



Table of Contents

[Overview](#)

[Program Panes and Preferences](#)

[Code Pane](#)

[Preferences](#)

[Edit/View](#)

[Variables Pane and Edit/Track Variable Window](#)

[PlayField Pane](#)

[Pivot And Throw PlayField](#)

[Seek And Score PlayField](#)

[Bot Information and Modelling](#)

[Dimensions:](#)

[Wheel Motors:](#)

[Servo Motors:](#)

[Sensors:](#)

[IR Receiver](#)

[VEX Bumper Switches](#)

[Yellow Pushbutton Switch](#)

[Wheel Encoders](#)

[LFM Line Tracker](#)

[IR Ranger](#)

[Grip Force](#)

[Attaching Pins to Motors and Sensors](#)

[The Uno Pins WIndow](#)

[Communications with the Bot](#)

[Infrared Communication from Beacons to the Bot](#)

[BlueTooth Communications with the Bot](#)

[Menus](#)

[File menu commands:](#)

[Load INO or PDE Prog \(ctrl-L\)](#)
[Edit/View \(ctrl-E\)](#)
[Save \(ctrl-S\)](#)
[Save As](#)
[Next \(#include\) file](#)
[Previous](#)
[Exit](#)

PlayField menu commands:

[Pivot and Throw](#)
[Seek and Score](#)

Find menu commands:

[Ascend Call Stack](#)
[Descend Call Stack](#)
[Set Search Text \(ctrl-F\)](#)
[Find Next Text](#)
[Find Previous Text](#)

Execute menu commands:

[Step Into \(F2\)](#)
[Step Over \(F4\)](#)
[Step Out Of](#)
[Run To](#)
[Run](#)
[Halt](#)
[Reset](#)
[Animate](#)
[Slow Motion](#)

Help menu commands:

[Quick Help File](#)
[Full Help File](#)
[Bug Fixes](#)
[Change/Improvements](#)
[About](#)

Options menu commands:

[Skip through Structors/Operators](#)
[Register-Allocation Modelling](#)
[Error on Uninitialized](#)
[Artificial loop\(\) Delay](#)
[Auto Beacons](#)
[Allow Nested Interrupts](#)

Configure menu commands:

[Wire Up PIns](#)
[Preferences](#)
[Wheel Speed Mismatch](#)

VarUpdates menu commands:

[Allow Auto \(-\) Collapse](#)
[Minimal](#)
[Highlight Changes](#)

Windows menu commands:

[BT Monitor](#)
['Uno' Pins](#)
[Restore All](#)
[Prompt](#)
[Digital Waveforms](#)
[Analog Waveform](#)

Timing and Sound Modelling

Intro

Timing

Sounds modelling

Limitations and Unsupported Elements

Included Files

Dynamic Memory allocations and RAM

'Flash' Memory Allocations

'String' Variables

Arduino Libraries

Pointers

'class' and 'struct' Objects

Scope

Qualifiers 'unsigned', 'const', 'volatile', 'static'

Compiler Directives

Arduino-language elements

C/C++-language elements

Function Templates

Real-Time Emulation

Release Notes

Bug Fixes

V3.1 June 2021 – First Release

Changes/Improvements

V3.1 – June 2021 First Release



Overview

Q2WDBotSim is a **real-time** (see Modelling for Timing **restrictions**) simulator tool that I developed for the student in second year Electrical & Computer Engineering. It is designed to allow you to experiment with, and to easily debug, Q2WD-Bot targetted Arduino programs **without the need for access to the actual physical robot**. It allows you to choose your Uno board's pin connections to the 2WD Bot's sensors, servos, and drive wheel otors, and to test the resulting Bot behaviour in the **PlayField Pane's virtual environments**.

Q2WDBotSim provides simple error messages for any parse or execution errors it encounters, and allows debugging with Reset, Run, Run-To, Run-Till Halt, and flexible Stepping in the **Code Pane**, with a simultaneous view of all global and currently-active local variables, arrays, and objects in the **Variables Pane**. Run-time array-bounds checking is provided, and 'Uno' RAM overflow will be detected (and the culprit program line highlighted!).

When an INO or PDE program file is opened, it is loaded into the program **Code Pane**. The program is then parsed, and "compiled" into a tokenized executable which is then ready for **simulated execution** (unlike Arduino.exe, a standalone binary executable is *not* created) Any parse error is detected and flagged by highlighting the line that failed to parse, and reporting the error on the **Status Bar** at the very bottom of the Q2WDBotSim application window. An **Edit/View** window can be opened to allow you to see and edit a syntax-highlighted version of your user program. Errors during simulated execution (such as mis-matched baud rates) are reported on the Status bar, and via a pop-up MessageBox.

Q2WDBotSim V3.1 includes a substantially complete implementation of the **Arduino Programming Language V1.8.8 as documented at arduino.cc**'s Language Reference web page, and with additions as noted in the version's Download page Release Notes. Although Q2WDBotSim does not support the full C++ implementation that Arduino.exe's underlying GNU compiler does, it is likely that only the most advanced programmers would find that some C/C++ element they wish to use is missing (and of course there are always simple coding work-arounds for such missing features). In general, I have supported only what I feel are the most useful C/C++ Arduino features for your courses -- for example, **enum's** and **#define's** are supported, but function-pointers are not. Even though user-defined objects (classes and structs) and (most) operator-overloads are supported, *multiple-inheritance is not*.

Because Q2WDBotSim is a high-level-language simulator, **only C/C++ statements are supported; assembly language statements are not**. Similarly, because it is not a low-level machine simulation, 'Uno'328 registers are not accessible to your program for either reading or writing.

Q2WDBotSim has built-in automatic support for a limited subset of the Arduino provided libraries, including :Servo.h, SoftwareSerial.h, and EEPROM.h, as no other libraries are needed (or even useful) with this robot. For any #include'd user-created libraries, Q2WDBotSim will **not** search the usual Arduino installation directory structure to locate the library; instead you **need** to copy the corresponding header (.h) and source (.cpp) file to the same directory as the program file that your are working on (subject of course to the limitation that the contents of any #include'd file(s) must be fully understandable to Q2WDBotSim's parser).

I developed Q2WDBotSim in QtCreator and it is currently only available for Windows™. Q2WDBotSim has been tested reasonably extensively, but there are bound to be a few bugs still hiding in there. If you would like to report a bug, please describe it (briefly) in an email to q2wdbotsim@gmail.com and **be sure to attach your full-bug-causing-program Arduino source code** so I can replicate the bug and fix it.

Cheers,
Stan Simmons, Ph.D, P.Eng.
Associate Professor (retired)
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
June 2021



Program Panes and Preferences

Code Pane

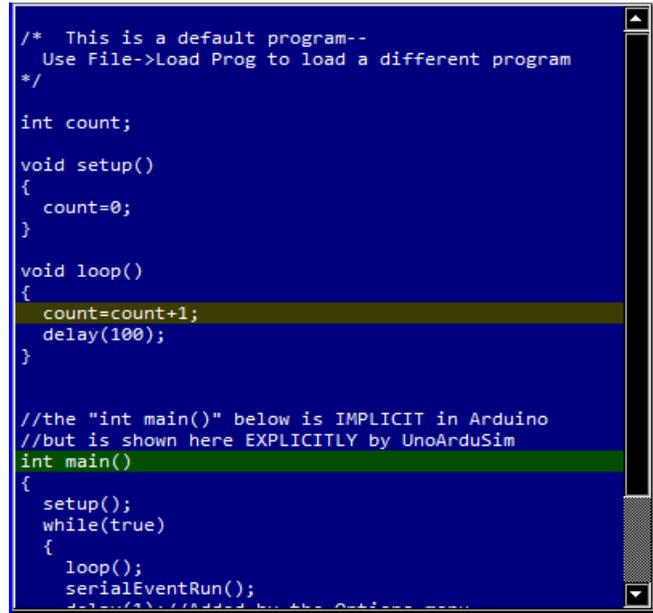
The **Code Pane** displays your user program, and **green** highlighting tracks its execution. (or highlights **red** for an error)

After a loaded program has a successful Parse, the first line in 'main()' is highlighted, and the program is ready for execution. Note that 'main()' is implicitly added by Arduino (and by UnoArduSim) and you do **not** include it as part of your user program file. Execution is under control of the menu **Execute**, and its associated **Tool-Bar** buttons and function-key shortcuts.

After stepping execution by one (or more) instructions (you can use **Tool-Bar** buttons **Step**, **Step In**, or **Step Out**), the program line that will be executed next is then highlighted in green – the green-highlighted line is always the next line **ready to be executed**.

If program execution is currently halted, and you click in the **Code Pane** window, the line you just clicked becomes highlighted in dark olive (as shown in the picture) – the next-to-be-executed line always stays highlighted in green (as of V2.7). But you can cause execution to progress up to the line you just clicked on by then clicking the **Run-To** **Tool-Bar** button. This feature allows you to quickly and easily reach specific lines in a program so that you could subsequently step line by line over a program portion of interest.

If your loaded program has any '#include' files, you can move between them by using **File | Previous** and **File | Next** (with **Tool-Bar** buttons **Left Arrow** and **Right Arrow**). The last user-clicked line in each of these modules remains highlighted, and defines a possible breakpoint line to be run to, but only the breakpoint in the *currently displayed module* is active at the next **Run-To**.



```
/* This is a default program--
   Use File->Load Prog to load a different program
*/

int count;

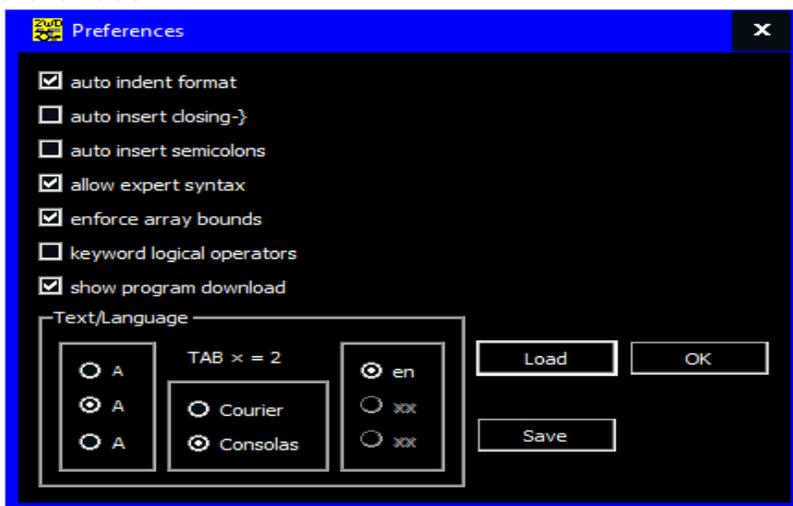
void setup()
{
  count=0;
}

void loop()
{
  count=count+1;
  delay(100);
}

//the "int main()" below is IMPLICIT in Arduino
//but is shown here EXPLICITLY by UnoArduSim
int main()
{
  setup();
  while(true)
  {
    loop();
    serialEventRun();
  }
}
```

The **Find** menu actions allow you to **EITHER** find text in the **Code Pane** or **Variables Pane** (**Tool-Bar** buttons **Find Text** and **Find Text**, or keyboard shortcuts **up-arrow** and **down-arrow**) **after first using Find | Set Search text** or **Tool-Bar** **Find Text**, **OR ALTERNATIVELY** to **navigate the call-stack** in the **Code Pane** (**Tool-Bar** buttons **Find Func** and **Find Func**, or keyboard shortcuts **up-arrow** and **down-arrow**). Keys **PgDn** and **PgUp** jump selection to the next/previous function..

Preferences



Options→Preferences allows users to set program options and viewing preferences (that a user will normally wish to adopt at the next session). These can therefore be saved and loaded from a **myQ2WDPrefs.txt** file that resides in the same directory as the loaded program (**myQ2WDPrefs.txt** is auto-loaded if it exists)

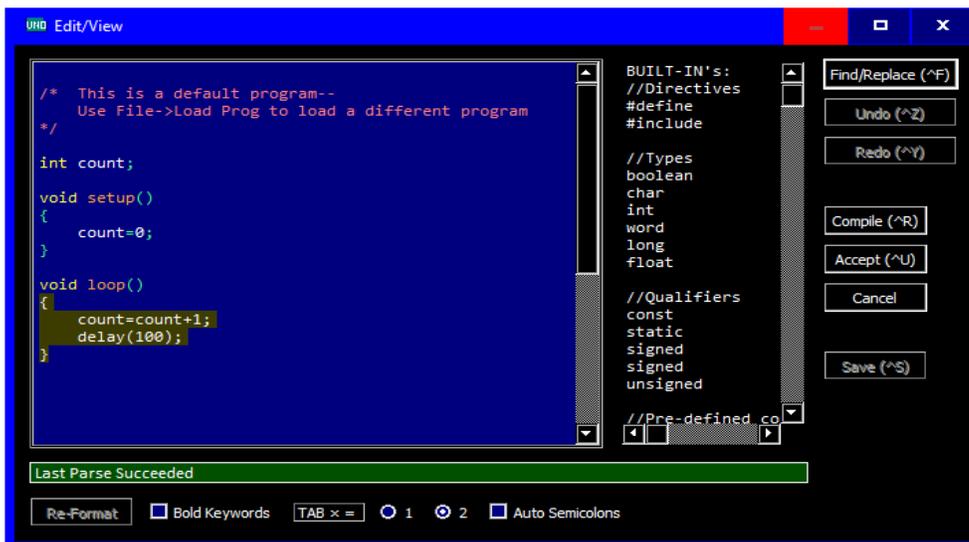
This dialog allows a choice between two mono-spaced fonts and three typeface sizes, as well as other miscellaneous user preferences.

Edit/View

By double-clicking on any line in the **Code Pane** (or using the menu **File**), an **Edit/View** window is opened to allow changes to your program file – it opens with the **currently selected line** in the **Code Pane** highlighted.

This window has full edit capability with dynamic syntax-highlighting (different highlight colours are used for C++ keywords, comments, etc.),. There is optional bold syntax highlighting, and automatic indent level formatting (assuming you have selected that using

Configure | Preferences). You can also conveniently select built-in function calls (or built-in '#define' constants) to be added into your program from the provided list-box – just double-click on the desired list-box item to add it to your program at the current caret position (function-call variable **types** are just for information and are stripped out to leave dummy placeholders when added to your program).



The window has **Find** (use **ctrl-F**) and **Find/Replace** capability (use **ctrl-H**). The **Edit/View** window has **Undo** (**ctrl-Z**), and **Redo** (**ctrl-Y**) buttons (which appear automatically).

Use ALT-right-arrow to request auto-completion choices for built-in **global variables**, and for **member variables and functions**.

To discard **all changes** you made since you first opened the program for editing, click the **Cancel** button. To accept the current state, click the **Accept** button and the program automatically receives another Parse (and is downloaded to the 'Uno' or 'Mega' if no errors are found) and the new status appears in the main UnoArduSim window **Status-Bar**.

A **Compile** (**ctrl-R**) button (plus an associated **Parse Status** message-box as seen in the image above) has been added to allow testing of edits without needing to first close the window. A **Save** (**ctrl-S**) button has also been added as a shortcut (equivalent to an **Accept** plus a later separate **Save** from the main window).

On either **Cancel** or **Accept** with no edits made, the **Code Pane** current line changes to become the **last Edit/View caret position**, and you can use that feature to jump the **Code Pane** to a specific line (possibly to prepare to do a **Run-To**), You can also use **ctrl-PgDn** and **ctrl-PgUp** to jump to the next (or previous) empty-line break in your program – this is useful for quickly navigating up or down to significant locations (like empty lines between functions). You can also use **ctrl-Home** and **ctrl-End** to jump to the program start, and end, respectively.

'Tab'-level automatic indent formatting is done when the window opens, if that option was set under **Configure | Preferences**. You can redo that formatting at any time by clicking the Re-Format button (it is only enabled if you have previously selected the **automatic indentation Preference**). You can also add or delete tabs yourself to a group of pre-selected consecutive lines using the keyboard **right-arrow** or **left-arrow** keys – but **automatic indentation Preference must be off** to avoid losing your own custom tab levels.

When **Auto Semicolons** is checked, pressing **Enter** to end a line automatically inserts the line-terminating semicolon.

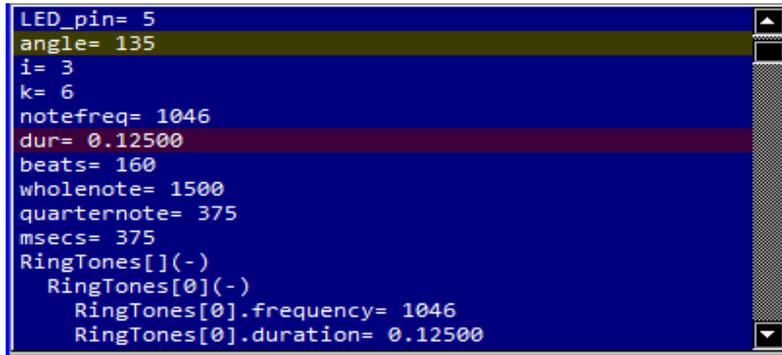
And to help you better keep track of your contexts and braces, clicking on a '{' or '}' brace **highlights all text between that brace and its matching partner**.

Variables Pane and Edit/Track Variable Window

The **Variables Pane** is located just below the **Code Pane**. It shows the current values for every user global and active (in-scope) local variable/array/object in the loaded program. As your program execution moves between functions, **the contents change to reflect only those local variables accessible to the current function/scope, plus any user-declared globals**. Any variables declared as 'const' or as 'PROGMEM' (allocated to 'Flash' memory) have values that cannot change, and to save space these are therefore *not displayed*.

'Servo' and 'SoftwareSerial' object instances contain no useful values so are, similarly, not displayed.

You can **find** specified **text** with the **Find** menu text-search commands (with **Tool-Bar** buttons  and , or keyboard shortcuts **up-arrow** and **down-arrow**), after first using **Find | Set Search** text or .



```
LED_pin= 5
angle= 135
i= 3
k= 6
notefreq= 1046
dur= 0.12500
beats= 160
wholenote= 1500
quarternote= 375
msecs= 375
RingTones[ ](-)
  RingTones[0](-)
    RingTones[0].frequency= 1046
    RingTones[0].duration= 0.12500
```

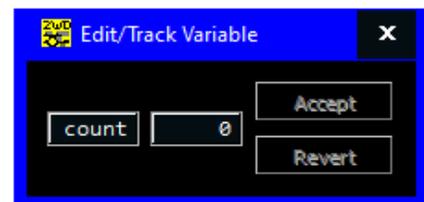
Arrays and **objects** are shown in either **un-expanded** or **expanded** format, with either a trailing plus ' (+) ' or minus ' (-) ' sign, respectively. The symbol for an array **x** shows as ' **x** [] ' . To expand it to show all elements of the array, just single-click on ' **x** [] (+) ' in the **Variables Pane**. To contract back to an un-expanded view, click on the ' **x** [] (-) ' . The un-expanded default for an object ' **p1** ' shows as ' **p1** (+) ' . To expand it to show all members of that ' **class** ' or ' **struct** ' instance, single-click on ' **p1** (+) ' in the **Variables Pane**. To contract back to an un-expanded view, single click on ' **p1** (-) ' .

If you **single-click on any line to highlight it in dark olive** (it can be simple variable, or the aggregate ' (+) ' or ' (-) ' line of an array or object, or an single array element or object-member), then doing a **Run-Till** will cause execution to resume and freeze at the next **write-access** anywhere inside that selected aggregate, or to that selected single variable location.

When using **Step** or **Run**, updates to displayed variable values are made according to user settings made under the menu **VarRefresh** – this allows a full range of behaviour from minimal periodic updates to full immediate updates. Reduced or minimal updates are useful to reduce CPU load and may be needed to keep execution from falling behind real-time under what would otherwise be excessive **Variables Pane** window update loads. When **Animate** is in effect, or if the **Highlight Changes** menu option is selected, changes to the value of a variable during **Run** will result in its displayed value being updated **immediately**, and it becomes highlighted in purple – this will cause the **Variables Pane** to scroll (if needed) to the line that holds that variable, and execution will no longer be real-time!.

When execution freezes after **Step**, **Run-To**, **Run-Till**, or **Run-then-Halt**, the **Variables Pane** highlights *in purple* the variable corresponding to the **address location(s) that got modified** (if any) by the **very last instruction** during that execution (including by variable declaration initializations) . If that instruction **completely** filled an **object or array**, the **parent (+) or (-) line** for that aggregate becomes highlighted. If, instead, the instruction modified a location that is currently visible, then it becomes highlighted. But if the modified location(s) is(are) currently hiding inside an un-expanded array or object, that aggregate **parent line** gets an **italic font highlighting** as a visual cue that something inside it was written to – clicking to expand it will then cause its **last** modified element or member to become highlighted.

This window also gives you **the ability to track any variable's value during execution**, or to **change its value in the middle of (halted) program execution** (to test what would be the effect of continuing on ahead with that new value). Halt execution first, then **left-double-click** on the variable whose value you wish to track or change. To simply monitor the value during program execution, you **leave the dialog open** and then do **Run**, **RunTo**, **RunTill**, or one of the **Step** commands – its value will be updated in **Edit/Track** according to the same rules that govern updates in the **Variables Pane**. **To change the variable's value**, fill in the right-hand Edit box, and **Adopt** the new value. Continue program execution (using any of the **Step** or **Run** commands) to use that new value from that point forward (you can **Revert** to the previous value if you change your mind before then)



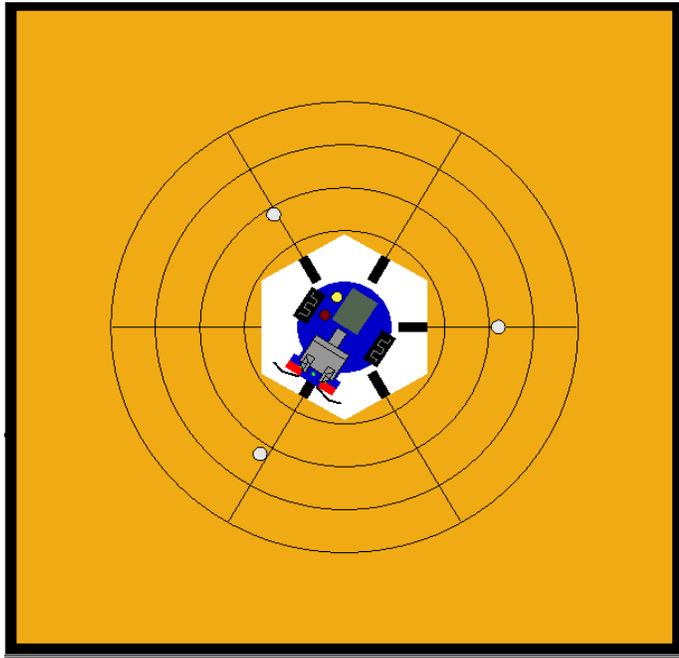
On program Load or Reset note that all un-initialized value-variables are reset to value 0, and all un-initialized pointer-variables are reset to 0x0000.

PlayField Pane

The PlayField Pane shows the 2WDBot in one of two virtual environments – a Pivot and Throw play-field, or the goal scoring competition Seek and Score play-field, depending on the user selection from the **PlayField** menu.

Pivot And Throw PlayField

This is a to-scale reproduction of the the real lab turntables, although in the simulator the turntable is flush with the floor. The **black tape strips** can be detected using the Q2WDBot's downward-looking reflective-infrared sensors. Bot motion is constrained to pure left or right rotations due to a central holding pin.



To “offer” a ball to the Bot, **left-click-and-hold** with the mouse pointer between the gripper jaws (before your program closes them). Hold the mouse left button down until the gripper grabs the ball, then release the mouse button (if you release too soon, the ball is taken back). As long as you click close enough to the center of the gripper jaws, Q2WDBotSim will position the ball so that it can be gripped centered between the jaws, and offset a bit toward the end of gripper jaws (you can position the gripper at any upward tilt angle to accept a ball).

Your program you can send “commands” to the Bot by accepting a single character typing into the Bluetooth communication “BT to 2WD” window Send-One edit box. Or your program can accept and interpret bumper-lever5 clicks to command actions.

The task is to move to the desired rotational position (tape line) and then to “throw” the grasped ball as far as possible – note that the ball can be thrown from the gripper using a combination of a rapid tilt-servo motion, with an appropriately-timed opening of the gripper jaws -- the ball's flight path is modelled accurately, and subsequent motion displayed.

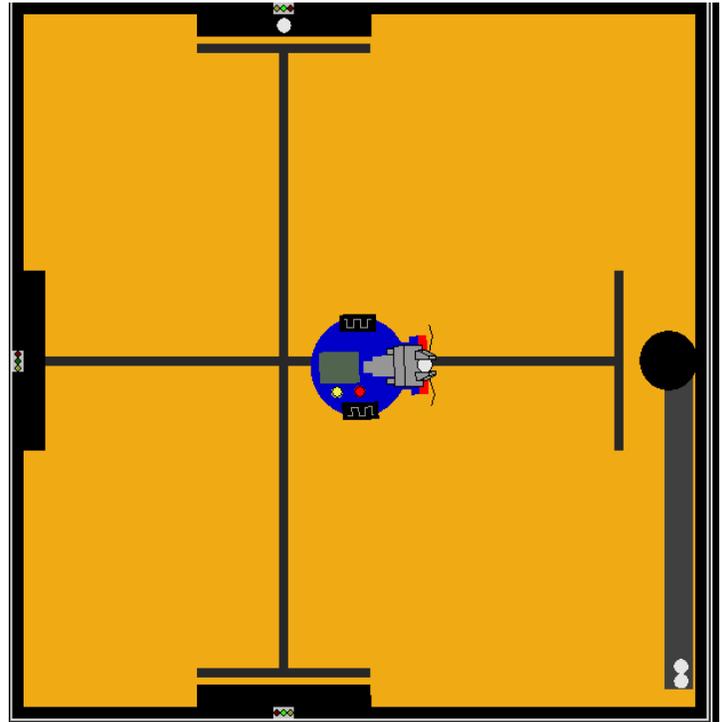
As an aid, Q2WDBotSim allows you to re-position the Bot using the mouse at any time: right-click anywhere ahead of the Bot and hold and drag to rotate the Bot to any desired angle. You can also simulate bumper switch closures at any time with a left-mouse-click. Click on the angled arm of the bumper switch to activate that bumper's contact sensor (and hold down if you want to keep the touch sensor contact closed) . To activate both bumper contacts simultaneously, click roughly half way between them.

Seek And Score PlayField

This is a to-scale reproduction of the the real lab competition playfield with walls, ball pick-up ledges and a goal mechanism. The **black-tape floor lines** (coloured here instead as grey for contrast with the ledges) can be detected using the Q2WDBot's downward-looking reflective-light sensors. Bot motion is unconstrained except by the walls, ledges, and goal apparatus .

Relevant Dimensions:

- PlayField is a 48 inches by 48 inches square
- Floor surface is 46.5 inches square
- Ball ledges are 12 inches long by 1.5 inches wide
- Central vertical tape-line is 18 inches from left wall
- Top and bottom horizontal tape lines are 2 inches from wall
- Goal vertical tape line is 5 inches from wall
- Balls are 1 inch (2.54 cm) in diameter
- Black-tape floor lines are 1.7 cm wide
- Top of a ball ledge is 1.25 cm below the gripper tilt-servo pivot axis
- Goal cup lip is 3 cm above the gripper tilt-servo pivot axis



To place a ball on one of the three pick-up ledges (and simultaneously activate the transmission of the 2-LED infrared homing beacon on the wall above it), left-click on that beacon (the small grey object with two LED's centered above each ledge). When enabled (its Green LED is ON), a beacon transmits its location at 300 baud (8N1) as the ASCII character value '0', '1', or '2' (corresponding to the top, left and bottom ledges, respectively). The single-character transmission is repeated after an idle gap of 300 milliseconds (the Red LED flashes on an off to remind you that transmissions have gaps between them). If **Options->Auto Beacons** is chosen, the beacon will automatically turn off its transmissions when the Bot (or the user via a mouse-click) picks up that beacon's ball.

The Bot must lower its gripper somewhat in order to be able to grasp a ledge ball, but lowering it too far will cause it the gripper jaws to fail to clear the ledge lip on approach. A ledge ball is initially stationary in a shallow bowl just in front of its beacon, but once contacted by the Bot gripper it will roll off the ledge if not successfully grasped, since the ledges have a shallow downward slope.

The goal apparatus consists of a 4-inch diameter cup (an ABS plastic drain fitting) feeding a short vertical chute leading to a collection-tube ramp. The Bot must raise its gripper to clear the lip of the goal cup.

A ball hitting the floor will land there (without bouncing) and its horizontal velocity will then decay rapidly. You can "pick up" any ball (once dropped by the gripper) by using the mouse (left-click, then release). You can then drop the picked up ball it anywhere by moving it with the mouse and then using either a *left*-click or a *right*-click: a left-click drops the ball from height; a right-click instead places the ball down onto the surface that is immediately below it (from which it may then roll off). A ball released from the Bot's gripper (or from your hand) will fall under gravitational acceleration, maintaining its current horizontal velocity until it strikes something. A ball striking a Bot surface or edge, or a ledge, or the goal edge, will bounce off. For better visualization, a ball moving upward has a white outline, while a ball moving downward, or stationary, has a black outline.

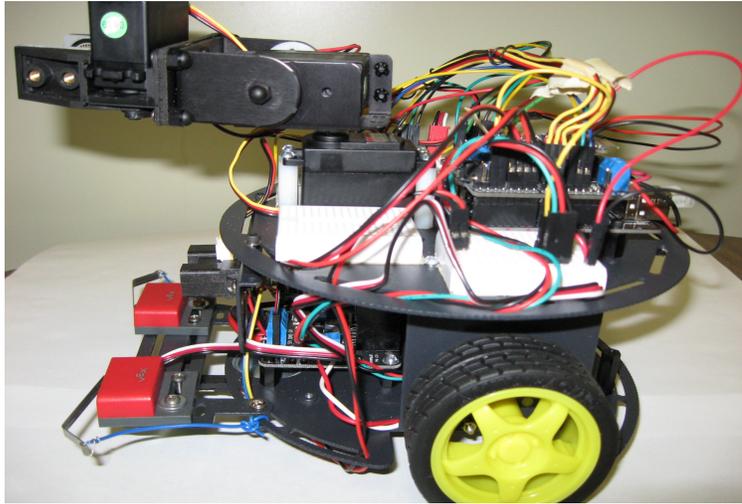
As an aid, Q2WDBotSim allows you to re-position the Bot using the mouse at any time: left click near the Bot's centre and **hold and drag to slide** the Bot to the desired location; or **right**-click **anywhere ahead of the Bot** and hold and drag to **rotate** the Bot to any desired angle. You can also **simulate collisions on either bumper at any time** with a left-mouse-click. Click on the angled arm of the VEX bumper switch to activate that bumper's contact sensor (and hold down if you want to keep the touch sensor contact closed) . To activate **both** bumper contacts simultaneously, click roughly half way between them.

Bot Information and Modelling

The Bot is based on the DFRobot 2WD robot kit (with pivot ball at the front), with the addition of a 3-servo pan/tilt/gripper mechanism. A photograph of the actual Bot is shown below:

Dimensions:

- Overall length = 23 cm (9 inches)
- Wheel base = 14 cm (5.5 inches)
- Wheel circumference = 20.5 cm (8 inches)
- Height of front bumpers = 4 cm (1.5 inches)
- Distance from wheel axes to front bumpers = 14 cm (5.5 inches)
- Distance from wheel axes to gripper tilt-servo pivot axis = 7 cm (2.75 inches)
- Gripper length (tilt-servo pivot axis to tip) = 9 cm (3.5 inches)
- Distance from wheel axes to line-tracker IR sensors = 12 cm (4.75 inches)
- Spacing gap between the three line-tracker IR sensors (Left, Centre, Right) = 2 cm (0.75 inches)
- Height of gripper tilt axis above floor = 14.5 cm (5.7 inches)



Wheel Motors:

- 1) These are DC gear motors driven by a dual motor driver with each driver having DIR and PWM pins.
- 2) A HIGH on the driver DIR pin creates forward vehicle motion on that wheel's side
- 3) The PWM pin will only respond properly in Q2WDBotS1m to `analogWrite()` or `tone()` generated signals, **not** to `digitalWrite()` "bit-banged" signals.
- 4) Maximum wheel rotation speed is about 1 revolution per second (at PWM duty cycle = 1.0, or 100% on-time)
- 5) These are high-inductance motors, and Arduino's 500 Hz `analogWrite()` signals is a bit too rapid for them. The result is that at lower duty cycles (below about 40% on-time), motor torque (and so speed) is reduced below expectations -- at duty-cycles less than about 20% on-time, the motors are almost stationary.

Servo Motors:

- 1) The three pan/tilt/gripper assembly servos are modelled after the HiTec -422 180-degree servo.
- 2) In Q2WDBotS1m the servo control pins **only respond accurately** to `Servo.write()` or `Servo.writeMicrosecond()`, **not to** `digitalWrite()` "bit-banged" signals.
- 3) Maximum servo rotation speed is 60 degrees in 140 milliseconds.
- 4) The pan servo is centered for a servo `write()` angle of 90 degrees.
- 5) The tilt servo is horizontal for a servo `write()` of 60 degrees,, and is maximally down-tilted at a `write()` angle of 0.
- 6) The gripper servo has the gripper jaws fully open for a servo `write()` angle of 30 degrees, and fully closed for a servo `write()` angle of 180 degrees.

Sensors:

These are connected to the desired pin using the **Config→Write Up Pins** dialog, as described in the next section.

IR Receiver

The forward-looking IR receiver is a standard modulated IR receiver as used in remote controls. It detects infrared transmissions from the three PlayField IR beacons, delivering a digital signal on its attached pin. This signal can be detected as a 300 baud 8N1 signal using the provided QSerial library functions (or you can use SoftwareSerial if you prefer) -- see the **Infrared Communications from Beacons to the Bot** section below.

VEX Bumper Switches

Each bumper sensor is a simple OPEN/CLOSED switch wired to a pull-up resistor to +5V to create a voltage that changes as the sensor is depressed. The bumper can be attached to a pin and sensed using `digitalRead()`. In Q2WDBotSim the switch is activated if it hits a wall or the goal collector tube, and you can also **manually** activate this sensor by clicking near it with the left mouse button (and holding it down for as long as you wish). QWDBotSim assumes that you have wired your real Bot so that when the VEX bumper switch closes, this will cause a LOW on the pin to which is attached.

Alternatively, encoded bumpers can combine the two bumper switch closures into a single analog voltage. The four possible switch combinations (CLOSED, CLOSED), (CLOSED, OPEN), (OPEN, CLOSED), (OPEN, OPEN) create corresponding analog levels that are modelled as 0, 255, 511, 767 in Q2WDBotSim.

Yellow Pushbutton Switch

This sensor is a simple OPEN/CLOSED switch that you can wire up to create a voltage that changes as the pushbutton is depressed. In Q2WDBotSim you can manually activate the pushbutton sensor by clicking on it with the left mouse button (and holding it down for as long as you wish).

If the pushbutton switch is wired to a pin and sensed using `digitalRead()`, a push will create a logic LOW, else if unpressed it will create a logic HIGH.

Wheel Encoders

The wheel encoders have 10 teeth per revolution, which interrupt an infrared beam and so so will create 20 digital-level changes per wheel revolution on the attached digital pin (every second change is from LOW to HIGH, and the others are from HIGH to LOW).

LFM Line Tracker

The downward-looking LFM Line Tracker produces 3 analog output voltages from its three active reflective IR sensors. The produced voltage when over a dark line is around 1.5 volts, and when over the lighter floor background, it is around 2.5 volts. These levels are BOTH below the logic HIGH threshold for a digital input (which is closer to 3 volts), so we cannot differentiate these as digital levels through `digitalRead()`. Instead we must read the levels using `analogRead()`

In Q2WDBotSim, the modelled `analogRead()` level is 600 over the lighter floor background. and 400 when over the dark floor lines.

IR Ranger

This forward-looking IR range sensor measures distance and produces an **analog voltage** that is non-linear with range to the wall in front of the Bot.. The output voltage is 0.4 volts at a range of 80 cm, rises in an upward steepening curve until it reaches a maximum of 2.6 volts at a range of 8 cm, and then starts to drop rapidly as the range decrease below 8 cm.

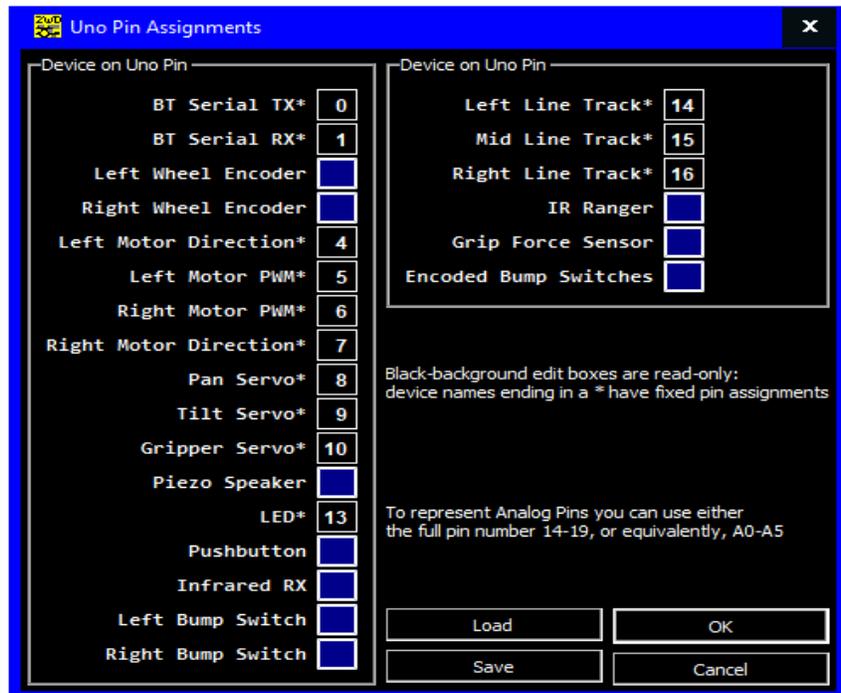
Grip Force

The grip force sensor measures ball contact force and produces a varying resistance. Q2WDBotSim assumes this sensor it is wired to the attached pin as a pull-up resistor (whose value will change non-linearly with the grip-force encountered from a maximum of about 500 K-ohms with no pressure to a minimum of about 1K-ohm at max grip force), with an attached fixed pull-down resistor of 10 K-ohms. This yields a pin voltage that ranges from almost a full 5 volt HIGH to almost zero volts.

Attaching Pins to Motors and Sensors

The **Configure→Wire Up Pins** menu item can be used to open a dialog to allow you to choose (several) of the sensor connections to the available subset of the Bot's 'Uno' board's 20 pins (some pins are for *fixed* purposes, and these are in black read-only boxes) . From this dialog you can also Save Pins to a text file, and/or Load Pins from a previously saved (or edited) text file.

Note that digital pins 0-13 are specified as 0-13 but analog pins 0-5 map to Uno pins 14-19.



The Uno Pins Window

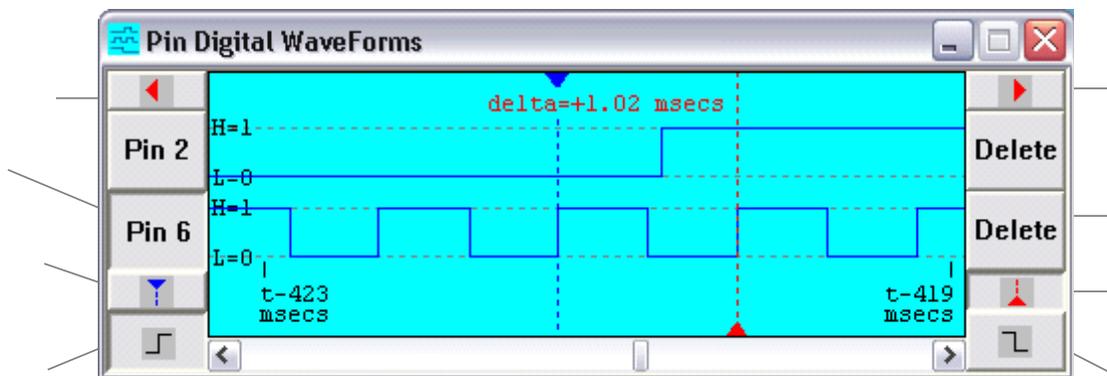
This is a depiction of the pin levels on the microcontroller board during program execution. When you load a new program into Q2WDBotSim, if it successfully parses it undergoes a "simulated download" to the 'Uno' that mimics the way an actual board behaves— you will see the levels on pins 0 and 1 toggling (pins 1 and 0 which are *hard-wired for serial communication with the Bluetooth adaptor*). This is immediately followed by a pin 13 flash that signifies board reset and Q2WDBotSim automatic halt at the beginning of your loaded program's execution. The window allows you to visualize the digital logic levels on all 20 'Uno' pins ('1' on red for HIGH, '0' on blue for LOW, and '?' on grey for an undefined indeterminate voltage), and programmed directions ('I' for INPUT, "O" for OUTPUT). For pins that are being pulsed using PWM via `analogWrite()`, or by `Tone()`, or by `Servo.write()`, the colour changes to purple and the displayed symbol becomes '^'.



Clicking on any of the pins can be done to open (or add to) either a **Pin Digital Waveforms** window or a **Pin Analog Waveform** window as shown on the next page-- both display the past one-second's worth of activity on that pin, as described below.

Left-clicking on any 'Uno' pin will open a **Pin Digital Waveforms** window that displays the past one-second's worth of **digital-level activity** on that pin. You can click on other pins to add these to the Pin Digital Waveforms display (to a maximum of 4 waveforms at any one time).

To **ZOOM IN** and **ZOOM OUT** (zoom is always centered on the **ACTIVE** cursor), use the mouse wheel, or keyboard shortcuts **CTRL-up_arrow** and **CTRL-down_arrow**.

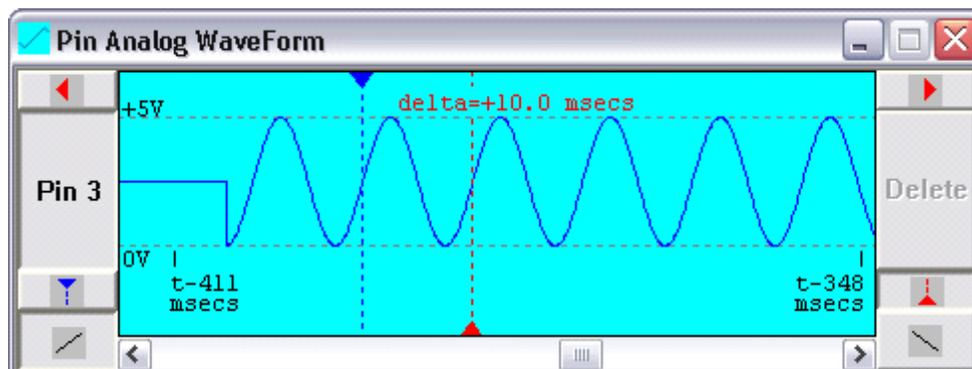


One of the displayed waveforms will be the **active pin waveform**, indicated by its "Pin number" button being shown as depressed (for example Pin 6 is active in the above Pin Digital WaveForms screen capture). You can select a waveform by clicking its Pin number button, and then select the edge-polarity of interest by clicking the appropriate rising/falling edge-polarity selection button, , or , or by using the shortcut keys **uparrow** and **downarrow**. You can then *jump-position* the active cursor (either blue or red cursor lines with their delta time shown) backward or forward to the chosen-polarity digital edge of *this active pin waveform* by using the forward/backward-to-edge arrow buttons (,  and ,  depending on activated cursor), or the keyboard shortcuts **leftarrow** and **rightarrow**.

To activate a cursor, click its coloured activation button as shown above – *this also jump-scrolls the view to that cursor's current location*. Alternatively, you can quickly alternate activation between cursors (with their respectively-centred views) using the shortcut **TAB** key.

You can jump-position the currently activated cursor by **left-clicking anywhere** in the on-screen waveform view region. Alternatively, you can select either the red or blue cursor line by clicking right on top of it (to activate it), then *drag it to a new location*, and release. When a desired cursor is currently somewhere off-screen, you can **right-click anywhere** in the view to jump it to that new on-screen location. If both cursors are already on-screen, right-clicking simply alternates between activated cursor.

Doing instead a **right-click on any 'Uno' pin** opens a **Pin Analog Waveform** window that displays the past one-second's worth of **analog-level activity** on that pin. Unlike the Pin Digital WaveForms window, you can only display one pin's worth of analog activity at any one time.



You can *jump-position* blue or red cursor lines to the next rising or falling "slope point" by using the forward/backward arrow buttons (,  or , , again depending on activated cursor, or **leftarrow** or **rightarrow**) in concert with the rising/falling slope selection buttons ,  (the "slope point" occurs where the analog voltage passes through the 'Uno' pin's high-digital-logic-level threshold). Alternatively, you can again click-to-jump, or drag these cursor lines similar to their behaviour in the Pin Digital WaveForms window

Communications with the Bot

Infrared Communication from Beacons to the Bot

The provided IR receiver sensor is tuned to 40 kHz infrared transmission of wavelength 900 nanometers. This is the same frequency and wavelength used by three infrared beacon transmitters located above the PlayField ball ledges. These three beacons transmit ASCII value '0', '1', and '2', respectively, and their data can be received using the `SoftwareSerial` library's `receive()` function. The beacons transmit IR for a LOW, so data arrive non-inverted at the output of the Bot's infrared receiver. Declare your `SoftwareSerial` program object to use *non-inverted* reception (the default), and the baud rate set by `begin()` must be 300 in order to match the baud rate of the beacons.

The beacons are activated/deactivated by *single*-clicking on them (the Green LED turns ON) and disabled by another *single*-click. The Red LED flashes when the beacon is enabled as a reminder of the gaps between subsequent characters.

Bluetooth Communications with the Bot

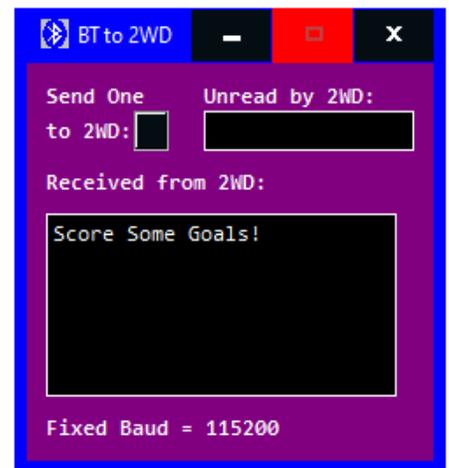
The DFRobot Bluetooth module allows communication with an external Bluetooth device over **serial pins 0 and 1**. The BT Comm window does not allow a choice of choice of baud rate – that is fixed at 115200 to match that of the real DFRobot Bluetooth adaptor module.

Typing a character in its “Send One to 2WD” edit box causes it be transmitted *immediately* at the chosen baud rate (assuming you program is running, and so time is advancing).

Characters transmitted in the BT Monitor window can be tested by `Serial.available()` and read using `Serial.read()`, because the built-in `Serial` object is associated with pins 0 and 1. In addition, you can call `Serial.flush()` at any time to flush any characters from its system-controlled receive queue.

Values returned by `read()` are the actual ASCII numeric byte codes. Although lower-case letters are accepted in the “Send One to 2WD” edit box, *only their capitalized versions are* transmitted, and only letter keys are allowed. You can only send at a (deliberately limited) maximum rate of about 3 characters each second (a new character typed before waiting for an imposed 300 millisecond gap will be ignored). Any characters fully received on pin 0 but that have not yet been read by your program will appear in the “Unread by 2WD” edit box.

All characters received from the Bot, for example by `Serial.print()` or `Serial.write()` function calls are displayed in the “Received from 2WD” edit box.



Menus

File menu commands:

| | |
|---|---|
| Load INO or PDE Prog (ctrl-L)  | Allows the user to choose a program file having the selected extension. The program is immediately parsed |
| Edit/View (ctrl-E) | Opens the loaded program for viewing/editing. |
| Save (ctrl-S)  | Save the edited program contents back to the original program file. |
| Save As | Save the edited program contents under a different file name. |
| Next (#include) file  | Advances the CodePane to display the next #include'd file |
| Previous  | Returns the CodePane display to the previous file |
| Exit | Exits Q2WDBotSim. |

PlayField menu commands:

| | |
|------------------------|---|
| Pivot and Throw | Chooses the Bot mounted on a turntable with black tape lines. |
| Seek and Score | Opens the Bot on the competition playfield with goal, ledges with infrared beacons, and the ability to click a beacon to enable it and add a ball on its ledge. |

Find menu commands:

| | |
|--|--|
| Ascend Call Stack  | Jump to the previous caller function in the call-stack – the Variables Pane will adjust to show that functions locals |
| Descend Call Stack  | Jump to the next called function in the call-stack – the Variables Pane will adjust to show that functions locals |
| Set Search Text (ctrl-F)  | Activate toolbar Find edit box to define your next-to-be-searched-for text.. |
| Find Next Text  | Jump to the next Text occurrence in the Code Pane (if it has the active focus), or to the next Text occurrence in the Variables Pane (if instead it has the active focus). |
| Find Previous Text  | Jump to the previous Text occurrence in the Code Pane (if it has the active focus), or to the previous Text occurrence in the Variables Pane (if instead it has the active focus). |

Execute menu commands:

| | | |
|------------------------------|---|--|
| <u>Step Into (F2)</u> |  | Steps execution forward by one instruction, or <i>into a called function</i> . |
| <u>Step Over (F4)</u> |  | Steps execution forward by one instruction, or <i>by one complete function call</i> . |
| <u>Step Out Of</u> |  | Advances execution by <i>just enough to leave the current function</i> . |
| <u>Run To</u> |  | Runs the program, <i>halting at the desired program line</i> -- you must first click to highlight a desired program line before using Run To. |
| <u>Run</u> |  | Runs the program. |
| <u>Halt</u> |  | Halts program execution (<i>and freezes time</i>). |
| <u>Reset</u> |  | Resets the program (all value-variables are reset to value 0, and all pointer variables are reset to 0x0000). |
| <u>Animate</u> | | Automatically steps consecutive program lines <i>with added artificial delay</i> and highlighting of the current code line. Real-time operation and sounds are lost. |
| <u>Slow Motion</u> | | Slows time by a factor of 10. |

Help menu commands:

| | |
|-----------------------------------|---|
| <u>Quick Help File</u> | Opens the Q2WDBotSim_QuickHelp PDF file. |
| <u>Full Help File</u> | Opens the Q2WDBotSim_FullHelp PDF file. |
| <u>Bug Fixes</u> | View significant bug fixes since the previous release.. |
| <u>Change/Improvements</u> | View significant changes and improvements since the previous release. |
| <u>About</u> | Displays version, copyright. |

Options menu commands:

| | |
|---|---|
| <u>Skip through Structures/Operators</u> | While stepping, do not not stop execution inside a con/de/structor function, or inside an operator function. |
| <u>Register-Allocation Modelling</u> | Model how the real Arduino compiler would allocate variables between registers and the stack. |
| <u>Error on Uninitialized</u> | Flag as a Parse error anywhere your program attempts to use a variable without having first initialized its value. |
| <u>Artificial loop() Delay</u> | Adds 1 millisecond of delay every time <code>loop()</code> is called (in case there are no other program <code>delay()</code> 's anywhere). |
| <u>Auto Beacons</u> | If chosen, beacons will automatically stop their transmission when the Bot picks up the ball on their ledge. |
| <u>Allow Nested Interrupts</u> | Allow <code>interrupts()</code> to be called inside a user interrupt routine (to re-nable interrupts inside that routine). |

Configure menu commands:

| | |
|------------------------------------|--|
| <u>Wire Up Pins</u> | Opens a dialog to allow you to set on which pins you will attach sensors (that allow such freedom of choice – many are on fixed pins) which should match the real-life hardware connections you have chosen on your Bot. From this dialog you can also Save pin connections to a text file, and/or Load connections from a previously saved (or edited) text file. |
| <u>Preferences</u> | Set compilation, text size, and other preferences which will be automatically saved into file <code>myQ2WDPrefs.txt</code> . That preferences in this file are automatically loaded at each launch of Q2WDBotSim . |
| <u>Wheel Speed Mismatch</u> | Opens a dialog to allow you to model real-world differences in the speed response of your Bot;s left and right motors (real-world motors are always slightly different). |

VarUpdates menu commands:

| | |
|---------------------------------------|---|
| <u>Allow Auto (-) Collapse</u> | Allow Q2WDBotSim to collapse displayed expanded arrays/structs/objects when falling behind real-time. |
| <u>Minimal</u> | Only refresh the variables Pane display 4 times per second. |
| <u>Highlight Changes</u> | Highlight the last-changed variable value (can cause slowdown). |

Windows menu commands:

| | |
|---------------------------------|---|
| <u>BT Monitor</u> | Restores (if minimized) the BT monitor window for communication through 'Uno' pins 0 and 1 when the Bluetooth adaptor is connected on your Bot. |
| <u>'Uno' Pins</u> | Restores (if minimized) the 'Uno' Pinsr window thatshows pin activity on all 20 pins. |
| <u>Restore All</u> | Restores all minimized windows. |
| Prompt | To left-Click or Right-Click an 'Uno' Pin to create a Waveform window: |
| <u>Digital Waveforms</u> | Restore a minimized Pin Digital Waveforms window. |
| <u>Analog Waveform</u> | Restore a minimized Pin Analog Waveform window. |

Timing and Sound Modelling

Intro

The 'Uno' pins and attached sensors and motors are all modelled electrically, and you will be able to get a good idea at home of how your programs will behave with the actual hardware.

Timing

Q2WDBotSim executes rapidly enough on a PC or tablet that it can (*in the majority of cases*) model program actions in real-time, **but only if your program incorporates** at least some small `delay()` calls or other calls that will naturally keep it sync'd to real time (see below).

To accomplish this, Q2WDBotSim makes use of a Windows callback timer function, which allows it to keep accurate track of real-time. The execution of a number of program instructions is simulated during one timer slice, and instructions that require longer execution (like calls to `delay()`) may need to use multiple timer slices. Each iteration of the callback timer function corrects system time using the system hardware clock so that program execution is constantly adjusted to keep in lock-step with real-time. *The only times execution rate must fall behind real-time* is most often when the user has tight loops **with no added delay** or that makes very frequent changes to variables values in a tight loop (causing a Variables Pane update overload). In addition, programs with large arrays being displayed, or having tight loops **with no added delay** can cause a high function call frequency and generate a high Variables Pane display update load causing it to fall behind real-time— this can be circumvented by allowing update reductions in the **VarUpdates** menu, or by selecting **Minimal Updates** there when necessary. Any computational overload is compensated for by skipping some timer intervals to compensate, and this would slow down program progression to below real-time.

Accurately modelling the sub-millisecond execution time for each program instruction or operation **is not done** – only very rough estimates for most have been adopted for simulation purposes. However, the timing of `delay()`, and `delayMicroseconds()` functions, and functions `millis()` and `micros()` are all perfectly accurate, and **as long as you use at least one of the delay functions** in a loop somewhere in your program, or you use a function that naturally ties itself to real-time operation (like `print()` which is tied to the chosen baud rate), then your program's simulated performance will be very close to real-time (again, barring excessive user-allowed Variables updates which could slow it down).

In order to see the effect of individual program instructions *when running*, it may be desirable to be able to slow things down. A time-slowdown factor of 10 can be set by the user under the Options menu.

Sounds modelling

The PIEZO device produces sound corresponding to the electrical level changes occurring on the attached pin, regardless of the source of such changes. Q2WDBotSim starts and stops an associated sound buffer as execution is started/halted to keep the sound buffer synchronized to program execution.

Limitations and Unsupported Elements

Included Files

A '<>' - bracketed '#include' of '<Servo.h>', '<SoftwareSerial.h>', and '<EEPROM.h>' is supported but these are only emulated – the actual files are not searched for; instead their functionality is directly "built into" Q2WDBotSim, and are valid for the fixed supported Arduino version.

Any quoted '#include' (for example of "supp.ino", "myutil.cpp", or "mylib.h") is supported, but all such files must **reside in the same directory as the parent program file** that contains their '#include' (there is no searching done into other directories). The '#include' feature can be useful for minimizing the amount of program code shown in the **Code Pane** at any one time. Header files with '#include' (i.e. those having a ".h" extension) will additionally cause the simulator to attempt including the same-named file having a ".cpp" extension (if it also exists in the directory of the parent program).

Dynamic Memory allocations and RAM

Operators 'new' and 'delete' are supported, as are native Arduino 'String' objects, **but not direct calls to** 'malloc()', 'realloc()' and 'free()' that these rely on.

Excessive RAM use for variable declarations is flagged at Parse time, and RAM memory overflow is flagged during program execution. An item on menu **Options** allows you to emulate the normal ATmega register allocation as would be done by the AVR compiler, or to model an alternate compilation scheme that uses the stack only (as a safety option in case a bug pops up in my register allocation modeling). If you were to use a pointer to look at stack contents, it should accurately reflect what would appear in an actual hardware implementation.

'Flash' Memory Allocations

'Flash' memory 'byte', 'int' and 'float' variables/arrays and their corresponding read-access functions are supported. Any 'F()' function call ('Flash' -macro) of any literal string is supported, but the only supported 'Flash' -memory string direct-access functions are 'strcpy_P()' and 'memcpy_P()', so to use other functions you will need to first copy the 'Flash' string to a normal RAM 'String' variable, and then work with that RAM 'String'. When you use the 'PROGMEM' variable-modifier keyword, it must appear **in front of** the variable name, and that variable **must also be declared** as 'const'.

'String' Variables

The native 'String' library is almost completely supported with a few very (and minor) exceptions .

The 'String' operators supported are +, +=, <, <=, >, >=, ==, !=, and []. Note that: 'concat()' takes a **single** argument which is the 'String', or 'char', or 'int' to be appended to the original 'String' object, **not** two arguments as is mistakenly stated on the Arduino Reference web pages).

Arduino Libraries

Only 'SoftwareSerial.h', 'Servo.h', and 'EEPROM.h' for the **Arduino V1.8.8** release are supported in Q2WDBotSim . Trying to '#include' the ".cpp" and ".h" files of other as-yet unsupported libraries will **not work** as they will contain low-level assembly instructions and unsupported directives and unrecognized files!

Pointers

Pointers to simple types, arrays, or objects are all supported. A pointer may be equated to an array of the same type (e.g. `'iptr = intarray'`), but then there would be *no subsequent arrays bounds checking* on an expression like `'iptr[index]'`.

Functions can return pointers, or `'const'` pointers, but any subsequent level of `'const'` on the returned pointer is ignored.

There is *no support* for function calls being made through *user-declared function-pointers*.

'class' and 'struct' Objects

Although poly-morphism, and inheritance (to any depth), is supported, a `'class'` or `'struct'` can only be defined to have at most **one** base `'class'` (i.e. *multiple*-inheritance is not supported). Base-`'class'` constructor initialization calls (via colon notation) in constructor declaration lines are supported, but *not* member-initializations using that same colon notation. This means that objects that contain `'const'` non-`'static'` variables, or reference-type variables, are not supported (those are only possible with specified construction-time member-initializations)

Copy-assignment operator overloads are supported along with move-constructors and move-assignments, but user-defined object-conversion ("type-cast") functions are not supported.

Scope

There is no support for the `'using'` keyword, or for `'namespace'`, or for `'file'` scope. All non-local declarations are by implementation assumed to be global.

Any `'typedef'`, `'struct'`, or `'class'` definition (i.e. that may be used for future declarations), must be made *global* scope (*local* definitions of such items inside a function are not supported).

Qualifiers 'unsigned', 'const', 'volatile', 'static'

The `'unsigned'` prefix works in all the normal legal contexts. The `'const'` keyword, when used, must *precede* the variable name or function name or `'typedef'` name that is being declared – placing it after the name will cause a Parse error. For function declarations, only pointer-returning functions can have `'const'` appear in their declaration.

All Q2WDBotSim variables are `'volatile'` by implementation, so the `'volatile'` keyword is simply ignored in all variable declarations. Functions are not allowed to be declared `'volatile'`, nor are function-call arguments.

The `'static'` keyword is allowed for normal variables, and for object members and member-functions, but is explicitly disallowed for object instances themselves (`'class'` / `'struct'`), for non-member functions, and for all function arguments.

Compiler Directives

`'#include'` and regular `'#define'` are both supported, but *not function-macro* `'#define'`. As of V2.9, conditional compilation is supported via the `'#ifdef'`, `'#ifndef'`, `'#else'`, and `'#endif'` directives (*but not* `'#if'` or `'#elif'`) – *nesting of these directives is also not supported*. The directives `'#pragma'`, `'template'`, `'#line'`, `'#error'` and predefined macros (like `'_LINE_'`, `'_FILE_'`, `'_DATE_'`, and `'_TIME_'`) are also *not supported*.

Arduino-language elements

All native Arduino language elements are supported with the exception of the dubious `'goto'` instruction (the only reasonable use for it I can think of would be as a jump (to a bail-out and safe shutdown endless-loop) in the event of an error condition that your program cannot otherwise deal with)

C/C++-language elements

Bit-saving "bit-field qualifiers" for members in structure definitions are ***not supported***.

`'union'` is ***not supported***.

The oddball "comma operator" is ***not supported*** (so you cannot perform several expressions separated by commas when only a single expression is normally expected, for example in `'while()'` and `'for(; ;)'` constructs).

Function Templates

User-defined functions that use the keyword "template" to allow it to accept arguments of "generic" type are ***not supported***.

Real-Time Emulation

As noted above, execution times of the many different individual possible Arduino program instructions are ***not*** modeled accurately, so that in order to run at a real-time rate your program will need some sort of dominating `'delay()'` instruction (at least once per `'loop()'`), or an instruction that is naturally synchronized to real-time pin-level changes (such as, `'pulseIn()'`, `'shiftIn()'`, `'Serial.read()'`, `'Serial.print()'`, `'Serial.flush()'` etc.).

See **Timing** and **Sounds** above for more detail on limitations.

Release Notes

Bug Fixes

V3.1 June 2021 – Frirst Release

Changes/Improvements

V3.1 – June 2021 Furst Release