run mode, Hedge toggle, Buy price ceiling) go in "Run once at start" of "1. Trade parameters". Then, in "2. Purchase conditions", the check blocks (Minimum payout gate, Quote freshness gate, Contract compatibility gate) decide if the bot may buy, and Multi purchase or Hedge purchase does the buying. After the contracts finish, Batch result lets you read what happened.

The blocks

Multi purchase

- **Kind:** BRINOVS custom, block #1
- **Type id:** `bnv_multi_purchase`
- **Shape:** Action block (it ends the stack, like Purchase: no block can go under it)
- **Where it goes:** Inside "2. Purchase conditions", usually under an if. One per stack, in place of the Purchase block.
- **What it does:** Buys 2 to 10 contracts of the same type in one go, one after the other, with a pause you choose. It uses the market, duration, stake and prediction from your Trade options. Every contract is checked by your safety rules first and tracked on its own.
- **Inputs:**
  - Contract type: a dropdown filled from your Trade parameters, like Purchase.
  - contracts: a whole number from 2 to 10 (toolbox starts at 2).
  - stagger ms: a pause between contracts, 0 to 1000 (starts at 0).
  - max total stake: the most the whole batch may cost (starts at 10).
  - on failure: "stop the batch" (first choice) or "continue".
- **How it works:** The bot waits up to 30 seconds until it is connected and ready, then sends the contracts through the execution layer. The first contract bought becomes the one the normal bot flow follows. The bot moves on to the sell and restart steps only when every contract of the batch has finished. Each result is added to the totals.
- **Example:**



If the balance check is true, the bot buys 3 Over contracts, 200 ms apart. The total (3 x the stake) must be 10 or less.

- **Good to know:**

- It refuses and stops the bot (nothing is bought) if contracts x stake is above the max total stake, a box is empty or out of range, the Trade options use Payout instead of Stake, the contract has no duration, or a safety rule says no.
- "stop the batch" holds back the later contracts only when the stagger is above 0. With stagger 0 all contracts are sent together.
- If no contract at all can be bought, the bot stops with the reason and does not retry. Known limit: the Transactions panel lists only the first contract (every contract has a journal line), and Sell at market sells only the first.
- More contracts do not make winning more likely: they multiply the money at risk.

Hedge purchase

- **Kind:** BRINOVS custom, block #2
- **Type id:** `bnv_hedge_purchase`
- **Shape:** Action block (it ends the stack, like Purchase)
- **Where it goes:** Inside "2. Purchase conditions", usually under an if. One per stack.
- **What it does:** Buys 2 to 10 different contracts at the same moment (leg A, leg B, and more with the + button). Each leg has its own contract type, barrier and stake. Before buying, the journal shows the net result of every outcome, worked out from the real payouts.
- **Inputs:**
  - Leg A, Leg B (and up to Leg J): the contract type for each leg, filled from Trade parameters. A new leg starts with the next choice in the list.
  - barrier (each leg): the digit or price level. Empty means the prediction of your Trade options. Even and Odd take no barrier.
  - stake (each leg): a number (toolbox starts both legs at 1). From leg B on, a dropdown picks "amount" or "ratio of A" (a multiple of leg A's stake).
  - if a leg is not bought: "keep the bought legs" (first choice) or "sell the bought legs".
  - + and - buttons: add or remove the last leg (2 to 10 legs).
- **How it works:** All legs are sent at once through the execution layer, which checks all the stakes together. First the bot asks Deriv for one live price per leg and prints the outcome table. If only some legs are bought you are UNHEDGED: the journal warns you and, if you chose "sell", the bot tries to sell the bought legs (many contracts cannot be sold early).
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ then  Hedge purchase                                      │
│   Leg A [ Over ▾ ]  barrier (1)  stake (1)                │
│   Leg B [ Under ▾ ] barrier (8)  stake (1) [ amount ▾ ]   │
│   if a leg is not bought [ keep the bought legs ▾ ]       │
└───────────────────────────────────────────────────────────┘
```

The bot buys Over 1 and Under 8 together, 1 each. The journal first shows what each last digit would net.

- **Good to know:**
  - A hedge is not a safe bet. A win pays back less than the stake, so a round where only some legs win can still lose money.
  - It refuses and stops the bot if a stake is not above 0, a digit is out of range (Over 0 to 8, Under 1 to 9, Matches and Differs 0 to 9), a type is not picked, the Trade options use Payout, or a safety rule says no.
  - The outcome table only applies the contract rules. It does not say which outcome is likely. If a price cannot be read, no numbers are shown but the hedge is still placed.

Hedge toggle

- **Kind:** BRINOVS custom, block #3
- **Type id:** `bnv_hedge_toggle`
- **Shape:** Action block (it can have blocks above and below)
- **Where it goes:** In "Run once at start" of "1. Trade parameters". One is enough.
- **What it does:** One switch that turns every Purchase block into a hedge. While it is ON, each Purchase buys the contract and also its opposite, with the same stake, duration and prediction from your Trade options. This doubles the money you put in.
- **Inputs:**
  - mode: "ON (buy the opposite too)" (first choice) or "OFF (one contract)".
- **How it works:** The bot looks up the opposite of the contract in a fixed list: Even/Odd, Over/Under, Match/Differ, Rise/Fall, Higher/Lower, Only Ups/Only Downs, Touch/No Touch, Ends Between/Ends Outside, Stays Between/Goes Outside. It then buys both through the same path as Hedge purchase, with the same price preview and UNHEDGED warning (the bought leg is kept).
- **Example:**

```
┌─ 1. Trade parameters ──────────────────────────────────────┐
│ Run once at start:                                         │
│   Hedge toggle [ ON (buy the opposite too) ▾ ]             │
└────────────────────────────────────────────────────────────┘
```

Now a Purchase [Even ▾] buys Even and Odd together, each with your stake.

- **Good to know:**
  - If the contract has no opposite in the list, the bot stops with a message instead of guessing. Switch the toggle OFF to buy it alone.
  - Some pairs can lose on both sides: Over and Under on the same digit both lose when that digit comes up, and Rise and Fall both lose on an unchanged price. The preview table shows this.
  - A hedge is not a safe bet: buying both sides costs money on average.
  - It needs the execution layer. If the layer is off, the block stops the bot instead of buying one side only.

Batch result

- **Kind:** BRINOVS custom, block #4
- **Type id:** `bnv_batch_result`
- **Shape:** Value block (gives a number, text or list)
- **Where it goes:** Anywhere a value is wanted. It is made for "3. Sell conditions" and "4. Restart trading conditions". It can be used many times.
- **What it does:** Reads one number or word about the latest Multi purchase or Hedge purchase. It lets your after-purchase logic see the combined result of all the contracts, which the normal single-contract blocks cannot do.
- **Inputs:**
  - field: requested (first choice), confirmed, failed, stake committed, net profit, status, contract id list.
- **Gives back:** requested, confirmed, failed and stake committed are numbers. net profit is a number, but not a number until every contract has finished. status is one of none, complete, partial, failed, denied or unknown. contract id list is a list. With no batch yet you get 0, none or an empty list.
- **How it works:** The execution layer keeps the result of the latest batch. This block only reads it. "failed" counts contracts that Deriv or the bot refused; contracts that were never sent because the batch stopped are not counted as failed.

- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ if  < ( Batch result [ confirmed ▾ ] ) < ( 3 ) >         │
│ then  Trade again                                        │
└──────────────────────────────────────────────────────────┘
```

If fewer than 3 contracts of the last batch were bought, the bot starts a new round.

- **Good to know:**
  - It describes what already happened. It does not predict the next result.
  - Partial results are normal: some contracts of a batch can fail while others are bought. Check "status" for "partial".
  - Compare net profit only after the contracts finish; before that it is not a number and every comparison with it is false.

Dry run mode

- **Kind:** BRINOVS custom, block #5
- **Type id:** `bnv_dry_run_mode`
- **Shape:** Action block (it can have blocks above and below)
- **Where it goes:** In "Run once at start" of "1. Trade parameters". One is enough.
- **What it does:** A free practice switch. While it is ON, every purchase of the bot (Purchase, Multi purchase, Hedge purchase) is only pretended. Nothing is bought and no money moves. Prices and payouts are real, and each pretend contract is settled from the real ticks.
- **Inputs:**
  - mode: "ON (nothing is bought)" (first choice) or "OFF (real purchases)".
- **How it works:** The bot still asks Deriv for the real price, so a closed market or a bad contract fails just like a real buy. The pretend contract then ends on the real ticks, so restart conditions, stake plans and totals behave as in a real run. Pretend contracts have a negative number in the journal.
- **Example:**

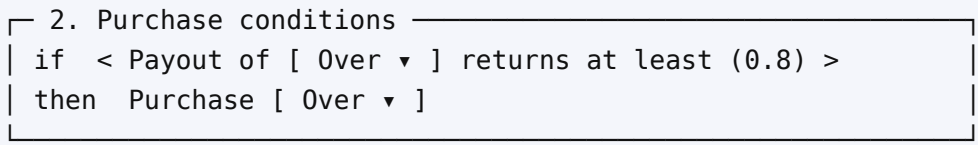
```
┌─ 1. Trade parameters ────────────────────────────────────┐
│ Run once at start:                                       │
│   Dry run [ ON (nothing is bought) ▾ ]                   │
└──────────────────────────────────────────────────────────┘
```

The bot trades as normal, but nothing is bought and your balance does not change. The journal says DRY-RUN is ON.

- **Good to know:**
  - Only digit contracts (Even, Odd, Over, Under, Matches, Differs) and Rise/Fall (including the "or equal" types) can be pretended. Anything else stops the bot with a plain message.
  - Without this block, or with OFF, purchases are real. Check the first journal lines: they say whether dry run is ON or OFF.
  - Pretend results are not real money and are not a promise of real results. It needs the execution layer; without it the bot stops instead of buying for real.

Minimum payout gate

- **Kind:** BRINOVS custom, block #6
- **Type id:** `bnv_min_payout_gate` (its helper block "Return ratio" is `bnv_min_payout_gate_ratio`)
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false block is wanted, usually in an if in "2. Purchase conditions".
- **What it does:** Says true when the live offer for the contract pays at least the minimum you set. Use it to skip offers that pay too little.
- **Inputs:**
  - Contract type: a dropdown filled from Trade parameters.
  - returns at least: a number (toolbox starts at 0.8). 0.8 means a win returns 80% of the stake as profit.
- **Gives back:** True when (payout - stake) / stake of the live offer is at least your number. False when it pays less, or when there is no readable offer.
- **How it works:** It looks at the offer the bot already holds for this purchase cycle and makes no new request. The companion value block "Return ratio" gives the same calculated number itself (not a number when there is no offer). A bot that uses either block asks Deriv for live offers.
- **Example:**

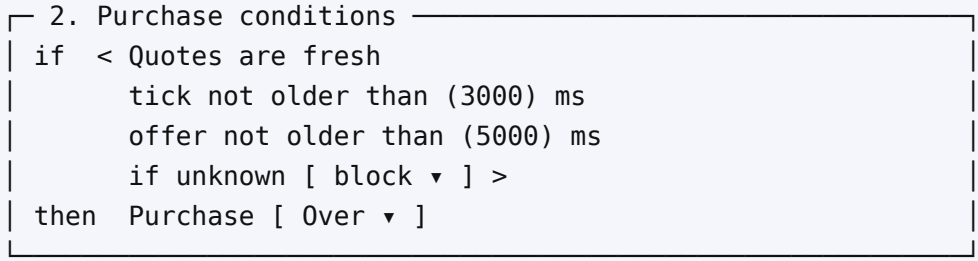


The bot buys Over only when the live offer returns at least 80% of the stake as profit.

- **Good to know:**
  - An empty or non-number box stops the bot with a message.
  - A higher payout does not mean a higher chance to win. Do not treat it as a quality signal.
  - Unknown means false: with no readable offer the gate blocks.

Quote freshness gate

- **Kind:** BRINOVS custom, block #7
- **Type id:** `bnv_quote_freshness_gate`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false block is wanted, usually in an if in "2. Purchase conditions".
- **What it does:** Says true when the last tick and the live offer are recent enough. It stops the bot from buying on old data after a lag or a lost connection.
- **Inputs:**
  - tick not older than ... ms: a number (toolbox starts at 3000).
  - offer not older than ... ms: a number (toolbox starts at 5000).
  - if unknown: "block" (first choice) or "allow".
- **Gives back:** True when both are fresh. False when either is too old, or when an age cannot be known (for example right after the start), unless you chose "allow".
- **How it works:** For ticks the bot checks when the tick arrived and also how late it was compared with its own time, against the usual delay. For the offer it checks the time since its last update from Deriv.
- **Example:**



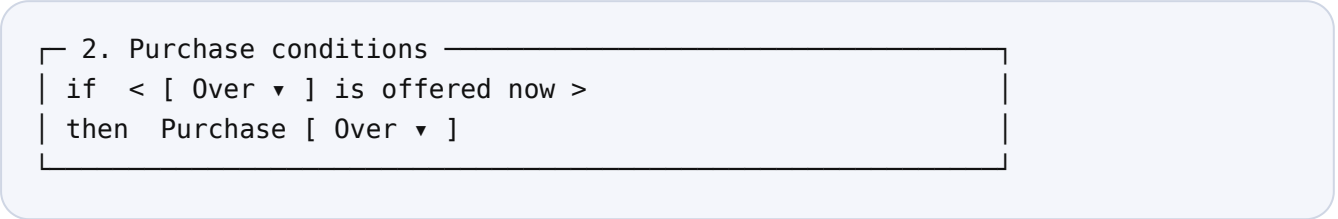
The bot buys only when the last tick is under 3 seconds old and the offer under 5 seconds old.

- **Good to know:**

- Both ages must be numbers above 0, or the bot stops with a message.
- Keep "block" as the choice unless you have a reason: it is the safe default.
- It checks how fresh the data is. It says nothing about the chance to win.

Contract compatibility gate

- **Kind:** BRINOVS custom, block #8
- **Type id:** `bnv_contract_compat_gate`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false block is wanted, usually in an if just before Purchase.
- **What it does:** Says true only when Deriv's live list for this market offers the contract you are about to buy. It checks the contract type, the duration and its unit, and the prediction digit or barrier from your Trade options.
- **Inputs:**
  - Contract type: a dropdown filled from Trade parameters ("... is offered now").
- **Gives back:** True when the contract is offered. False when it is not, or when anything is unclear (no answer in about 8 seconds, an empty list, a duration or barrier it cannot read).
- **How it works:** The bot asks Deriv for the live list of contracts for the market (it shares one answer for 5 seconds, so a loop cannot flood Deriv). When the answer is false, the journal says why, once for each reason. It never swaps to a different contract.
- **Example:**

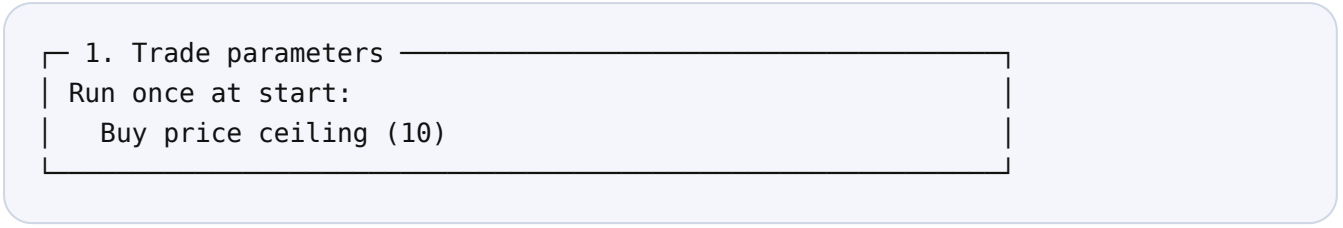


The bot buys Over only when Deriv currently offers Over with your duration and prediction.

- **Good to know:**
  - It only checks that the contract exists. It says nothing about the chance to win.
  - Unclear means false, so a slow or missing answer from Deriv blocks the purchase.
  - Use it to avoid repeated refused buys, for example a duration the market does not offer.

Buy price ceiling

- **Kind:** BRINOVS custom, block #9
- **Type id:** `bnv_buy_price_ceiling`
- **Shape:** Action block (it can have blocks above and below)
- **Where it goes:** In "Run once at start" of "1. Trade parameters". One is enough.
- **What it does:** Sets the most the bot will ever pay for one contract. Before every purchase (Purchase, Multi purchase, Hedge purchase) the bot checks the price. If it is above your number, nothing is bought and the bot stops with the reason.
- **Inputs:**
  - Buy price ceiling: a number above 0, in your account currency (toolbox starts at 10).
- **How it works:** The ceiling is one of the safety rules in the execution layer. A buy request is only sent to Deriv when its price is at or below the ceiling. In Multi purchase and Hedge purchase each contract is checked on its own.
- **Example:**



A contract that would cost 12 is refused and the bot stops with a message. A contract that costs 10 or less goes through.

- **Good to know:**
  - Without this block there is no ceiling.
  - A price the bot cannot read (for example a payout-based contract with no live offer) is also refused.
  - It needs the execution layer. If the layer is off, the block stops the bot instead of looking protected.
  - It protects your money. It does not change your chance of winning.

BRINOVS blocks: Stake management (10 to 13)

Stake management (blocks 10 to 13)

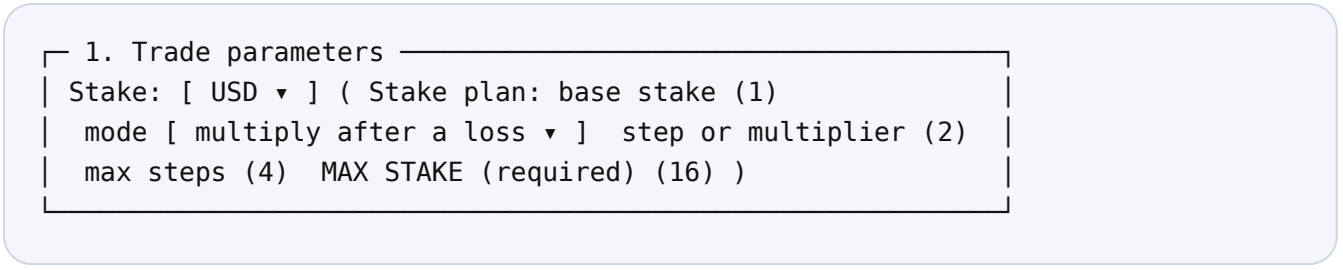
The **stake** is the money you put on one contract. The four blocks in this section work out a stake from what has already happened: your finished rounds, your balance, or how much the market is moving. They are value blocks, so each one gives back a number. You find them in the Blocks menu under BRINOVS, in the sub-folder "Stake management" (soft violet). The usual place is the Stake box of the Trade options inside "1. Trade parameters"; the bot asks the block for the next stake each time it needs one. You can also put one block inside another, for example a Stake plan inside a Stake percent cap. None of these blocks makes a win more likely, and raising a stake after a loss can empty an account quickly.

The blocks

Stake plan

- **Kind:** BRINOVS custom, block #10
- **Type id:** `bnv_stake_plan`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. It is made for the Stake box of the Trade options in "1. Trade parameters".
- **What it does:** Gives the next stake, worked out from your finished rounds. It can keep the stake fixed, or raise it after each loss by a step, by a multiplier or along your own list. A maximum stake is required.
- **Inputs:**
  - Stake plan: base stake: the starting stake (toolbox starts at 1).
  - mode: "fixed" (first choice), "add a step after a loss", "multiply after a loss" or "custom sequence".
  - step or multiplier: the amount added, or the number multiplied by, after each loss (starts at 2).
  - custom sequence: your own list of stakes, typed with commas (starts as 1, 2, 4, 8). Used only in custom mode.

  - max steps: how many raises are allowed, a whole number from 1 to 50 (starts at 4).
  - MAX STAKE (required): the hard cap, a number above 0 (starts at 16).
  - go back to the base stake: "after a win" (first choice), "only after max steps" or "by hand (Resume ticket)".
  - Resume ticket: a whole number, 0 or more (starts at 0).
- **Gives back:** The next stake, as a number, never above MAX STAKE.
- **How it works:** A round is one purchase; a hedge or a multi-contract purchase counts as one round with its profits added up. A loss moves one step up. A loss at the last step goes back to the base stake. A win goes back to the base stake only if you chose "after a win". A round that ends at exactly 0 changes nothing. To restart by hand, raise the Resume ticket above its old value.
- **Example:**



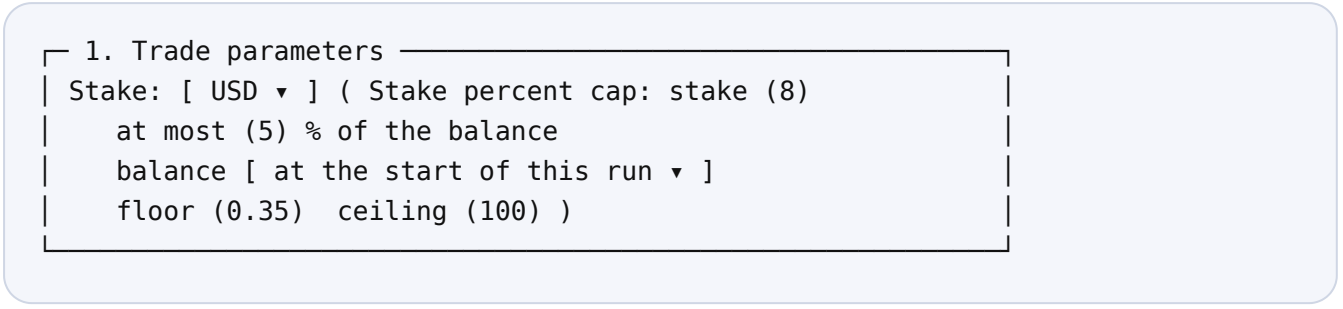
After losses the stakes run 1, 2, 4, 8, 16. A loss at 16 goes back to 1, and with the first "go back" choice a win does too. The tooltip shows the worst case, 16.

- **Good to know:**
  - It refuses and stops the bot without a MAX STAKE, if the base stake is above it, or if a custom sequence has a number above it or fewer than 2 numbers.
  - Once the block has run, its MAX STAKE is enforced on every real purchase even if you delete the block: a larger stake is refused. A pretend (dry run) purchase is not stopped and has its own step count.
  - It needs the execution layer. Multiply modes can empty an account quickly, so pick a cap you can afford to lose.

Stake percent cap

- **Kind:** BRINOVS custom, block #11
- **Type id:** `bnv_stake_percent_cap`
- **Shape:** Value block (gives a number)

- **Where it goes:** Anywhere a number is wanted, usually the Stake box of the Trade options in "1. Trade parameters". It can wrap a Stake plan.
- **What it does:** Keeps a stake from being bigger than a percentage of your balance, and never above your ceiling. It asks Deriv for a fresh balance every time it works out a stake.
- **Inputs:**
  - Stake percent cap: stake: the stake to check, a number above 0 (starts at 1).
  - at most ... % of the balance: above 0 and at most 100 (starts at 5).
  - balance to use: "at the start of this run" (first choice) or "now".
  - floor: the smallest stake you accept, 0 or more (starts at 0.35).
  - ceiling: the biggest stake you accept, above 0 (starts at 100).
- **Gives back:** An approved stake: the smaller of your stake, the percentage of the balance and the ceiling, rounded down to cents.
- **How it works:** "At the start of this run" uses the first fresh balance the bot saw in this run. "now" uses the balance at that moment. If the stake is cut, the journal says so. If the cap comes out below your floor, the bot stops instead of raising the stake.
- **Example:**



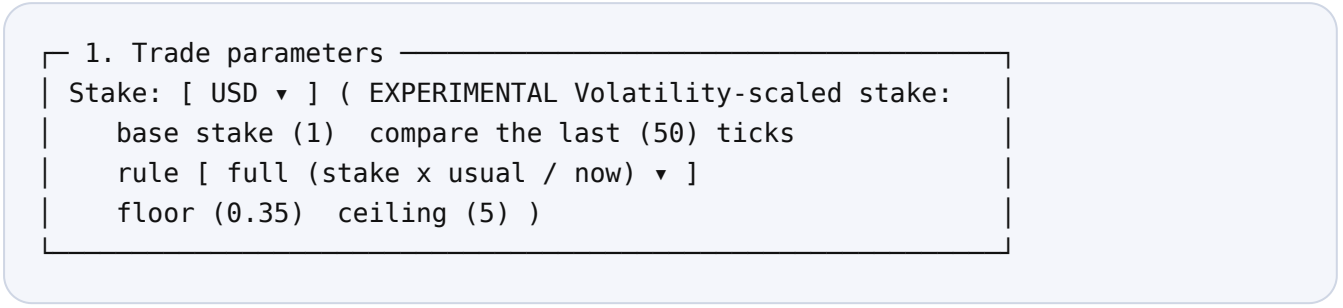
With a starting balance of 100, 5% is 5. The stake of 8 is cut to 5.

- **Good to know:**
  - The bot stops with the reason if the balance is unknown, zero, older than 60 seconds, or in a different currency than the trade.
  - It also stops if the floor is above the ceiling, or the cap is below the floor.
  - It checks only a stake that goes through this block. Unlike the Stake plan cap, it is not enforced on every purchase.

Volatility-scaled stake (experimental)

- **Kind:** BRINOVS custom, block #12

- **Type id:** `bnv_volatility_stake`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted, usually the Stake box of the Trade options in "1. Trade parameters".
- **What it does:** Makes the stake smaller when the market moves more than usual and bigger when it is calm, always between your floor and your ceiling. Volatility means how much the price jumps around.
- **Inputs:**
  - EXPERIMENTAL Volatility-scaled stake: base stake: a number above 0 (starts at 1).
  - compare the last ... ticks: a whole number from 10 to 200 (starts at 50).
  - rule: "full (stake x usual / now)" (first choice) or "gentle (square root of that)".
  - floor: the smallest stake, 0 or more (starts at 0.35).
  - ceiling: the biggest stake, above 0 (starts at 5).
- **Gives back:** The scaled stake as a number, between the floor and the ceiling, rounded down to cents.
- **How it works:** "Usual" is the average size of a tick move over the last 1000 ticks of the traded market. "Now" is the same measure over the last ticks you choose. Full means base stake x usual / now. Gentle means base stake x the square root of that.
- **Example:**



If the market moves twice as much as usual, the stake becomes 0.5. If it is half as lively as usual, the stake becomes 2.

- **Good to know:**
  - It is experimental. The bot stops with the reason if the price history cannot be read, has fewer than 100 prices, or did not move at all.
  - A bigger stake in calm periods is a bigger risk. Set a ceiling you can afford.
  - It does not make a win more likely.

Position size (experimental)

- **Kind:** BRINOVS custom, block #13
- **Type id:** `bnv_position_size`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted.
- **What it does:** A calculator for multiplier-style trades with a stop loss and a take profit. It is not useful for digit contracts. It only does arithmetic and buys nothing.
- **Inputs:**
  - EXPERIMENTAL Position size: capital: your money for the trade (starts at 1000).
  - risk ... % of it: the share of the capital you risk, above 0 and at most 100 (starts at 1).
  - entry: the entry price (starts at 100).
  - stop loss: the price where you give up (starts at 99).
  - take profit: the price where you take the win (starts at 102).
  - give me: "the position size" (first choice), "the risk / reward ratio" or "the amount at risk".
- **Gives back:** A number. The amount at risk is capital x risk share. The position size is that amount divided by the distance from the entry to the stop loss. The ratio is the distance to the take profit divided by the distance to the stop loss.
- **How it works:** It calculates and returns a number straight away. It does not look at the market. The position size is rounded down to 4 decimals, the amount at risk to cents and the ratio to 2 decimals.
- **Example:**

```
┌─ 1. Trade parameters ──────────────────────────────────┐
│ Run once at start:                                     │
│   set [ size ▾ ] to                                    │
│     ( EXPERIMENTAL Position size: capital (1000)       │
│       risk (1) % of it  entry (100)  stop loss (99)    │
│       take profit (102)  give me [ the position size ▾ ] ) │
└────────────────────────────────────────────────────────┘
```

The amount at risk is 10 and the stop loss is 1 away from the entry, so the size is 10. The ratio would be 2 and the amount at risk 10.

- **Good to know:**
  - The stop loss and the take profit must be on opposite sides of the entry, and neither can equal it, or the bot stops with the reason.

- The entry, the stop loss and the take profit must all be prices above 0.
- It is only arithmetic. It does not make a win more likely.

BRINOVS blocks: Risk management (14 to 19)

Risk management (blocks 14 to 19)

These blocks help you keep a bot inside limits you choose. They stop NEW purchases when a limit is reached. Contracts that are already open are never touched. You find them in the Blocks menu under BRINOVS, in the sub-folder "Risk management" (soft sand). In a bot, the setting blocks (Session limits, Exposure cap) go in "Run once at start" of "1. Trade parameters". The question blocks (Balance guard, Drawdown breaker, Trading window) usually sit in an if inside "2. Purchase conditions", and Loss streak pause is made for "4. Restart trading conditions". The limits are checked by the BRINOVS execution layer before every purchase. The block saves your numbers, and the last saved numbers stay in force even if you delete the block. A guard that is set up but cannot decide says no. Pretend (dry run) purchases are not stopped by the money limits.

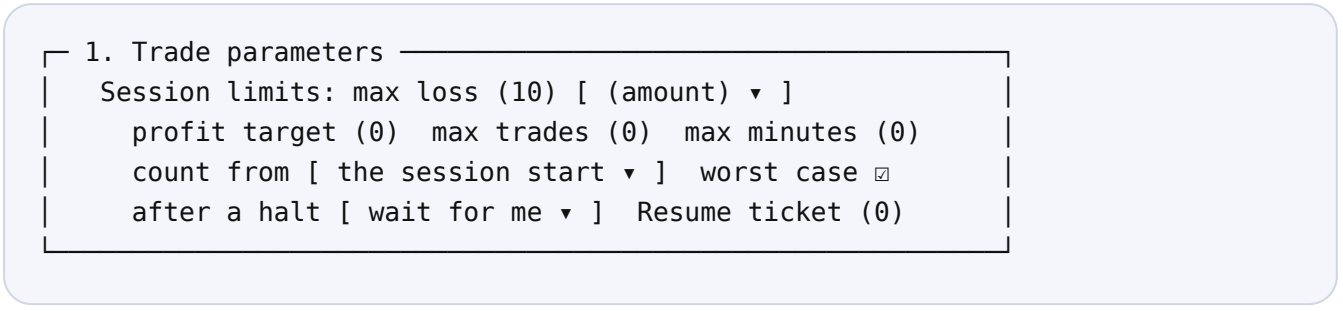
Two words used here. The **balance** is the money in your account. A **round** is one purchase; a hedge or a multi-contract purchase counts as one round, with all its profits and losses added together. The folder also holds a helper value block, Risk status. It reads ok or halted, the halt reason, the session profit or loss, open contracts and their stake, the balance, the losing rounds in a row and the drawdown. These blocks protect your money to a degree, but they cannot stop a single contract from losing its stake, and they never change your chance of winning.

The blocks

Session limits

- **Kind:** BRINOVS custom, block #14
- **Type id:** `bnv_session_limits`
- **Shape:** Action block (it can have blocks above and below)
- **Where it goes:** In "Run once at start" of "1. Trade parameters". Only inside the Trade parameters block.
- **What it does:** Stops all new purchases once a limit is hit: a maximum loss, an optional profit target, a maximum number of trades or a maximum time. Contracts that are already open keep running.
- **Inputs:**

  - Session limits: max loss: a number above 0 (toolbox starts at 10), with a dropdown "(amount)" (first choice) or "% of start balance".
  - profit target: 0 means off (starts at 0).
  - max trades: a whole number, 0 means off (starts at 0).
  - max minutes: 0 means off (starts at 0).
  - count from: "the session start" (first choice) or "00:00 UTC today".
  - after a halt: "wait for me (Resume ticket)" (first choice) or "resume next UTC day".
  - worst case: count open stakes: a tick box, ticked at the start.
  - Resume ticket: a whole number, 0 or more (starts at 0).
- **How it works:** Before every purchase the bot adds up your finished real trades since the start point and compares them with the limits. With worst case ticked, it also counts the stakes of open contracts and of this purchase as if they all lost. When a limit is hit, the halt is saved: a page refresh does not clear it. Only a higher Resume ticket (or the next UTC day, if you chose that) starts a new session.
- **Example:**



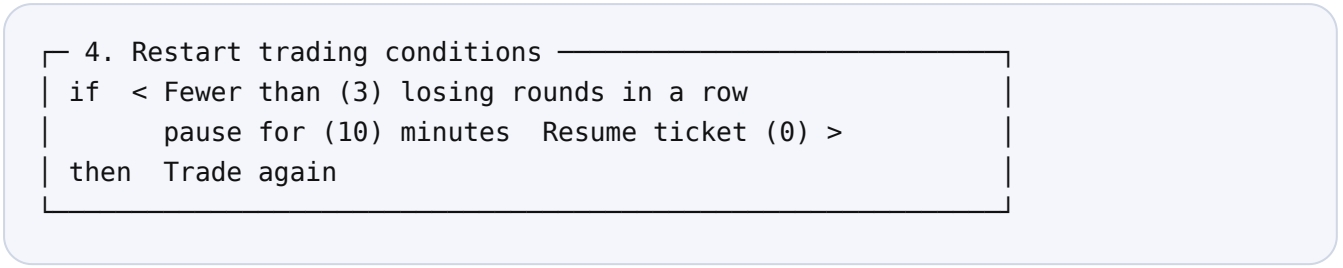
Once the finished trades have lost 10, the next purchase is refused and the bot stops with the reason.

- **Good to know:**
  - It refuses and stops the bot if the max loss is not above 0, a percent is above 100, a box is not a number, or (for percent) the start balance is unknown. It needs the execution layer.
  - A halt is never cleared silently. Raise the Resume ticket by one to continue. Max trades counts open contracts too.
  - It limits losses to a degree only: one contract can still lose its whole stake.

Loss streak pause

- **Kind:** BRINOVS custom, block #15
- **Type id:** `bnv_loss_streak_pause`

- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false block is wanted. Made for an if in "4. Restart trading conditions".
- **What it does:** Pauses new purchases after a number of finished losing rounds in a row. It says true while trading may go on.
- **Inputs:**
  - Fewer than ... losing rounds in a row: a whole number from 1 to 100 (toolbox starts at 3).
  - pause for ... minutes: 0 means until the Resume ticket is raised (starts at 10).
  - Resume ticket: a whole number, 0 or more (starts at 0).
- **Gives back:** True while fewer than the chosen number of rounds have lost in a row and no pause is running. False during a pause.
- **How it works:** The bot counts finished real rounds, newest first, until it finds one that is not a loss. A round that ends at exactly 0 ends the streak. When the streak is reached the pause is saved. After the pause time the count starts again from zero.
- **Example:**



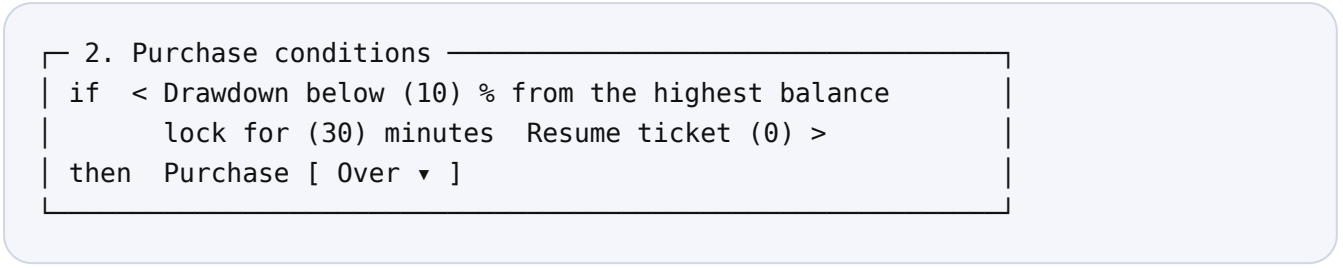
After 3 lost rounds in a row, no new purchase is made for 10 minutes.

- **Good to know:**
  - The pause never changes your stake. It only waits.
  - It is also enforced on every purchase once the block has saved its numbers, even if you delete the block later.
  - A wrong number (not 1 to 100, negative minutes) stops the bot with a message. It does not change your chance of winning.

Drawdown breaker

- **Kind:** BRINOVS custom, block #16
- **Type id:** `bnv_drawdown_breaker`
- **Shape:** True/false block

- **Where it goes:** Anywhere a true/false block is wanted, usually an if in "2. Purchase conditions".
- **What it does:** Locks new purchases when your balance falls too far from its highest point of the session. This fall is called a drawdown. It says true while trading is allowed.
- **Inputs:**
  - Drawdown below ... % from the highest balance: above 0 and at most 100 (toolbox starts at 10).
  - lock for ... minutes: 0 means until the Resume ticket is raised (starts at 30).
  - Resume ticket: a whole number, 0 or more (starts at 0).
- **Gives back:** True while no lock is running and the drawdown is below the limit. False when locked, or when the balance is unknown or older than 60 seconds.
- **How it works:** The bot tracks the highest balance, counting the stakes of open contracts as still yours. When the balance falls by the chosen percent from that peak, it locks. The peak and the lock are saved, so a refresh does not reset them. When the lock time is over, the peak starts again from the current balance.
- **Example:**



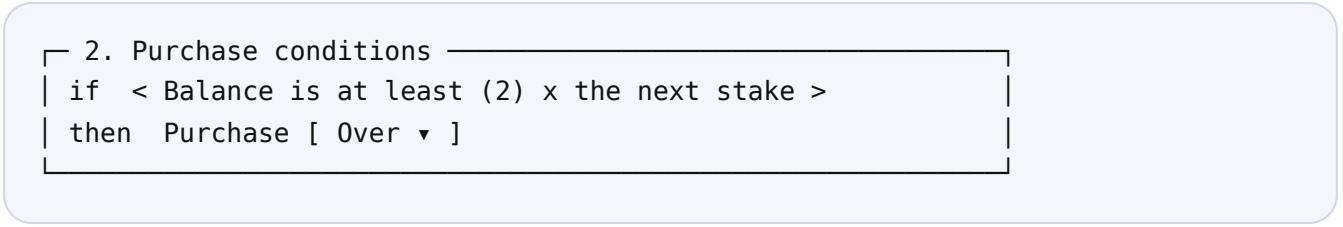
If the balance peaked at 100 and falls to 90, purchases are locked for 30 minutes.

- **Good to know:**
  - It is also enforced on every purchase once the block has saved its numbers, even if you delete the block later.
  - It does not touch open contracts. Its lock ends by time or by a higher Resume ticket.
  - It cannot stop one contract from losing. It does not change your chance of winning.

Balance guard

- **Kind:** BRINOVS custom, block #17
- **Type id:** `bnv_balance_guard`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false block is wanted, usually an if in "2. Purchase conditions".

- **What it does:** Says true when your live balance is at least a chosen number of times the stake of your next purchase. It blocks a purchase when you do not have enough money to cover it with a safety margin.
- **Inputs:**
  - Balance is at least ... x the next stake: a number of 1 or more (toolbox starts at 2).
- **Gives back:** True when the balance is at least that many times the stake (from your Trade options). False when it is lower, or when the balance is unknown or older than 60 seconds.
- **How it works:** The block also saves your number. The same rule is checked on every purchase, using the whole stake of the purchase. If you never use the block, the rule is still on with 1 x the stake. Money already promised to purchases being placed does not count as available. An old balance is asked for again once before the bot decides.
- **Example:**



With a stake of 5, the bot buys only if the balance is at least 10.

- **Good to know:**
  - If the balance cannot be read, the purchase is refused and the bot stops with a message.
  - The rule is on even if you delete the block. The last saved number stays.
  - It protects your balance. It does not change your chance of winning.

Exposure cap

- **Kind:** BRINOVS custom, block #18
- **Type id:** `bnv_exposure_cap`
- **Shape:** Action block (it can have blocks above and below)
- **Where it goes:** In "Run once at start" of "1. Trade parameters". Only inside the Trade parameters block.
- **What it does:** Limits how many contracts may be open at the same time and how much stake may be at risk at the same time. Exposure means the money you have out on open contracts.
- **Inputs:**
  - Exposure cap: max open contracts: a whole number, 0 means no limit (toolbox starts at 3).

  - max total stake at risk: a number, 0 means no limit (starts at 10).
- **How it works:** Before each purchase the bot adds the open contracts, the purchases still being placed and the new purchase, and compares with both caps. Two purchases made at the same moment cannot both use the same room. A purchase over a cap is refused and the bot stops with the reason.
- **Example:**

```
┌─ 1. Trade parameters ──────────────────────────────────────┐
│ Run once at start:                                         │
│   Exposure cap: max open contracts (3) (0 = no limit)      │
│     max total stake at risk (10) (0 = no limit)            │
└────────────────────────────────────────────────────────────┘
```

With 8 already at risk, a new purchase of 3 is refused (11 is above 10). A purchase of 2 goes through.

- **Good to know:**
  - It needs at least one limit above 0, or the bot stops with a message.
  - Multi purchase and Hedge purchase count every contract of the batch.
  - It is a safety limit. It does not change your chance of winning.

Trading window

- **Kind:** BRINOVS custom, block #19
- **Type id:** `bnv_trading_window`
- **Shape:** True/false block
- **Where it goes:** Anywhere a true/false block is wanted, usually an if before Purchase in "2. Purchase conditions".
- **What it does:** Says true while the time of day and the weekday are inside the window you set, in the time zone you choose. Use it to trade only at certain hours.
- **Inputs:**
  - from and to: two times typed as 09:30, 24-hour clock (starts at 09:00 to 17:00).
  - Mon Tue Wed Thu Fri Sat Sun: a tick box for each day (weekdays ticked at the start).
  - time zone: a fixed offset from UTC, from UTC-12 to UTC+14 (starts at UTC).
- **Gives back:** True inside the window, false outside.
- **How it works:** The start time is inside the window, the end time is not. A window that ends before it starts, like 22:00 to 06:00, runs overnight, and its weekday is the day it started. The zone is a

fixed offset, so it does not change with daylight saving.

- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < Inside the trading window [ 09:00 ] to [ 17:00 ]    │
│   Mon ☑ Tue ☑ Wed ☑ Thu ☑ Fri ☑ Sat ☐ Sun ☐               │
│   time zone [ UTC ▾ ] >                                   │
│ then  Purchase [ Over ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys on weekdays between 09:00 and 17:00 UTC only.

- **Good to know:**
  - It only answers the question. It does not stop a purchase by itself, so put it in an if.
  - The bot stops with a message if a time is not like 09:30, the start equals the end, or no day is ticked.
  - It says nothing about the chance to win.

BRINOVS blocks: Entry filters and restart (20 to 24)

Entry filters answer one question: "Is it OK to buy right now?" Each one gives back a simple yes or no (true or false), or a number you can compare. They never place a trade by themselves. You plug them into an `if` block so that Purchase only runs when the filter says yes.

A few words first. A **tick** is one new price from the market. A **candle** is a bar that sums up the price over a fixed time, for example 60 seconds. A **contract** is one trade you buy. A contract **settles** when it ends and the result is known. The **journal** is the log panel of the Bot Builder, where the bot writes what it is doing.

You find these blocks in the BRINOVS menu, in the folder "Entry filters and restart". Most of them go inside "2. Purchase conditions", in the `if` above Purchase. The Cooldown block is made for "4. Restart trading conditions", the part that runs after a trade ends, before the bot trades again. Everything here describes what already happened. None of it can tell you what the next tick or candle will do, and none of it makes a win more likely.

Single entry lock

- **Kind:** BRINOVS custom, block #20
- **Type id:** `bnv_single_entry_lock`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions", above Purchase. Use only one: its settings are saved in one place.
- **What it does:** Stops one lasting condition from turning into a purchase on every tick. It says yes the first time a signal asks and no once the allowed number of purchases was made for that signal. On the block it reads "Signal still free: one purchase per ...".
- **Inputs:**
  - one purchase per: [candle ▾] (default), [tick ▾] or [signal (candle + direction) ▾]. This sets what counts as "one signal".
  - candle size (seconds), default 60; timeout (seconds), default 300. Both are numbers of 1 or more.
  - purchases allowed per signal: a whole number from 1 to 100, default 1
  - release: [when it settles ▾] (default), [at the next candle ▾] or [after the timeout ▾]; contract type: pick the contract the Purchase block buys

- **Gives back:** true while this signal has not been bought yet, false once the allowed number of purchases was made.
- **How it works:** A signal is the market plus the candle it happens in (or the tick, or the candle plus the contract type you buy). The lock itself is a rule on every purchase, so it also works if you delete the block, and it is remembered after a page refresh. It is let go when the contract settles, at the next candle, or after the timeout, as you chose. Every lock also ends after the timeout, so a lost message can never lock the bot for ever. Market time (the time of the last tick) decides which candle you are in, so a wrong clock on your phone does not matter.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < Signal still free: one purchase per [ candle ▾ ] │
│       candle size [ 60 ] seconds                      │
│       purchases allowed per signal [ 1 ]              │
│       release [ when it settles ▾ ]                   │
│       timeout [ 300 ] seconds                         │
│       contract type [ Over ▾ ]                        │
│ then  Purchase [ Over ▾ ]                             │
└───────────────────────────────────────────────────────┘
```

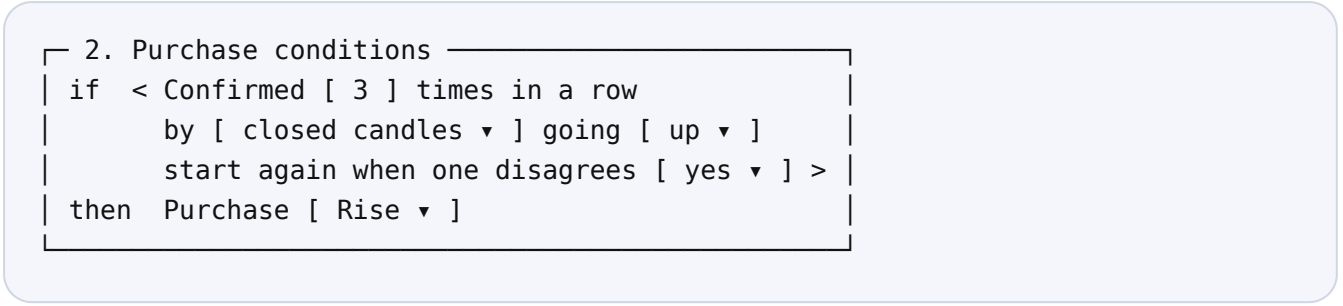
While the market stays in this 60-second candle, the first check says yes and the bot buys once. Every later check in the same candle says no until the contract settles.

- **Good to know:**
  - Refusals (the bot stops with a message): no scope chosen; candle size not a whole number of 1 or more; purchases allowed not a whole number from 1 to 100; no release chosen; timeout under 1; per-signal scope with no contract type; or the BRINOVS execution layer is off.
  - A purchase is refused, with a journal message and nothing bought, when the signal was already bought, when no price has arrived yet, or when the saved setting is damaged.
  - It comes preloaded on a fresh Bot Builder, switched off (disabled). Enable it and drag it into the `if` before Purchase. It says nothing about your chance to win.

Signal confirm

- **Kind:** BRINOVS custom, block #21
- **Type id:** `bnv_signal_confirm`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions". It can be used more than once.

- **What it does:** Says yes only after several different candles or ticks in a row all went the same way. This filters out one-off blips. On the block it reads "Confirmed ... times in a row".
- **Inputs:**
  - Confirmed [3] times in a row: a whole number from 1 to 100, default 3
  - by: [closed candles ▾] (default) or [ticks ▾]
  - going: [up ▾] (default) or [down ▾]
  - start again when one disagrees: [yes ▾] (default) or [no (skip it) ▾]
- **Gives back:** true when the count has reached your number, false before that.
- **How it works:** A closed candle is up when it closed higher than it opened. A tick is up when its price is higher than the tick before. Each candle or tick is counted once, however many times the block is checked. A candle still forming is not counted. With "start again" on, one disagreeing candle or tick (or an unchanged price) sets the count back to 0. With it off, a disagreeing one is skipped and the count stays. The candle size is the one in your Trade parameters, or 60 seconds if none is set.
- **Example:**



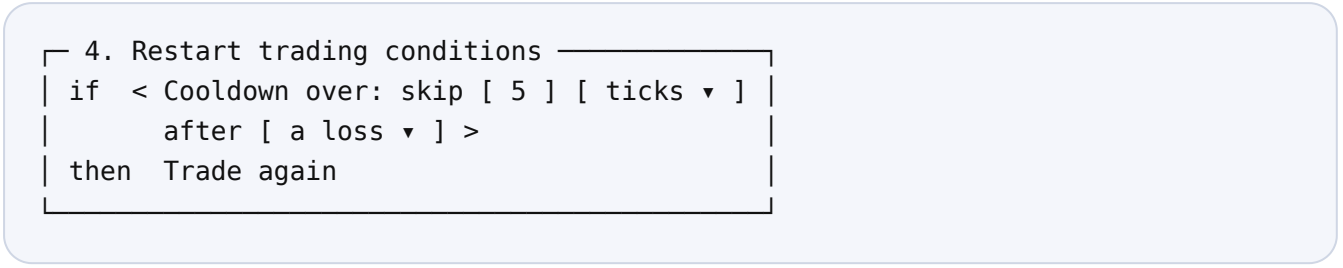
The bot buys only after three closed candles in a row each closed higher than they opened. One down candle starts the count again.

- **Good to know:**
  - Refusals: no candles or ticks chosen; no direction chosen; a number of confirmations that is not a whole number from 1 to 100; candles not available in this bot.
  - If the candles cannot be read, the block says no (never a guess).
  - The count is kept only while the bot is running.
  - It describes what already happened. It does not predict the next candle or tick.

Cooldown

- **Kind:** BRINOVS custom, block #22

- **Type id:** `bnv_cooldown`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "4. Restart trading conditions", before Trade again.
- **What it does:** Makes the bot wait after a trade. It says no while the pause is running and yes once trading is allowed again. On the block it reads "Cooldown over: skip ...".
- **Inputs:**
  - skip: a whole number of 1 or more, default 5
  - unit: [ticks ▾] (default), [candles ▾] or [seconds ▾]
  - after: [a win ▾] (default), [a loss ▾] or [any trade ▾]
- **Gives back:** true when trading is allowed again, false while the pause runs.
- **How it works:** The block looks at the latest finished real trade. If there is none, or if it was not the kind you chose (for example a win when you chose "a loss"), there is no pause and it says yes. Otherwise it waits until you have skipped your number of ticks, candles or seconds since that trade ended. Candles are counted as candle changes, using the candle size of your Trade parameters (60 seconds if none is set), so the first one can come sooner than a full candle. Seconds use the market's own time.
- **Example:**



After a losing trade the bot waits for five new ticks before it trades again. After a win it trades again at once.

- **Good to know:**
  - Refusals: a number that is not a whole number of 1 or more; no unit chosen; no trigger chosen.
  - Pretend (dry run) trades never start a pause.
  - After a page refresh, a pause counted in ticks counts only the ticks seen since the refresh.
  - A pause does not change your chance of winning.

Tick streak

- **Kind:** BRINOVS custom, block #23

- **Type id:** `bnv_tick_streak`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions".
- **What it does:** Says yes when the latest ticks form a run of a chosen kind, at least as long as the number you set. A streak is a description of history, not a prediction.
- **Inputs:**
  - Last [3] ticks are: a whole number from 2 to 100, default 3
  - kind: [rising ticks ▾] (default), [falling ticks ▾], [the same digit ▾] or [the same parity ▾] (all odd or all even last digits)
  - an unchanged price is: [a break ▾] (default) or [ignored ▾]
- **Gives back:** true when the run is at least that long. False when too few ticks are known yet.
- **How it works:** The block counts back from the newest tick. For rising or falling, each step up or each step down counts once, so 3 means three steps in a row. For the same digit or same parity, each tick counts. The digit is read with the market's own number of decimals. An unchanged price either ends a rising or falling run or is skipped, as you chose.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────┐
│ if  < Last [ 4 ] ticks are [ the same parity ▾ ] │
│       an unchanged price is [ a break ▾ ] >      │
│ then  Purchase [ Over ▾ ]                        │
└──────────────────────────────────────────────────┘
```

The bot buys only when the last four ticks all ended in odd digits, or all in even digits.

- **Good to know:**
  - Refusals: a length that is not a whole number from 2 to 100; no kind chosen; no choice for an unchanged price.
  - While the run goes on, the block keeps saying yes. Add a Single entry lock for one purchase per signal.
  - The companion value block "Tick streak length" (next entry) gives the length of the run itself.

Tick streak length

- **Kind:** BRINOVS custom, companion of block #23
- **Type id:** `bnv_tick_streak_length`

- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted, for example in a comparison inside an `if`.
- **What it does:** Tells you how long the current run of ticks is, so you can compare it with a number yourself. A run is a description of history, not a prediction.
- **Inputs:**
  - Current streak of: [rising ticks ▾] (default), [falling ticks ▾], [the same digit ▾] or [the same parity ▾]
  - an unchanged price is: [a break ▾] (default) or [ignored ▾]
- **Gives back:** a whole number: how many in a row, counting back from the latest tick. 0 when there is none.
- **How it works:** It counts exactly like the Tick streak block does: steps up or steps down for rising and falling, ticks for the same digit or the same parity.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < ( Current streak of [ rising ticks ▾ ]          │
│         an unchanged price is [ a break ▾ ] ) ≥ [ 4 ] > │
│ then  Purchase [ Rise ▾ ]                             │
└───────────────────────────────────────────────────────┘
```

The bot buys only when the price has gone up at least four steps in a row.

- **Good to know:**
  - Refusals: no kind chosen; no choice for an unchanged price.
  - It describes the ticks that already happened. It does not predict the next tick.

Tick momentum

- **Kind:** BRINOVS custom, block #24
- **Type id:** `bnv_tick_momentum`
- **Shape:** Value block (gives a number, or one of the words UP, DOWN, NEUTRAL)
- **Where it goes:** Anywhere a number is wanted, for example in a comparison inside an `if` in "2. Purchase conditions".
- **What it does:** Describes the last ticks of the market: how many of the price moves went up or down, or how big a move is. It does not say what comes next.
- **Inputs:**

  - Tick momentum over the last [20] ticks: a whole number from 5 to 200, default 20
  - measure: [% of moves rising ▾] (default), [% of moves falling ▾], [average size of a move ▾] or [direction class (UP / DOWN / NEUTRAL) ▾]
- **Gives back:** a percentage, an average move (in price units), or the word UP, DOWN or NEUTRAL. With too few ticks, or ticks missing inside the window, the number blocks give "not a number" and the class gives UNKNOWN.
- **How it works:** It takes the price moves between the last ticks. The two percentages are shares of all moves in the window; a move with no change counts in neither. The average size drops the biggest and the smallest tenth of the moves, so one jump cannot decide it. The class is UP when rising moves beat falling moves by at least 20 points (for example 60 against 40), DOWN the other way round, otherwise NEUTRAL.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────┐
│ if  < ( Tick momentum over the last [ 20 ] ticks  │
│         measure [ % of moves rising ▾ ] ) ≥ [ 60 ] > │
│ then  Purchase [ Rise ▾ ]                         │
└───────────────────────────────────────────────────┘
```

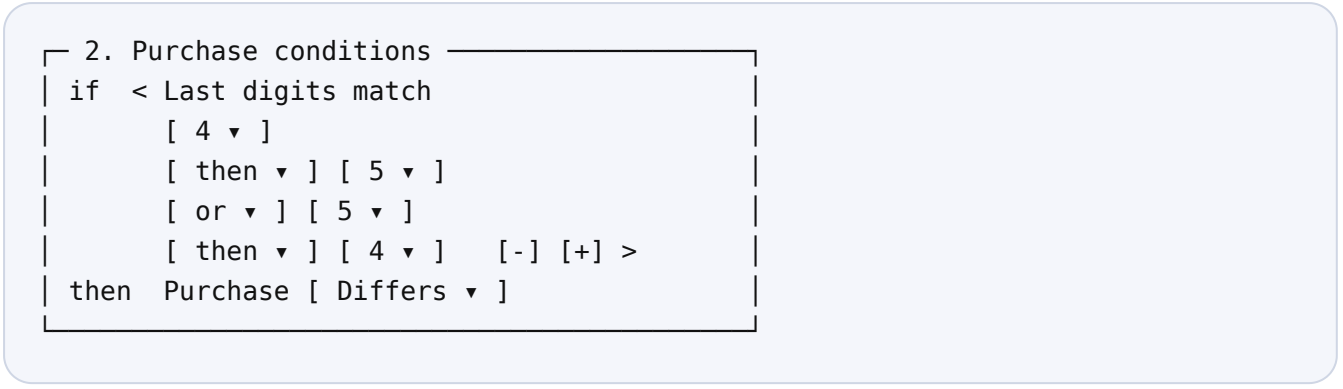
The bot buys when at least 60 percent of the last 20 moves went up. This only counts the past.

- **Good to know:**
  - Refusals: no measure chosen; a window that is not a whole number from 5 to 200.
  - When there is no answer the journal says why: too few ticks, or a gap in the price feed. It is never a guess.
  - Ticks are not trading volume.

Last digits match

- **Kind:** BRINOVS custom, extra block (no catalogue number)
- **Type id:** `bnv_last_digits_match`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions".
- **What it does:** Says yes when the last ticks end in the digits you wrote. You write patterns of last digits, such as "4 then 5". It describes the past only.
- **Inputs:**

  - digits: each one is [any ▾] or 0 to 9. A fresh block shows 4, 5, 5, 4.
  - joins between digits: [then ▾] (the digits come one after the other) or [or ▾] (start another pattern)
  - the - and + buttons remove or add a digit (from 2 up to 12)
- **Gives back:** true when the end of the tick list fits one of your patterns. False when too few ticks are known yet.
- **How it works:** Read from left to right. The oldest digit is first, the last digit is the newest tick. "then" binds tighter than "or", so "4 then 5 or 5 then 4" means the last two digits are 4, 5 or they are 5, 4. "any" matches every digit. Digits are read with the market's own number of decimals.
- **Example:**



The bot buys when the last two digits are 4 then 5, or 5 then 4.

- **Good to know:**
  - Refusals: no pattern written; a pattern with more than 12 digits; a digit that is not 0 to 9 or "any".
  - While the latest digits stay the same it keeps saying yes. Add a Single entry lock if you want one purchase per signal.

BRINOVS blocks: Analysis (25 to 33)

The Analysis blocks look at the past of the market and measure it. They count digits, measure how big the price moves are, read a trend from moving averages, spot named candle shapes and count how many of your conditions are true. They give back numbers, lists, words or yes and no.

Please read this once. Every one of these blocks describes what already happened. None of them predicts the next tick, the next candle or the result of your trade, and none of them makes a win more likely. They are tools for building your own rules, and you should test any rule on a demo account first.

A few words first. A **tick** is one new price from the market. A **candle** is a bar that sums up the price over a fixed time, with an open, a high, a low and a close. The **last digit** of a price is its final figure, read with the market's own number of decimals, so 876.50 has the last digit 0. A **window** is how many of the latest ticks or candles a block looks at.

You find these blocks in the BRINOVS menu, in the folder "Analysis". They can be plugged in anywhere a number, a word or a true/false is wanted, most often inside an `if` in "2. Purchase conditions". If a block cannot answer (too little data, a gap in the data, or the data cannot be read), it does not guess. Number blocks give "not a number", true/false blocks give false, and word blocks say NOT_READY or UNKNOWN. The candle blocks use only closed candles, never the one still forming.

Digit frequency

- **Kind:** BRINOVS custom, block #25
- **Type id:** `bnv_digit_frequency`
- **Shape:** Value block (gives a number, a list, or a true/false, depending on your choice)
- **Where it goes:** Anywhere a value is wanted.
- **What it does:** Counts how often each last digit (0 to 9) showed up in the latest ticks of the market the bot trades. It replaces the ten-variable counting loop people used to build by hand.
- **Inputs:**
  - Digit frequency over the last [1000] ticks: a whole number from 1 to 1000, default 1000. A bigger number is cut to 1000 and the journal says so.
  - give me: [the % of one digit ▾] (default), [the list of 10 percentages (digit 0 to 9) ▾], [the number of ticks counted ▾] or [true when the window is full ▾]

  - digit: a number from 0 to 9, default 0. It is used only for "the % of one digit".
- **Gives back:** a percentage, a list of ten percentages (digit 0 first), a count of ticks, or true/false.
- **How it works:** The bot takes the tick history it keeps for the traded market (at most the latest 1000 ticks) and counts the last digits. Percentages are shares of the ticks counted, with two decimals. Until the window is full, the percentages cover the ticks known so far. "The number of ticks counted" tells you how many, and "true when the window is full" is the warm-up flag.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────┐
│ if  < ( Digit frequency over the last [ 1000 ] ticks │
│         give me [ the % of one digit ▾ ]          │
│         digit [ 7 ] ) < [ 8 ] >                   │
│ then  Purchase [ Over ▾ ]                         │
└──────────────────────────────────────────────────┘
```

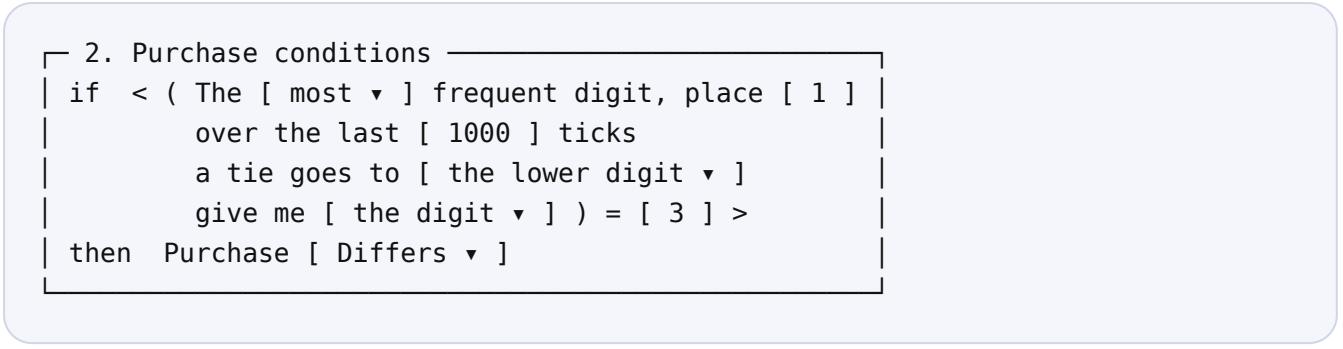
The bot buys only when the digit 7 made up less than 8 percent of the last 1000 ticks. This is a count of the past.

- **Good to know:**
  - Refusals: a window that is not a whole number of 1 or more; no choice of what to give back; a digit that is not a whole number from 0 to 9; tick history not available in this bot.
  - If the history or the market's decimals cannot be read, there is no answer (not a number, 0 or false) and the journal says why.
  - It is a descriptive statistic. On synthetic indices the past frequency of a digit may not predict the next digit.

Digit extremes

- **Kind:** BRINOVS custom, block #26
- **Type id:** `bnv_digit_extremes`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted.
- **What it does:** Finds the most frequent or least frequent last digit in the latest ticks, or the second, third and so on. It can give you the digit or its percentage.
- **Inputs:**
  - The [most ▾] frequent digit, place [1]: most (default) or least; the place is a whole number from 1 to 10, default 1

  - over the last [1000] ticks: a whole number from 1 to 1000, default 1000
  - a tie goes to: [the lower digit ▾] (default) or [the digit seen most recently ▾]
  - give me: [the digit ▾] (default) or [its % ▾]
- **Gives back:** the digit (0 to 9) or its percentage. "Not a number" until the window is full.
- **How it works:** It counts the last digits like the Digit frequency block and ranks the ten digits. Place 1 is the top of the ranking, place 2 the next one, and so on up to 10. When two digits have the same count, your tie rule decides.
- **Example:**



The bot buys only when the most common last digit of the last 1000 ticks is 3.

- **Good to know:**
  - Refusals: no most or least chosen; a place that is not a whole number from 1 to 10; no tie rule; no choice of what to give back; a window that is not a whole number of 1 or more.
  - Until the window is full there is no answer. The journal says how many ticks are known.
  - It is a descriptive statistic. A digit being "hot" in the past does not mean it will come again.

Digit transition

- **Kind:** BRINOVS custom, block #27
- **Type id:** `bnv_digit_transition`
- **Shape:** Value block (gives a number or a true/false, depending on your choice)
- **Where it goes:** Anywhere a value is wanted.
- **What it does:** Tells you how often one last digit was followed by another one in the latest ticks. For example: of all the times a 4 appeared, in what share of cases did the next tick end in a 7?
- **Inputs:**
  - How often digit [0] is followed by digit [1]: two numbers from 0 to 9, defaults 0 and 1
  - over the last [1000] ticks: a whole number from 1 to 1000, default 1000

- give me: [the % ▾] (default), [times the first digit was counted ▾], [true when that is under 30 (a small sample) ▾] or [true when the window is full ▾]

- **Gives back:** a percentage, a count, or true/false.
- **How it works:** The bot looks at every time the first digit appeared and had a tick after it. The percentage is the share of those times where the next tick ended in the second digit. If the first digit never had a tick after it, there is no answer.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < ( How often digit [ 4 ] is followed by digit [ 7 ] │
│         over the last [ 1000 ] ticks                  │
│         give me [ the % ▾ ] ) ≥ [ 12 ] >              │
│ then  Purchase [ Matches ▾ ]                          │
└───────────────────────────────────────────────────────┘
```

The bot buys only when, in the last 1000 ticks, a 4 was followed by a 7 in at least 12 percent of the cases.

- **Good to know:**
  - Refusals: a first or second digit that is not a whole number from 0 to 9; no choice of what to give back; a window that is not a whole number of 1 or more.
  - Under 30 times counted is a small sample. The percentage is then very uncertain, and the journal warns once.
  - Until the window is full the percentage is "not a number".
  - It is a descriptive statistic and does not predict the next digit.

Volatility gate

- **Kind:** BRINOVS custom, block #28
- **Type id:** `bnv_volatility_gate`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions".
- **What it does:** Says yes when the market's volatility is inside the range you set. Volatility means how much the price moves around. It says nothing about which way the price will go.
- **Inputs:**

  - Volatility is normal: measure: [ATR (average true range) ▾] (default), [standard deviation of returns ▾] or [average tick move ▾]
  - over the last [14] candles or ticks: a whole number from 2 to 200, default 14
  - normal range from [0] to [1]: two numbers of 0 or more, minimum first. The defaults are only placeholders.
  - candle size: [1 minute ▾] (default) and others up to 1 day. It is not used by the average tick move.
- **Gives back:** true when the value is from your minimum to your maximum (both included). False when it is below, above, or not known.
- **How it works:** ATR is the average size of a candle, in price units. The standard deviation of returns (how much the closes jump from candle to candle) and the average tick move are in percent. Only closed candles are used. A missing candle inside the data, or too few candles, gives no answer.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < Volatility is normal: measure [ average tick move ▾ ] │
│       over the last [ 50 ] candles or ticks               │
│       normal range from [ 0.01 ] to [ 0.05 ]              │
│       candle size [ 1 minute ▾ ]                          │
│       (not used by the tick move) >                       │
│ then  Purchase [ Rise ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys only while the average tick move of the last 50 ticks is between 0.01 and 0.05 percent.

- **Good to know:**
  - Refusals: no method chosen; a lookback that is not a whole number from 2 to 200; a minimum or maximum below 0; a minimum above the maximum; a candle size not from the list.
  - The right range is different for every market and candle size. Set it for yours.
  - It comes with a companion value block, "Volatility value" (`bnv_volatility_value`). It has the same settings and gives back the number itself, or the state LOW, NORMAL, HIGH or NOT_READY.

Volatility value

- **Kind:** BRINOVS custom, companion of block #28
- **Type id:** `bnv_volatility_value`

- **Shape:** Value block (gives a number or a word)
- **Where it goes:** Anywhere a value is wanted. Use the same settings as your Volatility gate.
- **What it does:** Gives you the number that the Volatility gate looks at, or its state. It is handy when you want to compare the volatility yourself.
- **Inputs:**
  - measure, lookback, normal range and candle size: the same as in the Volatility gate, with the same defaults (ATR, 14, 0 to 1, 1 minute)
  - give me: [the value ▾] (default) or [the state (LOW / NORMAL / HIGH) ▾]
- **Gives back:** the volatility number, or one of LOW (under your minimum), NORMAL, HIGH (over your maximum) or NOT_READY (not known).
- **How it works:** It works out the same value as the gate, with the same rules for closed candles and missing data, and then returns either the value or the state.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────┐
│ if  < ( Volatility value: measure                 │
│         [ ATR (average true range) ▾ ]            │
│         over the last [ 14 ] candles or ticks     │
│         normal range from [ 0.2 ] to [ 1.5 ]      │
│         candle size [ 5 minutes ▾ ]               │
│         (not used by the tick move)               │
│         give me [ the state ▾ ] ) = [ NORMAL ] >  │
│ then  Purchase [ Over ▾ ]                         │
└───────────────────────────────────────────────────┘
```

The bot buys only when the ATR of the 5-minute candles is in the normal range.

- **Good to know:**
  - It has the same refusals as the Volatility gate, plus one more: no choice of what to give back.
  - Volatility says nothing about direction.

Multi-timeframe bias

- **Kind:** BRINOVS custom, block #29
- **Type id:** `bnv_multi_timeframe_bias`
- **Shape:** Value block (gives a word: UP, DOWN, NEUTRAL or NOT_READY)
- **Where it goes:** Anywhere a value is wanted, for example in a comparison inside an `if`.

- **What it does:** Reads the usual "bias" of a candle size by comparing a fast moving average with a slow one. It is meant for a bigger candle size than the one you trade with, to give a wider picture of the past.
- **Inputs:**
  - Bias on the [1 minute ▾] candles: a size from 1 minute up to 1 day, default 1 minute
  - fast average [9] slow average [21]: whole numbers from 2 to 200, defaults 9 and 21; fast must be smaller than slow
  - neutral band [0] %: a number from 0 to 10, default 0
- **Gives back:** UP when the fast average is above the slow one by more than the neutral band, DOWN when it is below by more than the band, otherwise NEUTRAL. NOT_READY when it cannot tell.
- **How it works:** It uses two exponential moving averages (averages that give more weight to the newest candles) of the closed candles, built the same way as Deriv's own EMA block. The band is a share of the slow average. It needs twice the slow period in closed candles; with fewer, or with a missing candle, it says NOT_READY and never a quiet NEUTRAL.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────┐
│ if  < ( Bias on the [ 15 minutes ▾ ] candles   │
│         fast average [ 9 ] slow average [ 21 ] │
│         neutral band [ 0.05 ] % ) = [ UP ] >   │
│ then  Purchase [ Rise ▾ ]                      │
└────────────────────────────────────────────────┘
```

The bot buys only when the 9-candle average of the 15-minute candles is clearly above the 21-candle average.

- **Good to know:**
  - Refusals: a candle size not from the list; a fast or slow period that is not a whole number from 2 to 200; fast not smaller than slow; a band outside 0 to 10.
  - A moving-average bias lags the price. Test your strategy with and without it before trusting it.
  - It describes the past. It does not predict.

ATR

- **Kind:** BRINOVS custom, block #30 (Indicator pack, 1 of 4)
- **Type id:** `bnv_atr`

- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted.
- **What it does:** Gives the average true range of the closed candles: the average size of a candle, counting gaps between candles too. It is in price units. It says nothing about direction.
- **Inputs:**
  - ATR (average true range) on the [1 minute ▾] candles: a size from 1 minute up to 1 day, default 1 minute
  - period [14]: a whole number from 2 to 200, default 14
- **Gives back:** a number in price units, or "not a number" when there are too few closed candles or a candle is missing.
- **How it works:** The "true range" of a candle is the biggest of its high minus low, and its distance from the previous close at the high and at the low. The first ATR is the plain average of the first period's true ranges, then it is smoothed (Wilder's method). It needs the period plus one closed candles.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────┐
│ if  < ( ATR (average true range) on the   │
│         [ 5 minutes ▾ ] candles           │
│         period [ 14 ] ) < [ 2.5 ] >       │
│ then  Purchase [ Rise ▾ ]                 │
└───────────────────────────────────────────┘
```

The bot buys only when the average 5-minute candle is smaller than 2.5 price units.

- **Good to know:**
  - Refusals: a candle size not from the list; a period that is not a whole number from 2 to 200.
  - It uses closed candles only. It describes candles that already closed and does not predict the next one.
  - The same measure is available as a choice inside the Volatility gate.

Rate of change

- **Kind:** BRINOVS custom, block #30 (Indicator pack, 2 of 4)
- **Type id:** `bnv_roc`
- **Shape:** Value block (gives a number)

- **Where it goes:** Anywhere a number is wanted.
- **What it does:** Gives how much the closing price changed, in percent, compared with the close some candles ago.
- **Inputs:**
  - Rate of change on the [1 minute ▾] candles: a size from 1 minute up to 1 day, default 1 minute
  - period [10]: a whole number from 1 to 200, default 10
- **Gives back:** a percentage (it can be negative), or "not a number" when there are too few closed candles or a candle is missing.
- **How it works:** The formula is (latest close divided by the close one period ago, minus 1) times 100. It needs the period plus one closed candles.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────┐
│ if  < ( Rate of change on the [ 1 minute ▾ ] candles │
│         period [ 10 ] ) > [ 0 ] >                 │
│ then  Purchase [ Rise ▾ ]                         │
└───────────────────────────────────────────────────┘
```

The bot buys only when the close is higher than it was 10 candles ago.

- **Good to know:**
  - Refusals: a candle size not from the list; a period that is not a whole number from 1 to 200.
  - It describes the past. It does not predict the next candle.

Stochastic

- **Kind:** BRINOVS custom, block #30 (Indicator pack, 3 of 4)
- **Type id:** `bnv_stochastic`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted.
- **What it does:** Gives the stochastic oscillator of the closed candles, from 0 to 100. It shows where the latest close sits inside the recent high-to-low range.
- **Inputs:**
  - Stochastic on the [1 minute ▾] candles [%K ▾]: a size from 1 minute up to 1 day; %K (default) or %D
  - period [14]: a whole number from 2 to 200, default 14

  - smooth %K over [1] %D over [3]: smoothing of %K from 1 to 50, default 1 (1 means not smoothed); the %D period from 1 to 50, default 3
- **Gives back:** a number from 0 to 100, or "not a number" when there are too few closed candles, a candle is missing, or the highest high equals the lowest low.
- **How it works:** %K is 100 times (close minus lowest low) divided by (highest high minus lowest low) over the period, averaged over the smoothing. %D is the average of the last few %K values.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ( Stochastic on the [ 5 minutes ▾ ] candles [ %K ▾ ] │
│         period [ 14 ]                                     │
│         smooth %K over [ 3 ] %D over [ 3 ] ) < [ 20 ] >   │
│ then  Purchase [ Rise ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys only when %K of the 5-minute candles is under 20, which means the close is near the bottom of its recent range.

- **Good to know:**
  - Refusals: a candle size not from the list; a period, smoothing or %D period outside its range; no choice of %K or %D.
  - Closed candles only. It describes the past and does not predict.

ADX

- **Kind:** BRINOVS custom, block #30 (Indicator pack, 4 of 4)
- **Type id:** `bnv_adx`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted.
- **What it does:** Gives the average directional index (ADX), from 0 to 100. It shows how strong a trend has been, not which way it goes. +DI and -DI show the push up and the push down.
- **Inputs:**
  - ADX on the [1 minute ▾] candles [ADX (trend strength) ▾]: a size from 1 minute up to 1 day; ADX (default), +DI or -DI
  - period [14]: a whole number from 2 to 100, default 14

- **Gives back:** a number from 0 to 100, or "not a number" when there are too few closed candles or a candle is missing.
- **How it works:** It uses Wilder's method on the closed candles. It needs twice the period in closed candles. A high ADX means a strong move in one direction or the other, and a low ADX means little trend.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────┐
│ if  < ( ADX on the [ 15 minutes ▾ ] candles │
│         [ ADX (trend strength) ▾ ]          │
│         period [ 14 ] ) > [ 25 ] >          │
│ then  Purchase [ Rise ▾ ]                   │
└─────────────────────────────────────────────┘
```

The bot buys only when the 15-minute candles have shown a strong trend (ADX above 25). It does not say which way the price will go next.

- **Good to know:**
  - Refusals: a candle size not from the list; a period that is not a whole number from 2 to 100; no choice of ADX, +DI or -DI.
  - It describes candles that already closed and does not predict.

Support / resistance zone

- **Kind:** BRINOVS custom, block #31
- **Type id:** `bnv_sr_zone`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions".
- **What it does:** Says yes when the latest price is near a support or resistance zone found in the closed candles. Support is a price area where the price turned up before. Resistance is an area where it turned down before.
- **Inputs:**
  - Price is near a [support ▾] zone on the [1 minute ▾] candles: support (default), resistance or support or resistance; candle size default 1 minute
  - look back [100] closed candles (30 to 500); swing width [3] candles each side (1 to 10)
  - zone width [0.5] x ATR (0.1 to 3); fewest touches [2] (2 to 10)

  - near means within [0.1] % of the price (0.01 to 5); work the zones out again every [5] new candles (1 to 100)
- **Gives back:** true when the price is near a zone of the kind you chose, false otherwise or when there is no answer.
- **How it works:** A swing high is a candle whose high is above the candles around it (a swing low is the mirror). Swing points that sit close together, within the zone width (a share of the ATR of 14), form a zone, and it counts only with enough touches. The newest candles cannot be swing points yet, so a zone always lags a little. The zones are worked out again only every few new candles, to keep the browser light.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < Price is near a [ support ▾ ] zone on the       │
│       [ 5 minutes ▾ ] candles                         │
│       look back [ 100 ] closed candles                │
│       swing width [ 3 ] candles each side             │
│       zone width [ 0.5 ] x ATR                        │
│       fewest touches [ 2 ]                            │
│       near means within [ 0.1 ] % of the price        │
│       work the zones out again every [ 5 ] new candles > │
└───────────────────────────────────────────────────────┘
```

The bot buys only when the price sits within 0.1 percent of a support zone.

- **Good to know:**
  - Refusals: a candle size not from the list; any of the numbers outside the ranges above; no kind of zone chosen. A zone is where the price turned before. It says nothing about the next turn, and the price can go straight through it.
  - The companion value block "Zone touches" (`bnv_sr_zone_touches`) has the same settings and gives the number of touches of the zone the price is near.

Zone touches

- **Kind:** BRINOVS custom, companion of block #31
- **Type id:** `bnv_sr_zone_touches`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. Use the same settings as your Support / resistance zone block.

- **What it does:** Tells you how many times the price turned at the zone it is near. The more touches, the more often the price turned there.
- **Inputs:** The same seven settings as the Support / resistance zone block, with the same defaults and ranges. Only the first line of the label differs: "Touches of the [support ▾] zone the price is near, on the [1 minute ▾] candles".
- **Gives back:** a whole number of touches. 0 when the price is not near a zone. "Not a number" when there is no answer yet.
- **How it works:** It finds the zones like the Support / resistance zone block. If several zones are near, it uses the nearest one, and on a tie the one with more touches.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ( Touches of the [ support ▾ ] zone the price is near, │
│         on the [ 5 minutes ▾ ] candles                    │
│         look back [ 100 ] closed candles                  │
│         swing width [ 3 ] candles each side               │
│         zone width [ 0.5 ] x ATR                          │
│         fewest touches [ 2 ]                              │
│         near means within [ 0.1 ] % of the price          │
│         work the zones out again every [ 5 ] new candles )   │
│       ≥ [ 3 ] >                                          │
└───────────────────────────────────────────────────────────┘
```

The bot buys only when the nearby support zone has been touched at least three times.

- **Good to know:**
  - It has the same refusals as the Support / resistance zone block.
  - Many touches do not mean the zone will hold. It describes the past only.

Candle pattern

- **Kind:** BRINOVS custom, block #32
- **Type id:** `bnv_candle_pattern`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions".
- **What it does:** Says yes when the latest closed candle completes a named shape. A pattern is only a shape. Studies of whether these shapes predict anything are mixed, so a pattern does not make a win more likely.

- **Inputs:**
  - Latest closed candle is a [bullish engulfing ▾]: bullish engulfing (default), bearish engulfing, hammer, shooting star, doji, inside bar, any bullish (engulfing, hammer), any bearish (engulfing, shooting star) or any of the six
  - on the [1 minute ▾] candles: default 1 minute
  - only after a fall (UP shapes) or a rise (DOWN shapes): [yes ▾] (default) or [no ▾]
- **Gives back:** true when the shape is completed by the latest closed candle.
- **How it works:** Engulfing: the body covers the body before it and is bigger. Hammer: the body is over 10 percent of the range, the lower shadow is at least 2 times the body, the upper shadow is at most 10 percent of the range. Shooting star: the mirror of a hammer. Doji: the body is at most 10 percent of the range. Inside bar: the high is lower and the low is higher than the candle before. With "only after a fall or rise" on, the UP shapes need the close before them to be lower than 3 candles earlier, and the DOWN shapes higher.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────────┐
│ if  < Latest closed candle is a [ hammer ▾ ] on the           │
│       [ 5 minutes ▾ ] candles                                 │
│       only after a fall (UP shapes) or a rise (DOWN shapes)   │
│       [ yes ▾ ] >                                             │
│ then  Purchase [ Rise ▾ ]                                     │
└───────────────────────────────────────────────────────────────┘
```

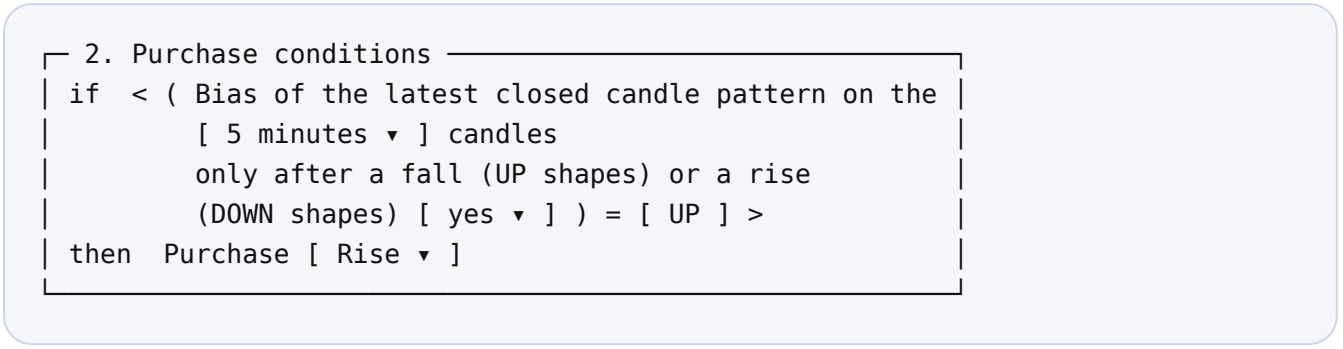
The bot buys only when the last closed 5-minute candle is a hammer that came after a fall.

- **Good to know:**
  - Refusals: a candle size not from the list; no choice for "only after a fall or rise"; a pattern not from the list.
  - With "only after" on it needs 6 closed candles, otherwise 2. With too few candles, or a missing candle, it says no.
  - The companion value block "Candle pattern bias" (`bnv_candle_pattern_bias`) gives the word UP, DOWN, NEUTRAL, NONE or NOT_READY. The word is only the traditional name of the shape. It does not say the price will go that way.

Candle pattern bias

- **Kind:** BRINOVS custom, companion of block #32

- **Type id:** `bnv_candle_pattern_bias`
- **Shape:** Value block (gives a word)
- **Where it goes:** Anywhere a value is wanted.
- **What it does:** Gives the traditional name of the shape that the latest closed candle completes, as one word.
- **Inputs:**
  - Bias of the latest closed candle pattern on the [1 minute ▾] candles: default 1 minute
  - only after a fall (UP shapes) or a rise (DOWN shapes): [yes ▾] (default) or [no ▾]
- **Gives back:** UP (bullish engulfing, hammer), DOWN (bearish engulfing, shooting star) or NEUTRAL (doji, inside bar). If an UP and a DOWN shape both fit, NEUTRAL. NONE when no shape is completed. NOT_READY when there are too few candles or a candle is missing.
- **How it works:** It checks the six shapes of the Candle pattern block on the latest closed candle and turns the result into one word.
- **Example:**



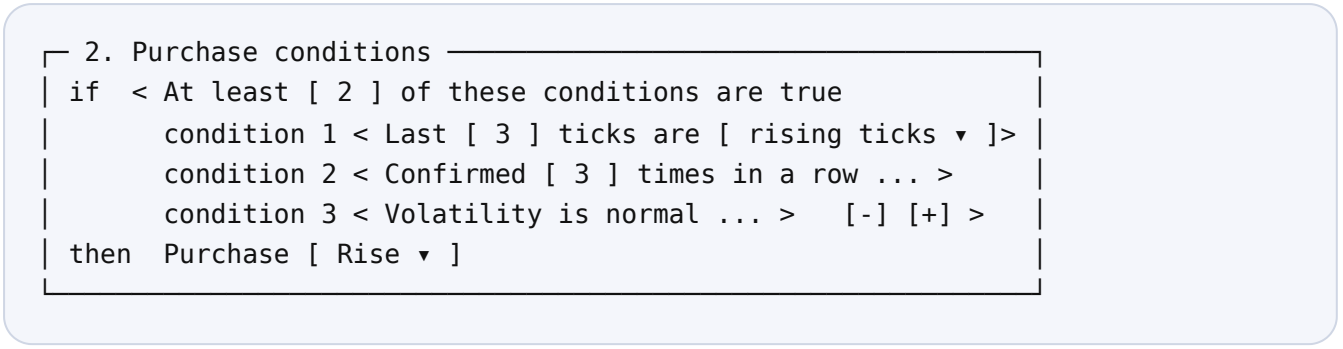
The bot buys only when the last closed candle completes a shape that is traditionally read as UP.

- **Good to know:**
  - Refusals: a candle size not from the list; no choice for "only after a fall or rise".
  - UP and DOWN here are names of shapes. Mixed studies show they do not reliably say what comes next.

Confluence score

- **Kind:** BRINOVS custom, block #33
- **Type id:** `bnv_confluence_score`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions". Its sockets take other true/false blocks.

- **What it does:** Says yes when at least N of the conditions you plug in are true. It is a plain "N of M" count, with no hidden weights and no score out of 100.
- **Inputs:**
  - At least [2] of these conditions are true: a whole number from 1 up to the number of sockets, default 2
  - condition 1, condition 2, condition 3: sockets for true/false blocks. A fresh block has 3.
  - the - and + buttons remove or add a socket (from 2 up to 10)
- **Gives back:** true when at least your number of conditions are true.
- **How it works:** The bot looks at every socket every time, from left to right. Only a value that is exactly true counts. An empty socket, false, "not a number" or text counts as not true. Every condition counts the same.
- **Example:**



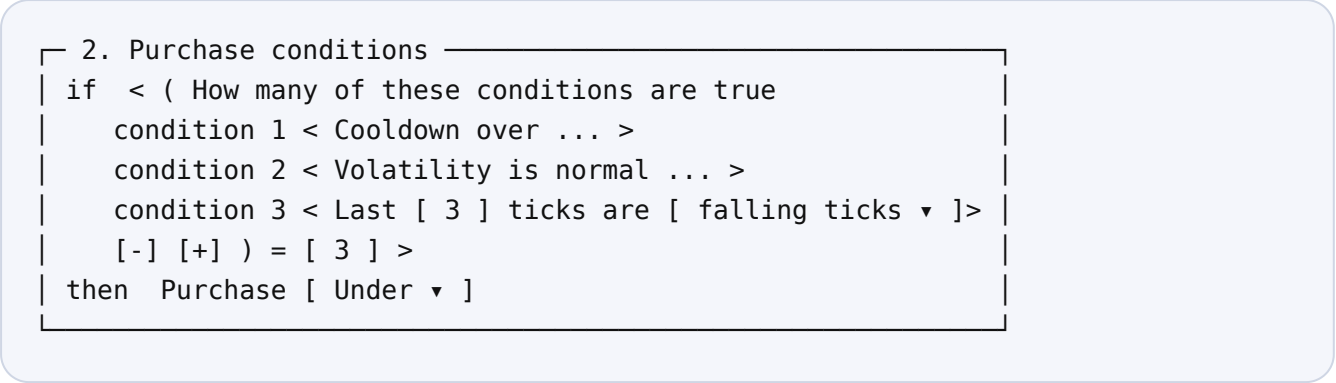
The bot buys when at least two of the three conditions are true at the same moment. (The plugged-in blocks are shortened here.)

- **Good to know:**
  - Refusals: fewer than 2 or more than 10 conditions; a number needed that is not a whole number from 1 to the number of sockets.
  - Agreement does not make a win more likely. Many conditions are made from the same prices.
  - The companion value block "Confluence count" (`bnv_confluence_count`) gives the number of true conditions (0 up to the number of sockets) instead of a yes or no.

Confluence count

- **Kind:** BRINOVS custom, companion of block #33
- **Type id:** `bnv_confluence_count`
- **Shape:** Value block (gives a number)
- **Where it goes:** Anywhere a number is wanted. Its sockets take true/false blocks.

- **What it does:** Counts how many of the conditions you plug in are true right now. Use it in a comparison when you want more than a yes or no.
- **Inputs:**
  - condition 1, condition 2, condition 3: sockets for true/false blocks. A fresh block has 3.
  - the - and + buttons remove or add a socket (from 2 up to 10)
- **Gives back:** a whole number from 0 up to the number of sockets.
- **How it works:** It works like Confluence score, but returns the count. Only an exactly true value counts, every socket is looked at every time, and an empty socket counts as not true.
- **Example:**



The bot buys only when all three conditions are true. (The plugged-in blocks are shortened here.)

- **Good to know:**
  - Refusals: fewer than 2 or more than 10 conditions.
  - It only counts. It does not say that more agreement means a better chance to win.

BRINOVS blocks: Sell conditions and market selection (34 to 37)

This part has two small folders of BRINOVS blocks. The first, "Sell conditions", is about closing a contract before it ends. The second, "Market selection", is about which market (symbol) the bot trades and which markets it is allowed to use.

A few words first. A **contract** is one trade you buy. It **settles** when it ends and the result is known. To **sell** a contract means to close it early at the **market price**, which is the price the market offers at that moment, not a price you choose. A **round** is one finished trade (a hedge, which buys two opposite contracts together, counts as one round). A **market** or **symbol** is the thing you trade, for example the Volatility 100 Index, written R_100. The **journal** is the log panel of the Bot Builder, where the bot writes what it is doing.

Important about selling early: many contract types cannot be sold before they end. Digit contracts are an example. Deriv decides, and these blocks always ask Deriv first. When a contract cannot be sold, the bot leaves it to settle by itself and says so in the journal.

You find these blocks in the BRINOVS menu, in the folders "Sell conditions" and "Market selection". The sell blocks work while a contract is open, in "3. Sell conditions". The market blocks run between trades or before a purchase. Nothing here makes a win more likely.

Early exit

- **Kind:** BRINOVS custom, block #34
- **Type id:** `bnv_early_exit`
- **Shape:** Action block
- **Where it goes:** Only inside "3. Sell conditions". Outside it, the block is greyed out (disabled).
- **What it does:** Watches your open contract and asks to sell it at the market price when its profit reaches a target, when its loss reaches a limit, or when a number of ticks have passed since the purchase. You can use any one of these or all three.
- **Inputs:**
  - sell when the profit reaches: a number of 0 or more, default 2 (0 = not used)
  - or the loss reaches: a number of 0 or more, default 1 (0 = not used)
  - or [0] ticks have passed since the purchase: a whole number of 0 or more, default 0 (0 = not used)
  - the condition must hold [1] ticks in a row: a whole number from 1 to 1000, default 1

  - look at: [each contract ▾] (default) or [the whole batch together ▾]
- **How it works:** On every check the block works out the profit of the contract. "Must hold K ticks" means the condition has to be true at K different ticks in a row, and the count starts again if it stops being true. With "the whole batch together" it adds the profit of all contracts bought together (a hedge), settled ones included, and sells the open ones. It sells only when Deriv says the contract can be sold. A sell is never sent twice at once, and it is tried at most 3 times, 3 seconds apart. Ticks are counted by this run, so a contract that was already open when the bot started may be counted short.
- **Example:**

```
┌─ 3. Sell conditions ──────────────────────────────┐
│ Early exit: sell when the profit reaches [ 3 ]  │
│   or the loss reaches [ 2 ]                     │
│   or [ 0 ] ticks have passed since the purchase │
│   the condition must hold [ 2 ] ticks in a row  │
│   look at [ each contract ▾ ]                   │
└─────────────────────────────────────────────────┘
```

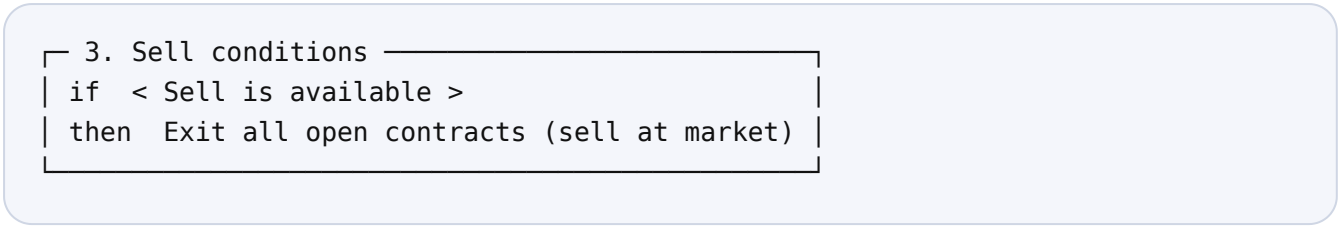
When the profit of the contract stays at 3 or more for two ticks in a row, the bot asks Deriv to sell it at the market price. The same happens if the loss stays at 2 or more.

- **Good to know:**
  - Refusals (the bot stops with a message): a target or maximum loss that is empty or below 0; "after N ticks" that is not a whole number of 0 or more; "must hold" that is not a whole number from 1 to 1000; no scope chosen; all three conditions at 0 ("set at least one condition"); the BRINOVS execution layer is off.
  - The sale is at the market price at that moment. A maximum loss here is NOT a guaranteed price, and the real loss can be bigger than the number you wrote.
  - If the contract cannot be sold, the journal says so once and the contract settles by itself. After 3 failed tries the bot gives up on that contract and says so.
  - The profit and loss numbers are in the money of your account.

Exit all contracts

- **Kind:** BRINOVS custom, block #35
- **Type id:** `bnv_exit_all`
- **Shape:** Action block

- **Where it goes:** Anywhere an action block fits, in the middle of a stack. It is not limited to one root block.
- **What it does:** The panic button. It asks to sell every open contract at the market price, one by one, and writes in the journal how many were sold and which ones could not be. On the block it reads "Exit all open contracts (sell at market)".
- **Inputs:** None. The block has no fields.
- **How it works:** For each open contract the bot first checks that Deriv says it can be sold. Contracts that cannot be sold (many types, digits for example) are left to settle by themselves, and the journal gives the reason for each. A contract is never sold twice. The price you get is the market price at that moment, not a price you choose.
- **Example:**



When the contract can be sold, the bot sells all its open contracts it is allowed to sell. The journal then says, for example, "sold 1 of 1 open contract".

- **Good to know:**
  - Refusal: the bot stops if the BRINOVS execution layer is off.
  - If there are no open contracts, the journal says there is nothing to sell. If Deriv does not answer, the journal says nothing was assumed and tells you to check your open contracts in Deriv.
  - If Deriv asks the bot to slow down, press the button again in a moment. If an answer is lost, check that contract in Deriv before doing anything else.
  - It does not undo losses. It only closes contracts at the market price.

Market switcher

- **Kind:** BRINOVS custom, block #36
- **Type id:** `bnv_market_switcher`
- **Shape:** Action block
- **Where it goes:** Only inside "4. Restart trading conditions", before Trade again.
- **What it does:** Changes the market the bot trades, between trades, and only to a market from your own list. It can rotate through the list, move on after losses, or pick the market with the best

score. With switching OFF it is a one-time picker.

- **Inputs:**
  - switching: [ON (switch between my markets) ▼] (default) or [OFF (pick one market, once) ▼]. With OFF only the Market, Submarket and Symbol lists of "1. Trade parameters" show.
  - my markets: names separated by commas, 2 to 30 different ones (the default is R_10, R_25, R_50, R_75, R_100 and the 1HZ markets); mode: [rotate every N rounds ▼] (default), [switch after N losing rounds in a row ▼] or [pick the best score every N rounds ▼]; N: a whole number from 1 to 1000, default 5
  - best score means: [highest volatility ▼] (default), [lowest volatility ▼], [highest custom number ▼] or [not picked recently ▼]; custom numbers: one per market, same order (only for "highest custom number")
  - do not re-pick the last [1] markets (0 up to the list size minus 2); at most [6] switches per hour (1 to 3600)
- **How it works:** The block counts finished rounds (pretend rounds on their own counter). When the count reaches N, and no contract is open, it picks the next market. It skips any market that does not offer the contract types of your bot right now (Over, Under or both), any of the last markets it used, and any market outside your list. After a switch it waits for the first fresh price of the new market. Volatility here is the average size of a price move, as a share of the price, over the latest 100 ticks of each market.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────┐
│ Market switcher: switching                           │
│   [ ON (switch between my markets) ▾ ]               │
│   my markets [ R_10, R_25, R_50, R_75, R_100 ]       │
│   mode [ rotate every N rounds ▾ ]                   │
│   N [ 5 ]                                            │
│   best score means [ highest volatility ▾ ]          │
│   custom numbers (one per market, same order) [   ]  │
│   do not re-pick the last [ 1 ] markets              │
│   at most [ 6 ] switches per hour                    │
└──────────────────────────────────────────────────────┘
```

After every 5 finished rounds, when no contract is open, the bot moves to the next market in the list and waits for its first price. (The real default list is longer.)

- **Good to know:**
  - Refusals: a name that is not letters, digits and _; fewer than 2 or more than 30 different markets; a missing mode or score; any number outside its range; custom numbers that do not

match the markets; no price within 20 seconds after a switch; with switching OFF, no market chosen or one that cannot be used.

- If no market in your list can be used, or the hourly limit is reached, the bot stays on the current market and the journal says why. A market written twice is used once. It comes preloaded on a fresh Bot Builder, switched off (disabled): enable it and write your own list. Switching markets does not make a win more likely.

Market allowlist

- **Kind:** BRINOVS custom, block #37
- **Type id:** `bnv_market_allowlist`
- **Shape:** True/false block
- **Where it goes:** In an `if` inside "2. Purchase conditions", above Purchase.
- **What it does:** Says yes only when the market the bot trades, and the contract it is about to buy, are on your list. It only answers. It never changes the market and never swaps in another one.
- **Inputs:**
  - Market allowed: my markets: names separated by commas, default R_10, R_25, R_50
  - or any market of the category: [(none) ▾] (default), [synthetic indices ▾], [forex ▾], [stock indices ▾], [cryptocurrencies ▾] or [commodities ▾]
  - my contract types (empty = any): Deriv codes in capitals, such as DIGITOVER, DIGITUNDER. Empty by default.
  - the contract I am about to buy: pick the contract the Purchase block buys
- **Gives back:** true only when all checks pass. False in every unclear case.
- **How it works:** The market must be on your list (the names you wrote, or the category you chose) and must be in Deriv's live list of active markets right now, not suspended and not closed. If you wrote contract types, the contract you are about to buy must be one of them. With no names and the category on (none), any active market passes. Deriv's list is asked again when needed, and a slow answer (8 seconds) counts as no answer.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < Market allowed: my markets [ R_10, R_25, R_50 ] │
│       or any market of the category [ (none) ▾ ]      │
│       my contract types (empty = any) [ DIGITOVER ]   │
│       the contract I am about to buy [ Over ▾ ] >     │
│ then  Purchase [ Over ▾ ]                             │
└───────────────────────────────────────────────────────┘
```

The bot buys only when it is trading R_10, R_25 or R_50, the market is open at Deriv, and the contract is Over.

- **Good to know:**
  - Refusals: market names with other characters than letters, digits and _; no category chosen; contract types that are not capital letters and digits separated by commas; contract types written but no contract chosen.
  - When it says no, the journal says why once: the market service is not available, the market is not known yet, Deriv's list could not be read, the market is not in the live list, it is suspended or closed, it is not on your list, or the contract is not allowed.
  - It does not say anything about your chance to win.

Barrier switcher

- **Kind:** BRINOVS custom, extra block (no catalogue number)
- **Type id:** `bnv_barrier_switcher`
- **Shape:** Value block (gives a number, a digit)
- **Where it goes:** In the Prediction box of Trade options, or in a barrier box of the Hedge purchase. A barrier is the digit your digit contract is measured against.
- **What it does:** Gives the digit to trade and changes it between rounds, from a list you write. It is the little sister of the Market switcher, for digits instead of markets.
- **Inputs:**
  - Barrier switcher: my barriers: digits from 0 to 9 separated by commas, default 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (at least 2 different)
  - mode: [next barrier every N rounds ▾] (default), [next barrier after N losing rounds in a row ▾] or [random barrier every N rounds ▾]
  - N: a whole number from 1 to 1000, default 5

  - skip digits that do not fit: [the contracts of my bot ▾] (default), [Over ▾], [Under ▾], [Over and Under ▾] or [nothing (use all) ▾]
- **Gives back:** the digit to use now.
- **How it works:** The block counts finished rounds. When the count reaches N, it moves to the next digit in your list (back to the first after the last one), or picks another digit at random, never the same twice in a row. A digit the contract cannot use is skipped, and the journal says so: Over takes 0 to 8, Under takes 1 to 9, Matches and Differs take 0 to 9. The counts are saved, so a refresh does not lose them. Pretend rounds count on their own counter.
- **Example:**

```
┌─ 1. Trade parameters ──────────────────────────────────────┐
│ Prediction: ( Barrier switcher: my barriers [ 3, 4, 5, 6 ] │
│               mode [ next barrier every N rounds ▾ ]       │
│               N [ 5 ]                                      │
│               skip digits that do not fit [ Over ▾ ] )     │
└────────────────────────────────────────────────────────────┘
```

The Prediction starts at 3. After every 5 finished rounds it moves to 4, then 5, then 6, then back to 3.

- **Good to know:**
  - Refusals: a barrier that is not a single digit 0 to 9; fewer than 2 different digits; no mode or check chosen; N outside 1 to 1000; none of your digits fits the contract.
  - It does not check markets, wait for prices or limit changes per hour. Changing a digit moves no money and touches no open contract.
  - It does not make a win more likely.

BRINOVS blocks: Control, state and statistics (38 to 44)

This part of the guide covers two menus of the BRINOVS blocks in the Bot Builder: **Control and state** (blocks 38 to 41) and **Statistics** (blocks 42 to 44). The first menu helps a bot remember things, stop or pause itself, wait, and check that it is safe to buy. The second menu reads back how your trades went. Both menus sit inside the BRINOVS menu of the Blocks list.

A few words used below. A **tick** is one new price from the market. A **stake** is the money you put on one trade. A **trade** (or **contract**) is one purchase. A trade is **settled** when it has finished and its result is known. The **Journal** is the log panel of the Bot Builder, where the bot writes messages to you. Like all Deriv blocks, a BRINOVS block only works when it sits inside one of the four main blocks: **1. Trade parameters**, **2. Purchase conditions**, **3. Sell conditions** or **4. Restart trading conditions**. A block put anywhere else is switched off (greyed out).

Nothing here promises profit. These blocks give you control and information. They do not make a strategy win.

Control and state (blocks 38 to 41)

These blocks look after the bot itself. Persistent variable and Persistent value give the bot a memory that survives a page refresh. Stop or pause the bot ends the run, or holds it, with a reason you can read. Wait until candle close makes the bot wait for the next candle (one bar on a price chart). Bot is ready tells you if the bot is allowed to buy right now. Clear saved rules is a small helper at the end of this section. In a bot they usually run in this order: set-up blocks in Run once at start, then the checks in Purchase conditions, then the stop or pause rules in Sell or Restart conditions.

Persistent variable

- **Kind:** BRINOVS custom, block #38
- **Type id:** `bnv_persistent_variable`
- **Shape:** Action block
- **Where it goes:** inside any of the four main blocks, in the middle of a stack of action blocks (in Trade parameters, use the Run once at start part). It can be used many times, with different names.
- **What it does:** Keeps a named number, text or true/false value that is still there after you refresh the page or restart the bot. Use it as a counter or a flag, for example "how many rounds today".

Read the value back with the Persistent value block.

- **Inputs:**
  - Name: a text box, default `counter`. Letters, digits, `_` and `-`, up to 40 characters, no spaces.
  - Action: `set to` or `change by` (default `set to`).
  - Value socket: a number, text or true/false (the toolbox gives `0`). For `change by` it must be a number.
  - reset: `never`, `after a win`, `after a loss` or `every UTC day` (default `never`).
- **How it works:** The first time a name is set, its first value is remembered as its starting value. `set to` replaces the value. `change by` adds a number to it (a new counter starts at 0). The reset rule sends the value back to its starting value after a settled winning trade, after a settled losing trade, or when a new UTC day begins (UTC is the world time standard; a new UTC day starts at 00:00 UTC). A trade that ends at exactly 0 resets neither. Pretend (dry run) trades never trigger a reset. The value is kept for the logged-in account.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ Persistent variable [ rounds ] [ change by ▾ ] ( 1 )     │
│     reset [ every UTC day ▾ ]                            │
│ Trade again                                              │
└──────────────────────────────────────────────────────────┘
```

This counts one more round each time the bot gets here, and puts the counter back to 0 at the start of every UTC day. The real block is one long row on screen.

- **Good to know:**
  - The bot stops with a message if the name is wrong, the value socket is empty, a text is longer than 500 characters, or `change by` is given something that is not a number.
  - A name keeps its type: a number variable cannot later be set to a text (the bot stops with a message).
  - If the saved data is damaged, it is ignored and the Persistent value block gives its fallback.

Persistent value

- **Kind:** BRINOVS custom, block #38 (the reading half of Persistent variable)
- **Type id:** `bnv_persistent_variable_value`

- **Shape:** Value block (gives a number, text or true/false)
- **Where it goes:** anywhere a value is wanted, inside the four main blocks. It can be used many times.
- **What it does:** Reads a Persistent variable by its name. If that name does not exist yet, it gives the fallback value you put in the second box.
- **Inputs:**
  - Name: a text box, default `counter`. It must match the name used in Persistent variable.
  - if none yet: a value socket for the fallback (the toolbox gives `0`).
- **Gives back:** the saved number, text or true/false. If the name is new or its saved data is damaged, the fallback.
- **How it works:** It looks up the saved value. A variable with the `every UTC day` rule shows its starting value once a new UTC day has begun. Reading never changes the variable.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────┐
│ if  < ( Persistent value [ rounds ] if none yet ( 0 ) )    │
│        ≥ ( 50 ) >                                     │
│ then  Stop the bot ...                                │
└───────────────────────────────────────────────────────┘
```

The bot stops once the saved counter reaches 50 or more.

- **Good to know:**
  - The name must follow the same rules as in Persistent variable, or the bot stops with a message.
  - Put a sensible fallback, so the first run gives a number you expect.

Stop or pause the bot

- **Kind:** BRINOVS custom, block #39
- **Type id:** `bnv_stop_pause`
- **Shape:** Action block
- **Where it goes:** only inside 2. Purchase conditions, 3. Sell conditions or 4. Restart trading conditions, in the middle of a stack. It can be used more than once. A disabled copy comes ready in Restart trading conditions in a fresh Bot Builder.
- **What it does:** On the block it reads "Stop the bot" or "Pause the bot". Stop ends the run now and shows your reason. Pause holds the bot for some seconds, or until you raise a number called the

Resume ticket.

- **Inputs:**
  - Action: `Stop` or `Pause` (default `Stop`).
  - reason: a text socket, shown in the Journal (the toolbox gives `Stopped by my rule`; at most 200 letters are kept).
  - pause seconds: a number socket (toolbox `60`). `0` means "until the Resume ticket is raised".
  - Resume ticket: a number socket (toolbox `0`), used for a pause of 0 seconds.
  - sell open contracts first: a tick box, off by default.
- **How it works:** Stop ends the bot at once. A Pause with seconds above 0 waits that long and then carries on. A Pause with 0 seconds stops the bot and saves the pause: no purchase is allowed until you raise the Resume ticket above the saved one and run the bot again. With the tick box on, the bot first tries to sell every open contract. If selling fails, it still stops or pauses.
- **Example:**

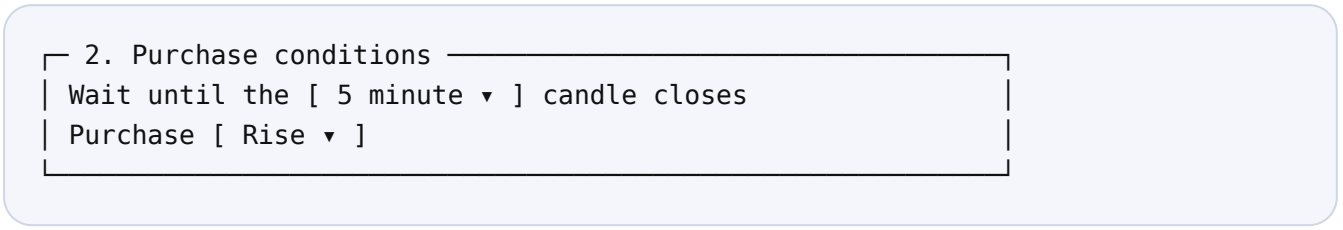
```
┌─ 4. Restart trading conditions ──────────────────────────┐
│ [ Pause ▾ ] the bot                                      │
│ reason ( "Cooling off" )                                 │
│ pause seconds ( 300 ) (0 = until Resume ticket)          │
│ Resume ticket ( 0 )                                      │
│ sell open contracts first □                              │
│ Trade again                                              │
└──────────────────────────────────────────────────────────┘
```

The bot waits five minutes and then trades again. With seconds set to 0 it would stop instead and stay paused until the ticket is raised to 1 or more.

- **Good to know:**
  - Stopping never depends on a licence or any server, so Stop always works.
  - The two number boxes must hold numbers even for Stop. For a pause of 0 seconds the ticket must be a whole number of 0 or more. A pause needs seconds of 0 or more. Otherwise the bot stops with a message.
  - A saved manual pause stays in force even if you delete the block (every purchase is refused with a message). Raise the ticket, or use Clear saved rules, to get out of it.
  - This block does not protect your money by itself: it only acts where you place it and when it is reached.

Wait until candle close

- **Kind:** BRINOVS custom, block #40
- **Type id:** `bnv_wait_until`
- **Shape:** Action block
- **Where it goes:** inside any of the four main blocks, in the middle of a stack of action blocks. It can be used more than once.
- **What it does:** On the block it reads "Wait until the (size) candle closes". The bot pauses at this block until the current candle of the chosen size has closed, then goes on with the next block. For waiting a fixed number of seconds, use Deriv's own blocks.
- **Inputs:**
  - candle size: `1 minute`, `2 minute`, `3 minute`, `5 minute`, `10 minute`, `15 minute`, `30 minute`, `1 hour`, `2 hour`, `4 hour`, `8 hour` or `1 day` (default `1 minute`).
- **How it works:** A candle has closed when the first price of the next candle arrives. The bot uses the time stamps on the market's own prices, so a wrong clock on your phone does not matter. While waiting, the app stays responsive and the Stop button works at any time. The Journal says how many seconds the wait should take.
- **Example:**



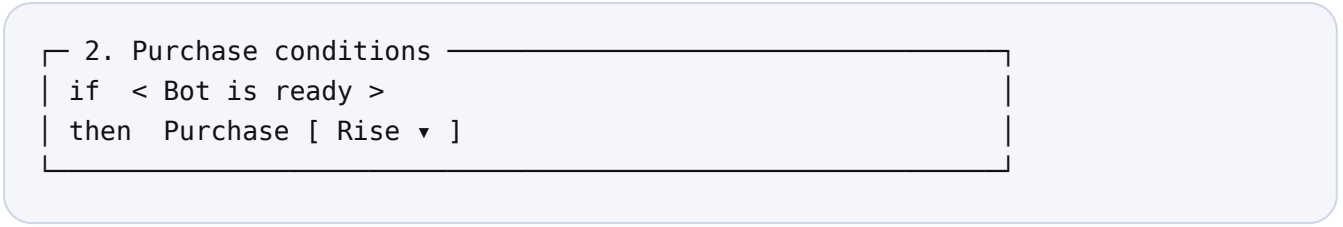
The bot waits until the 5 minute candle is over, and then buys.

- **Good to know:**
  - If no price arrives for 30 seconds after the candle should have closed, the bot stops with a plain message instead of waiting for ever. Check your connection.
  - If you press Stop while it waits, the wait ends quietly.

Bot ready

- **Kind:** BRINOVS custom, block #41
- **Type id:** `bnv_bot_ready`
- **Shape:** True/false block

- **Where it goes:** anywhere a true/false is wanted, for example in an if. It can be used many times.
- **What it does:** On the block it reads "Bot is ready". It is true when the bot is in a safe state to buy: connected, logged in, receiving live prices, and sure of every open or unfinished purchase. It describes the bot, not the market.
- **Inputs:** none.
- **Gives back:** true when everything below is in order, otherwise false.
- **How it works:** The bot checks, in this order: the connection is up, you are logged in, the live data feeds are running, any open contracts and unfinished buys have been matched with Deriv's records, and a fresh price arrived in the last 15 seconds. It is false while it is reconnecting or still checking. An unknown state counts as not ready.
- **Example:**



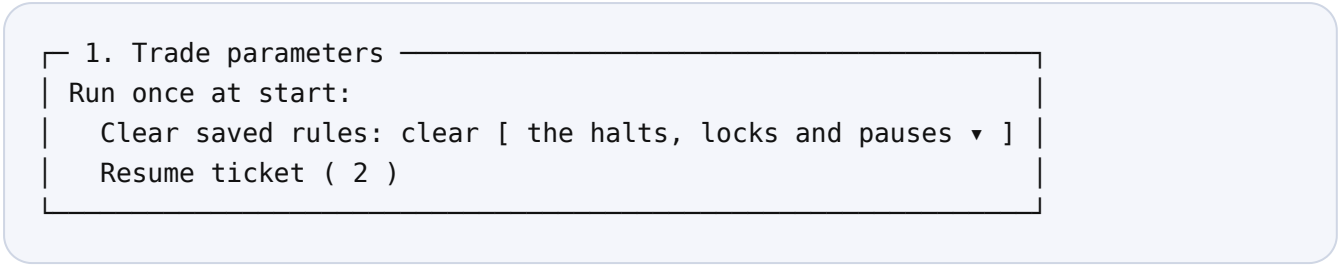
The bot buys only when it is ready.

- **Good to know:**
  - Separate from this block, the bot already refuses every purchase while it is not ready, whether or not you use the block. The block just lets your own rules see the same answer.
  - If the BRINOVS execution layer (the part that buys for the bot) is switched off, this block always gives false.

Clear saved rules

- **Kind:** BRINOVS custom (a helper in the Control and state menu; it has no catalogue number)
- **Type id:** `bnv_clear_saved_rules`
- **Shape:** Action block
- **Where it goes:** only in the Run once at start part of 1. Trade parameters. A disabled copy comes ready there in a fresh Bot Builder.
- **What it does:** Some BRINOVS blocks save their rule (a loss limit, a pause, a lock, a max stake) and the rule stays in force even after you delete the block. This block is the way out: it clears those saved rules so an old rule cannot stop a new bot.
- **Inputs:**

  - Clear: `the halts, locks and pauses (my settings stay)` or `everything saved (settings too)` (default: the halts).
  - Resume ticket: a number socket (toolbox `1`).
- **How it works:** It clears only when its Resume ticket is higher than the last ticket it used (the first time, 1). Leaving it in the bot therefore does not keep wiping your limits. The Journal says what was cleared. It does not touch your trade history, totals or Persistent variables.
- **Example:**



If the last ticket used was 1, ticket 2 clears the saved halts once.

- **Good to know:**
  - The ticket must be a whole number of 0 or more, or the bot stops with a message.
  - The block comes disabled, so it never clears anything by accident. Enable it only when you need it.

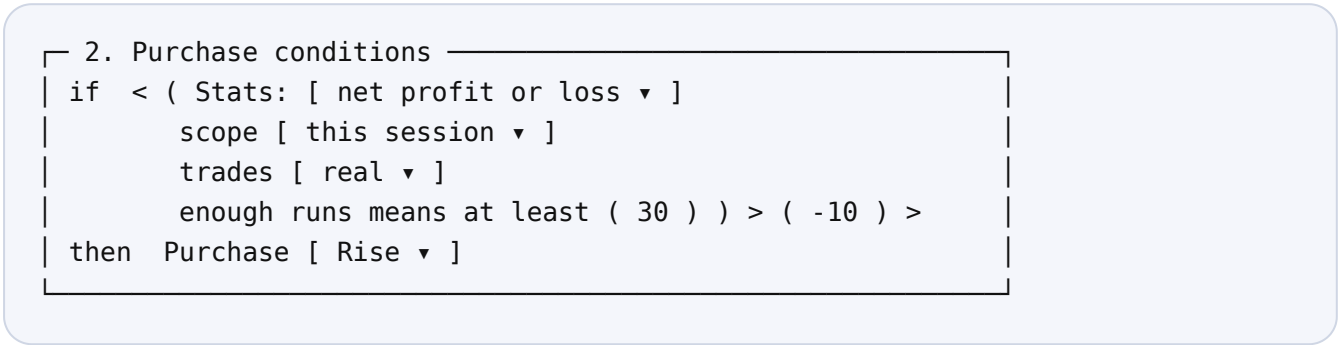
Statistics (blocks 42 to 44)

These blocks are value blocks: they give back numbers or lists that you plug into other blocks, such as an if or a stake box. They read the trades that have already settled, so they describe the past. They never predict the next trade and they never change your strategy by themselves. A **run** (also called a round) is one round of buying: a hedge or a multi purchase counts as one run, with its profits added up. A trade that ends at exactly 0 profit is **break-even**: a run, but neither a win nor a loss. With no trades yet, the answer is 0. Real trades and pretend (dry run) trades are never mixed: you choose which one to read.

Session stats

- **Kind:** BRINOVS custom, block #42
- **Type id:** `bnv_session_stats`
- **Shape:** Value block (gives a number, or true/false)

- **Where it goes:** anywhere a number is wanted (or a true/false, for the last field), inside the four main blocks. It can be used many times.
- **What it does:** On the block it reads "Stats: (field)". It gives one statistic of your settled rounds for the scope you pick: this session, this bot, or today.
- **Inputs:**
  - Stats: `runs`, `wins`, `losses`, `break-even rounds`, `win rate (%)`, `net profit or loss`, `profit factor`, `current streak (+ wins, - losses)`, `longest winning streak`, `longest losing streak`, `average duration` or `enough runs? (true/false)` (default `runs`).
  - scope: `this session` (since this run started), `this bot (all kept records)` (the last 5000 trades kept for this account) or `today (UTC)` (since 00:00 UTC). Default `this session`.
  - trades: `real` or `pretend (dry run)` (default `real`).
  - enough runs means at least: a whole number from 1 (toolbox `30`).
- **Gives back:** a number (a true/false for `enough runs?`).
- **How it works:** Win rate is wins divided by runs, times 100. Profit factor is what the winning rounds made divided by what the losing rounds lost. It is "not a number" when there are wins and no losses. A streak is how many wins (+) or losses (-) came in a row; a break-even round ends it. Average duration uses the unit the trades were bought with (ticks, seconds and so on) and is "not a number" if the units are mixed. `enough runs?` is true when you have at least the number of runs you set: small samples mean little.
- **Example:**



The bot buys only while this session's net result is above -10. (The ">" in the middle is a Deriv compare block.)

- **Good to know:**
  - The bot stops with a message if the minimum is not a whole number from 1 to 100000.

- A statistic does not enforce a limit. To stop a bot after a loss, use a block that does (such as Session limits, or Stop or pause the bot).
- Few runs say little: add the `enough runs?` field to your rule.

Rolling stats

- **Kind:** BRINOVS custom, block #43
- **Type id:** `bnv_rolling_stats`
- **Shape:** Value block (gives a number)
- **Where it goes:** anywhere a number is wanted, inside the four main blocks. It can be used many times.
- **What it does:** Gives one statistic of only your most recent settled trades, for example the win rate of the last 20. You can narrow it to one market and one contract type.
- **Inputs:**
  - Rolling stats: `win rate (%)`, `average result per trade`, `average win`, `average loss (as a positive number)` or `sample size (trades counted)` (default `win rate (%)`).
  - over the last: a whole number of trades from 1 to 5000 (toolbox `20`).
  - only market (empty = any): a text box, empty by default. Letters, digits and `_`, for example `R_100`.
  - only contract type (empty = any): a text box, empty by default. Written in capitals, for example `DIGITOVER`.
  - trades: `real` or `pretend (dry run)` (default `real`).
- **Gives back:** a number. With no matching trades, 0.
- **How it works:** The market and contract-type filters are applied first, then the last K of what is left. This block counts single trades, not rounds. Win rate is wins divided by trades, times 100. Average result per trade is the sum of the results divided by the number of trades. Average loss is shown as a positive number.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ( Rolling stats: [ win rate (%) ▾ ]                 │
│         over the last ( 20 ) trades                       │
│         only market (empty = any) [ R_100 ]               │
│         only contract type (empty = any) [ DIGITOVER ]    │
│         trades [ real ▾ ] ) > ( 50 ) >                    │
│ then  Purchase [ Over ▾ ]                                 │
└───────────────────────────────────────────────────────────┘
```

The bot buys only if the last 20 real DIGITOVER trades on R_100 won more than half the time.

- **Good to know:**
  - The bot stops with a message if the count is not a whole number from 1 to 5000, or the market or type is written wrongly (for example lower case letters in the type).
  - Ask for the sample size next to the other number: a win rate over 3 trades means almost nothing.
  - It describes history. It does not predict and it never changes your strategy on its own.

Trade history

- **Kind:** BRINOVS custom, block #44
- **Type id:** `bnv_trade_history`
- **Shape:** Value block (gives a list)
- **Where it goes:** anywhere a list is wanted, inside the four main blocks, for example in a Deriv list block. It can be used many times.
- **What it does:** Gives a list of your last settled trades, oldest first and newest last. You can get whole rows, or just one column (all the profits, for example).
- **Inputs:**
  - Trade history: the last (N) settled trades: a whole number from 1 to 500 (toolbox `20`).
  - give me: `a list of rows (time, market, type, stake, profit)`, `a list of times (seconds)`, `a list of markets`, `a list of contract types`, `a list of stakes` or `a list of profits` (default: rows).
  - trades: `real` or `pretend (dry run)` (default `real`).
- **Gives back:** a list. A row is [time in seconds, market, contract type, stake, profit]. An empty history gives an empty list.

- **How it works:** It takes the settled trades of the kind you chose, keeps the last N, and gives them back as rows or as one column. The time is in seconds, not milliseconds. At most 500 trades are given, whatever the saved history holds.
- **Example:**

```
┌─ 4. Restart trading conditions ──────────────────────────────┐
│ set [ recent ▾ ] to ( Trade history: the last ( 20 ) settled │
│       trades give me [ a list of profits ▾ ]                 │
│       trades [ real ▾ ] )                                    │
└──────────────────────────────────────────────────────────────┘
```

The variable holds the profits of the last 20 real trades. You can then use Deriv's list blocks, such as the sum of a list.

- **Good to know:**
  - The bot stops with a message if N is not a whole number from 1 to 500.
  - It reads and describes. It buys or sells nothing and changes nothing.

BRINOVS blocks: Alerts, testing and licensing (45 to 51)

This part of the guide covers three menus of the BRINOVS blocks in the Bot Builder: **Alerts and logging** (blocks 45 to 47), **Testing** (block 48) and **Brinovs identity and licensing** (blocks 49 to 51). The first two menus are for you, the person who builds and watches a bot. The third menu is for bots that are shared through the BRINOVS Plug & Play library. All three menus sit inside the BRINOVS menu of the Blocks list.

A few words used below. A **tick** is one new price from the market. A **stake** is the money you put on one trade. A **trade** (or **contract**) is one purchase. A trade is **settled** when it has finished and its result is known. The **Journal** is the log panel of the Bot Builder, where the bot writes messages to you. A **CSV file** is a plain table file that opens in Excel or Google Sheets. Like all Deriv blocks, a BRINOVS block only works when it sits inside one of the four main blocks: **1. Trade parameters**, **2. Purchase conditions**, **3. Sell conditions** or **4. Restart trading conditions**.

Nothing here promises profit. These blocks record, warn, test and protect shared bots. They do not make a strategy win.

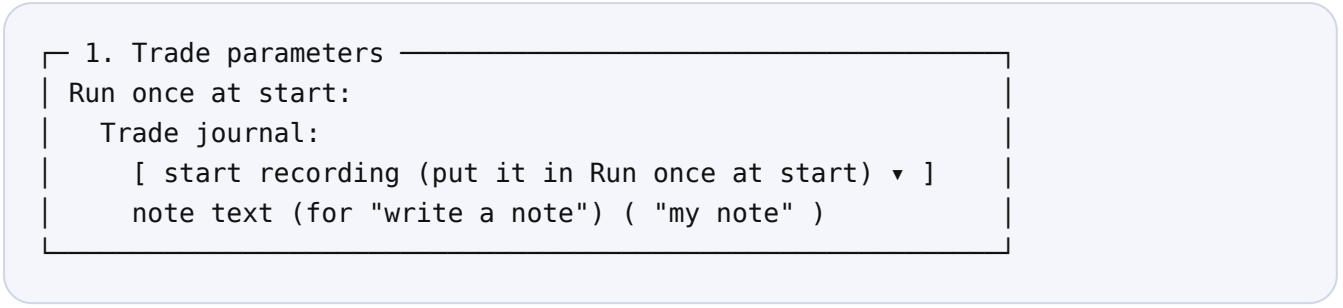
Alerts and logging (blocks 45 to 47)

These three blocks tell you what the bot did, and why. Trade journal keeps a saved record you can download. Alert shows you a message when something you care about happens. Bot narration says in plain words what the bot is doing as it goes. They write and show messages inside the app, and nothing leaves it except files you choose to download. One block has a stronger power: an Alert can also pause new purchases if you tick its box. A normal order is to switch the Trade journal and Bot narration on in Run once at start, and to place Alert blocks where the event happens.

Trade journal

- **Kind:** BRINOVS custom, block #45
- **Type id:** `bnv_trade_journal`
- **Shape:** Action block
- **Where it goes:** inside any of the four main blocks, in the middle of a stack of action blocks. "Start recording" belongs in Run once at start of 1. Trade parameters. It can be used more than once (for example one to start, one to write notes).

- **What it does:** Keeps a record of what the bot did that is still there after you refresh the page. You can add notes of your own and download everything as a CSV file.
- **Inputs:**
  - Trade journal: `start recording (put it in Run once at start)`, `write a note`, `download as a CSV file` or `clear the saved journal` (default: start recording).
  - note text (for "write a note"): a text socket (the toolbox gives `my note`).
- **How it works:** Once recording is on, the bot writes one row for each of these: bot started or stopped, a buy request, a purchase that was refused (with the rule and the reason), the contract id, a buy that failed, a settlement with its profit, a sell, a market change and a batch result. Rows of one purchase are linked by a request id, and each row carries the bot version. Ticks and balance updates are not recorded. Tokens and secret links are never written. The last 5000 rows are kept, and the same row twice within 2 seconds is written once.
- **Example:**



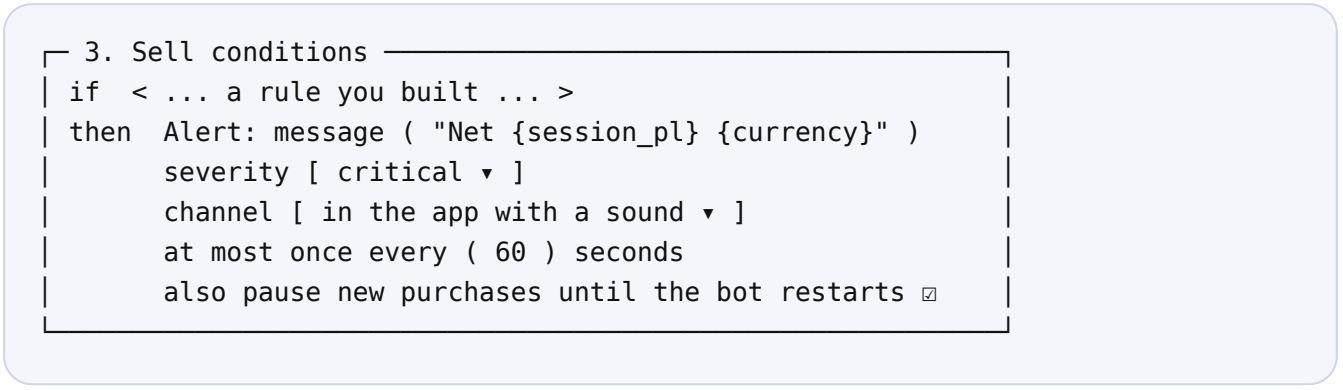
From now on the bot logs every buy, refusal and result. The real block shows "Trade journal:" and the choice on one row.

- **Good to know:**
  - "write a note" with an empty text stops the bot with a message ("the note is empty").
  - A phone browser may block the CSV download. The journal stays saved, so you can download it later from a computer.
  - If the browser storage is full or blocked, the bot keeps running and says so once in the log. Some rows may then be missing.
  - "download" with nothing recorded only tells you the journal is empty.

Alert

- **Kind:** BRINOVS custom, block #46
- **Type id:** `bnv_alert`
- **Shape:** Action block

- **Where it goes:** inside any of the four main blocks, in the middle of a stack of action blocks, usually under an if. It can be used many times.
- **What it does:** Shows you a message in the Journal, with live values filled in, and an optional sound. It can also pause new purchases. An alert alone does not stop anything.
- **Inputs:**
  - message: a text socket. Use `{time}`, `{balance}`, `{currency}`, `{session_pl}` (net result of real trades this session) and `{runs}` where you want live values. Any other `{word}` stays as you typed it. The toolbox gives `Session result {session_pl} {currency} after {runs} runs`.
  - severity: `info`, `warning` or `critical` (default `info`). It changes the colour and the sound.
  - channel: `in the app` or `in the app with a sound` (default `in the app`).
  - at most once every (N) seconds: a number socket (toolbox `60`). The first alert always goes out.
  - also pause new purchases until the bot restarts: a tick box, off by default. It switches itself on when you pick `critical` (you can change it after).
- **How it works:** The bot fills in the live values and shows the message. The same alert (same text, severity and channel) is shown at most once per the seconds you set. With the tick box on, every later purchase is refused until you stop the bot and run it again.
- **Example:**



When the rule is true, you see and hear a critical message with the session result, and the bot buys nothing more until you restart it.

- **Good to know:**
  - It does not send messages outside the app. For Telegram, use the Notify Telegram block that Deriv already provides.

- An alert does not enforce a limit. Only the tick box makes it a real pause, and that pause only covers new purchases.
- The bot stops with a message if the text is empty or longer than 300 characters, or the seconds are not a number from 0 to 86400.
- The pause needs the BRINOVS execution layer (the part that buys for the bot). If it is switched off, an alert with the tick box on stops the bot with a message.

Bot narration

- **Kind:** BRINOVS custom, block #47
- **Type id:** `bnv_bot_narration`
- **Shape:** Action block
- **Where it goes:** inside any of the four main blocks, in the middle of a stack of action blocks. "Start" belongs in Run once at start of 1. Trade parameters. It can be used more than once.
- **What it does:** The bot tells you, in plain words, what it is doing: it started, what it is buying, what a rule refused and why ("skipped: ..."), how a trade ended (won or lost), what it sold, a market change, and the stop. It only talks. It changes nothing in the bot.
- **Inputs:**
  - Bot narration: `start telling me what you do (put it in Run once at start)`, `say my line`, `download the last 200 lines` or `clear the saved lines` (default: start).
  - my line (for "say my line"): a text socket (the toolbox gives `skipped: payout below minimum`).
  - detail: `beginner (short)` or `pro (ids and rule names)` (default beginner).
- **How it works:** After "start", the bot listens to what happens and writes a line for each event. Beginner keeps lines short. Pro adds contract ids, request ids, rule names and smaller steps. To keep it readable, only one line is shown every 2.5 seconds; the others wait in a short queue of up to 20. If too many pile up, the oldest minor ones are dropped and counted ("n lines were left out"). The same sentence twice within 10 seconds is shown once. The last 200 lines are kept and can be downloaded as a text file.
- **Example:**

```
┌─ 2. Purchase conditions ──────────────────────────────────┐
│ if  < ... a rule you built ... >                          │
│ then  Bot narration: [ say my line ▾ ]                    │
│       my line (for "say my line") ( "skipped: payout low" ) │
│       detail [ beginner (short) ▾ ]                       │
└───────────────────────────────────────────────────────────┘
```

When the rule is true, "BRINOVS says: skipped: payout low" appears in the Journal (after any waiting lines).

- **Good to know:**
  - "say my line" with an empty line stops the bot with a message. A line is cut at 200 letters.
  - Tokens and secret links are never written.
  - A busy bot can show lines a few seconds late, because of the one-line-per-2.5-seconds rule.

Testing (block 48)

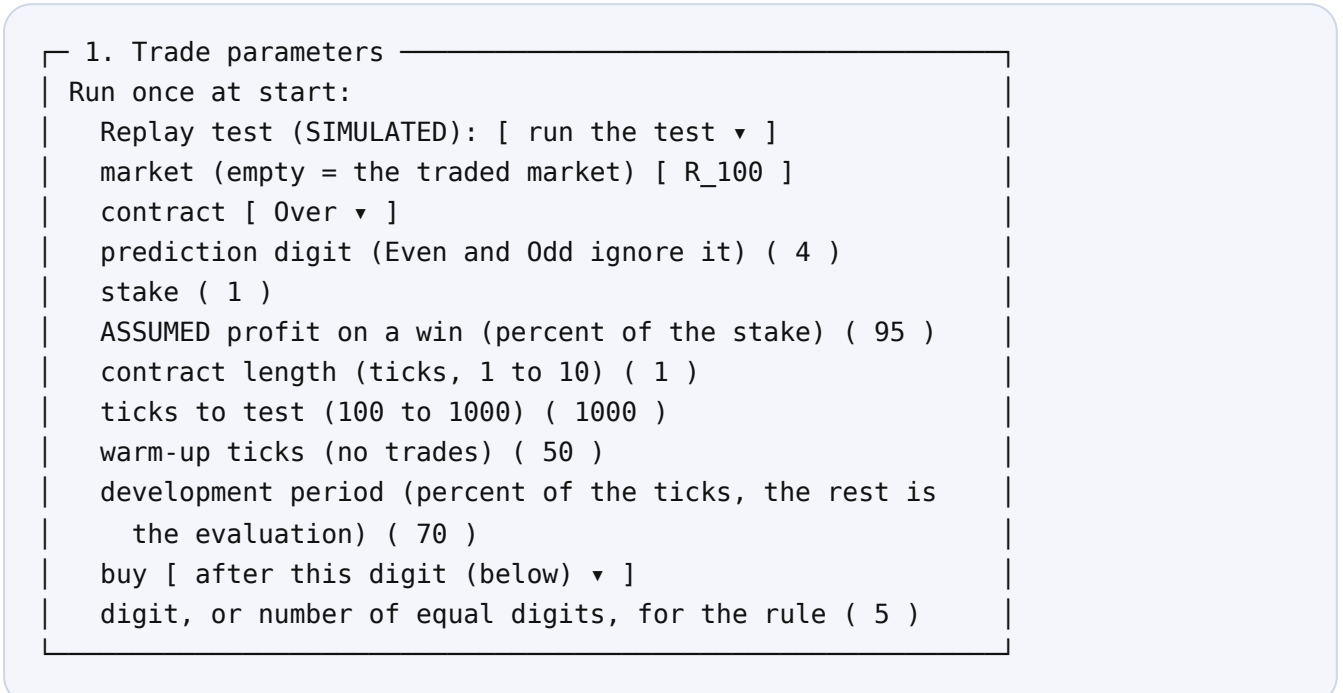
Testing means trying a rule on past prices instead of risking money. The one Testing block replays a simple digit rule over the latest stored ticks of a market. Everything it shows is **simulated**: it buys and sells nothing and does not touch your real trade history. Use it in a separate test bot, never in the bot that trades.

Replay test (simulated)

- **Kind:** BRINOVS custom, block #48
- **Type id:** `bnv_replay_test`
- **Shape:** Action block
- **Where it goes:** inside any of the four main blocks, in the middle of a stack of action blocks, normally in Run once at start of a test bot. It can be used more than once.
- **What it does:** On the block it reads "Replay test (SIMULATED)". It plays one digit rule over the latest ticks of a market, one tick at a time, and writes the result to the Journal. The last trade log is saved and can be downloaded as a CSV file.
- **Inputs:**
  - action: `run the test` or `download the last trade log` (default run).
  - market (empty = the traded market): a text box, empty by default. Letters, digits and `_`, for example `R_100`.

- contract: `Over`, `Under`, `Matches`, `Differs`, `Even` or `Odd` (default Over).
- prediction digit: a number socket (toolbox `4`); Over 0 to 8, Under 1 to 9, Matches and Differs 0 to 9; Even and Odd ignore it.
- stake: a number socket above 0 (toolbox `1`).
- ASSUMED profit on a win (percent of the stake): 1 to 1000 (toolbox `95`). A loss loses the whole stake. Deriv's real payout differs, so this is your assumption.
- contract length (ticks, 1 to 10): toolbox `1`.
- ticks to test (100 to 1000): toolbox `1000`.
- warm-up ticks (no trades): 0 up to the ticks to test minus 50 (toolbox `50`).
- development period (percent of the ticks): 10 to 90 (toolbox `70`). The rest is the evaluation period.
- buy: `at every chance`, `after this digit (below)` or `after this many equal digits in a row (below)` (default at every chance).
- digit, or number of equal digits, for the rule: toolbox `5`. For "after this digit" it is 0 to 9; for "equal digits in a row" it is 2 to 10.

- **How it works:** The bot asks Deriv for the latest ticks (at most 1000) and plays the rule on them. It never looks at a later tick when it decides. It reports the development period (where you may tune the rule) and the evaluation period (the honest one) apart, with trades, wins, losses, win rate, net result, biggest drop and longest losing streak. It also tells you the win rate you need just to break even with your assumed payout.
- **Example:**



The Journal gets lines starting "BRINOVS replay (SIMULATED, not real trades)", one for each period, plus the break-even win rate and warnings.

- **Good to know:**
  - It warns you when Deriv gave fewer ticks than asked, when ticks are missing, and when a period has fewer than 30 trades (too few to mean anything).
  - It describes what a rule would have done in the past. It does not say what the rule will do next. A rule tuned on the same ticks it is judged on looks better than it is, so trust the evaluation period.
  - It needs a connection and the BRINOVS execution layer (the part that buys for the bot, on unless switched off) to read ticks. If the ticks cannot be read, or too few come back, nothing is tested.
  - The bot stops with a message if a number is outside its range, a market name has odd characters, or the market box is empty and the traded market is not known.

Brinovs identity and licensing (blocks 49 to 51)

These three blocks are not for everyday personal bots. They are for **bots published through the BRINOVS Plug & Play library**. Plug & Play is the tab where clients load bots, free ones and premium ones, that the developer has put there. The **developer** (the person who publishes the bot) adds the three blocks to the bot. When the developer publishes the bot, the BRINOVS platform writes a **seal** into the BRINOVS core block. The seal is made from the bot's blocks.

A client who loads the bot can change the numbers (for example the stake, take profit or stop loss). The client cannot change the blocks: adding, removing or rearranging blocks, picking a different dropdown choice or changing a text makes the bot count as changed, and a changed bot does not buy. Premium bots are also loaded in a locked mode, where they cannot be saved, downloaded or have blocks moved or deleted. The three blocks go in Run once at start of 1. Trade parameters and are not preloaded in a fresh Bot Builder, so a personal bot never carries them.

All three work on **new purchases only**. They never block the Stop button, never block selling, and never stop the bot watching contracts that are already open. An honest limit: anything that runs in a browser can be changed by an expert. Together with locked mode and the BRINOVS server, these blocks make copying and misuse hard and traceable, not impossible.

BRINOVS core

- **Kind:** BRINOVS custom, block #49

- **Type id:** `bnv_brinovs_core`
- **Shape:** Action block
- **Where it goes:** only in Run once at start of 1. Trade parameters. A bot has exactly one.
- **What it does:** Says who the bot is: its id, its version and the BRINOVS app version it needs. When the developer publishes the bot on the platform, the platform writes the seal into the manifest box. If the bot is changed after that, it no longer matches its seal and does not buy.
- **Inputs:**
  - bot id: a text box, default `my-bot`. 3 to 64 letters, digits, `-` or `_`, starting with a letter or digit.
  - version: a text box, default `1.0.0`. Three whole numbers with dots.
  - needs app version (..) or newer: a text box, default `1.0.0`. Three whole numbers with dots, or empty.
  - manifest (written by the platform when published): a text box, empty by default. Leave it alone: the platform fills it in.
- **How it works:** When the bot starts, the block works out a fingerprint of the bot's blocks and compares it with the seal in the manifest. Positions, folding and plain numbers are not part of the fingerprint, so moving blocks around and changing numbers is fine. A bot with no manifest (not published) just says who it is. If this page is older than the app version the bot needs, the bot stops and asks you to reload.
- **Example:**

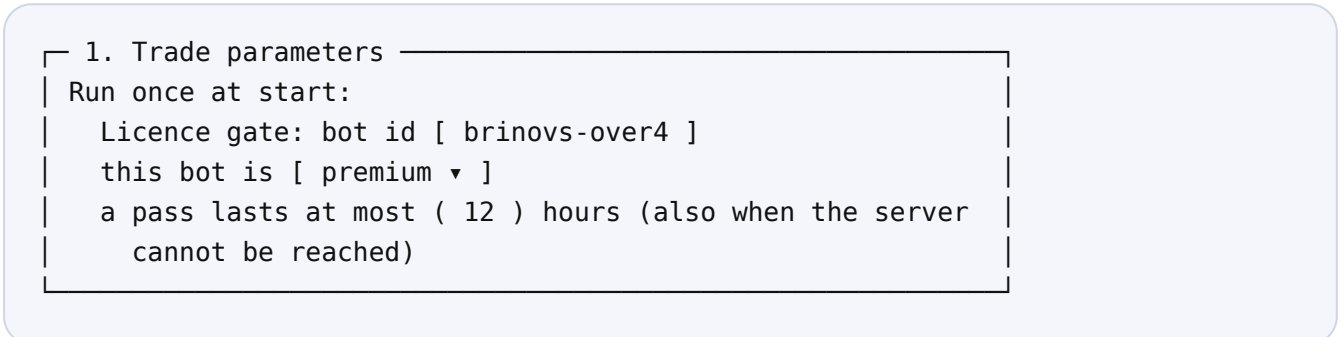
This bot calls itself brinovs-over4, version 1.0.0. The manifest shown is shortened here; the platform writes the full seal.

- **Good to know:**
  - If the bot was changed after publishing, the Journal says so and every purchase is refused with "Nothing was bought". Load the bot again from Plug & Play.

- The bot stops with a message if the id or version is badly written, if the manifest is damaged, if the bot has two core blocks with different ids, or if the app is too old.
- The bot id in the Licence gate and Remote switch blocks must be the same as here.
- This is a seal, not a lock. It makes casual changes useless; it cannot stop an expert who controls his own browser.

Licence gate

- **Kind:** BRINOVS custom, block #50
- **Type id:** `bnv_license_gate`
- **Shape:** Action block
- **Where it goes:** only in Run once at start of 1. Trade parameters, next to BRINOVS core, with the same bot id as the core block.
- **What it does:** Decides if this bot is free or premium. A premium bot asks the BRINOVS server, every time it starts, whether the client who is logged in may run it now. Without a yes, the bot places no new purchases.
- **Inputs:**
  - bot id: a text box, default `my-bot`. Same rules as in BRINOVS core.
  - this bot is: `free` or `premium` (default `free`).
  - a pass lasts at most (N) hours: a whole number from 1 to 72 (toolbox `12`). It also applies when the server cannot be reached.
- **How it works:** Free: nothing is checked. Premium: the bot asks the server. The server knows who the client is from his Deriv login, and decides from the access the developer gave (by login id or by a code). Yes gives a pass for at most the hours you set. No (no access, expired, revoked, or a changed copy of the bot) gives no new purchases, with the reason and the way to get access. A pass that runs out while the bot runs is renewed at the next purchase, at most once a minute.
- **Example:**



A client with access gets a pass of up to 12 hours. A client without access sees the reason in the Journal, and no purchase is made.

- **Good to know:**
  - If the server cannot be reached, a pass from the last hours still works. Without one, there are no new purchases until it can be reached.
  - A "no" tells the client to open Plug & Play and use "Chat with dev / Request code".
  - The server decides, never a claim made by the browser. The bot only sends its id, version, fingerprint and seal, plus which of the client's accounts runs it.
  - The bot stops with a message if the id is bad, the hours are not a whole number from 1 to 72, or the id differs from the core block's id.
  - A developer can also switch on a free trial of 10 runs for demo accounts; the server decides that.

Remote switch

- **Kind:** BRINOVS custom, block #51
- **Type id:** `bnv_remote_kill_switch`
- **Shape:** Action block
- **Where it goes:** only in Run once at start of 1. Trade parameters, next to BRINOVS core, with the same bot id as the core block.
- **What it does:** Lets the developer switch the bot off from the BRINOVS server, for one bot, one client, one version or all of them. The bot checks the switch when it starts and every N minutes.
- **Inputs:**
  - bot id: a text box, default `my-bot`. Same rules as in BRINOVS core.
  - check every (N) minutes: a whole number from 1 to 1440 (toolbox `10`).
  - if it cannot be checked: `keep trading` or `stop new purchases (after two missed checks)` (default keep trading).
  - when it is switched off: `stop new purchases` or `stop new purchases and sell the open contracts` (default stop new purchases).
- **How it works:** Switched off: no new purchases, with the developer's message in the Journal. If you chose to sell, the bot also tries to sell the open contracts once; any that cannot be sold settle by themselves. Switched on again: the bot trades again. If the switch cannot be checked, it either keeps trading or stops new purchases, as you chose. With the stop choice, the bot also stops new purchases when no check has ever worked, or none has worked for two rounds.

- **Example:**

```
┌─ 1. Trade parameters ──────────────────────────────────────┐
│ Run once at start:                                         │
│   Remote switch: bot id [ brinovs-over4 ]                  │
│   check every ( 10 ) minutes                               │
│   if it cannot be checked [ keep trading ▾ ]               │
│   when it is switched off [ stop new purchases ▾ ]         │
└────────────────────────────────────────────────────────────┘
```

The bot asks the server every 10 minutes. If the developer switches it off, it makes no new purchases.

- **Good to know:**
  - Open contracts are always still watched, and the Stop button always works.
  - The bot stops with a message if the id is bad, the minutes are not a whole number from 1 to 1440, a choice is missing, or the id differs from the core block's id.
  - It is a rule on new purchases. It cannot stop a determined expert who controls his own browser.

BRINOVS Digits is a free bot builder and trading page built on Deriv's Bot Builder. Trading carries risk and you can lose money. Nothing in this guide is financial advice or a promise of profit. Test every bot on a demo account first.

Guide home · Open BRINOVS