

UnoArduSimV2.8.2 Full Help

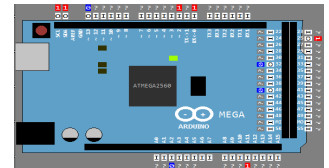


Table of Contents

[Overview](#)

[Code Pane, Preferences, and Edit/View](#)

[Code Pane](#)

[Preferences](#)

[Edit/View](#)

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

[Lab Bench Pane](#)

[The 'Uno' or 'Mega'](#)

['I/O' Devices](#)

[Serial Monitor \('SERIAL'\)](#)

[Alternate Serial \('ALTSER'\)](#)

[SD Disk Drive \('SD_DRV'\)](#)

[TFT Display \('TFT'\)](#)

[Configurable SPI Slave \('SPISLV'\)](#)

[Two-Wire I2C Slave \('I2CSLV'\)](#)

[Text LCD I2C \('LCDI2C'\)](#)

[Text LCD SPI \('LCDSPI'\)](#)

[Text LCD D4 \('LCD_D4'\)](#)

[Multiplexer LED I2C \('MUXI2C'\)](#)

[Multiplexer LED SPI \('MUXSPI'\)](#)

[Expansion Port SPI \('EXPSPi'\)](#)

[Expansion Port I2C \('EXPI2C'\)](#)

['1-Wire' Slave \('OWIISLV'\)](#)

[Shift Register Slave \('SRSLV'\)](#)

[Programmable 'I/O' Device \('PROGIO'\)](#)

[One-Shot \('1SHOT'\)](#)

[Digital Pulser \('PULSER'\)](#)

[Analog Function Generator \('FUNCGEN'\)](#)

[Stepper Motor \('STEPR'\)](#)

[Pulsed Stepper Motor \('PSTEPR'\)](#)

[DC Motor \('MOTOR'\)](#)

[Servo Motor \('SERVO'\)](#)

[Piezo Speaker \('PIEZO'\)](#)

[Slide Resistor \('R=1K'\)](#)

[Push Button \('PUSH'\)](#)

[Coloured LED \('LED'\)](#)

[4-LED Row \('LED4'\)](#)

[7-Segment LED Digit \('7SEG'\)](#)

[Analog Slider](#)

[Pin Jumper \('JUMP'\)](#)

[Menus](#)

[File:](#)

[Load INO or PDE Prog \(ctrl-L\)](#)

[Edit/View \(ctrl-E\)](#)

[Save](#)

[Save As](#)

[Next \('#include'\)](#)

[Previous](#)

[Exit](#)

Find:

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

Execute:

- [Step-Into \(F4\)](#)
- [Step-Over \(F5\)](#)
- [Step-Out-Of \(F6\)](#)
- [Run-To \(F7\)](#)
- [Run-Till \(F8\)](#)
- [Run \(F9\)](#)
- [Halt \(F10\)](#)
- [Reset](#)
- [Animate](#)
- [Slow Motion](#)

Options:

- [Step-Over Structors/Operators](#)
- [Register-Allocation](#)
- [Error on Uninitialized](#)
- [Added 'loop\(\)' Delay](#)
- [Allow Nested Interrupts](#)

Configure:

- ['I/O' Devices](#)
- [Preferences](#)

VarRefresh:

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

Windows:

- [Serial Monitor](#)
- [Restore All](#)
- [Pin Digital Waveforms](#)
- [Pin Analog Waveform](#)

Help:

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

['Uno' or 'Mega' Board and 'I/O' Devices](#)

[Timing](#)

['I/O' Device Timing](#)

[Sounds](#)

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 – from V2.5 onward](#)

[Bug Fixes](#)

[V2.8.2 September 2020](#)

[V2.8.1– June 2020](#)

[V2.8.0– June 2020](#)

[V2.7– Mar. 2020](#)

[V2.6.0– Jan 2020](#)

[V2.5.0– Oct 2019](#)

[V2.4– May 2019](#)

[V2.3– Dec. 2018](#)

[V2.2– Jun. 2018](#)

[V2.1.1– Mar. 2018](#)

[V2.1– Mar. 2018](#)

[V2.0.2 Feb. 2018](#)

[V2.0.1– Jan. 2018](#)

[V2.0– Dec. 2017](#)

[Changes/Improvements](#)

[V2.8.2- Septemver 2020](#)

[V2.8.0- June 2020](#)

[V2.7 Mar. 2020](#)

[V2.6.0 Jan. 2019](#)

[V2.5.0 Oct 2019](#)

[V2.4 May 2019](#)

[V2.3 Dec. 2018](#)

[V2.2 Jun. 2018](#)

[V2.1 Mar. 2018](#)

[V2.0.1 Jan. 2018](#)

[V2.0 Sept. 2017](#)

Overview

UnoArduSim is a freeware **real-time** (see for Timing **restrictions**) simulator tool that I have developed for the student and Arduino enthusiast. It is designed to allow you to experiment with, and to easily debug, Arduino programs **without the need for any actual hardware**. It is targeted to the **Arduino 'Uno' or 'Mega'** board, and allows you to choose from a set of virtual 'I/O' devices, and to configure and connect these devices to your virtual 'Uno' or 'Mega' in the **Lab Bench Pane**. – you do not need to worry about wiring errors, broken/loose connections, or faulty devices messing up your program development and testing.

UnoArduSim 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 **Step** operations 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 ATmega RAM overflow will be detected (and the culprit program line highlighted!). Any electrical conflicts with attached 'I/O' devices are flagged and reported as they occur.

When an INO or PDE program file is opened, it is loaded into the program **Code Pane**. The program is then given a Parse, to transform it 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 UnoArduSim 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 a mis-matched baud-rate) are reported on the Status-Bar, and via a pop-up message-box.

UnoArduSim V2.7 is a substantially complete implementation of the **Arduino Programming Language V1.8.8** as **documented at the arduino.cc**. Language Reference web page, and with additions as noted in the version Download page Release Notes. Although UnoArduSim does not support the full C++ implementation that the Arduino.exe 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++ features for Arduino hobbyists and students – for example, '**enum**' and '**#define**' are supported, but function-pointers are not. Even though user-defined objects ('**class**' and '**struct**') and (most) operator overloads are supported, *multiple inheritance is not*.

Because UnoArduSim 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, **ATmega328 registers are not accessible to your program** for either reading or writing, although register allocation, passing and return are emulated (it you choose that under the menu **Options**).

As of V2.6, UnoArduSim has built-in automatic support for a limited subset of the Arduino provided libraries, these being: '**Stepper.h**', '**Servo.h**', '**SoftwareSerial.h**', '**SPI.h**', '**Wire.h**', '**OneWire.h**', '**SD.h**', '**TFT.h**', and '**EEPROM.h**' (version 2). V2.6 introduces a mechanism for 3rd party library support via files provided in the '**include_3rdParty**' folder that can be found inside the UnoArduSim install directory. For any '**#include**' of other (i.e. user-created) libraries, UnoArduSim 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 you are working on (subject of course to the limitation that the contents of any '**#include**' file must be fully understandable to the UnoArduSim parser).

I developed UnoArduSimV2.x in QtCreator with multi-language support, and it is currently only available for Windows™. Porting to Linux or MacOS, is a project for the future! UnoArduSim grew out of simulators I had developed over the years for courses I taught at Queen's University, and it 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 unoArduSim@gmail.com and **be sure to attach your full bug-inducing program Arduino source code** so I can replicate the bug and fix it. I will not be replying to individual bug reports, and I have no guaranteed timelines for fixes in a subsequent release (remember there are almost always workarounds!).

Cheers,

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



Code Pane, Preferences, and Edit/View


(Aside: The sample windows shown below are all under a user-chosen Windows-OS colour theme that has a dark blue window background color).



Code Pane

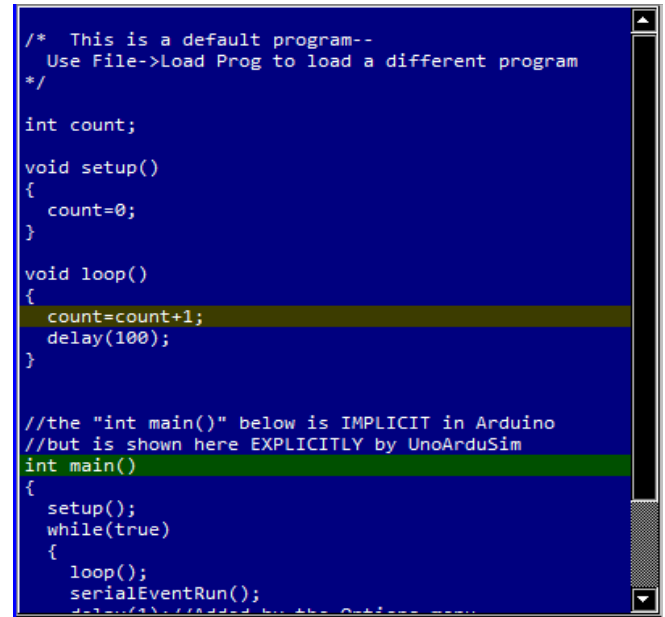
The **Code Pane** displays your user program, and **green** highlighting tracks its execution. (or highlights **red** for an error)

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 **, , or**), 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  and ). 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;

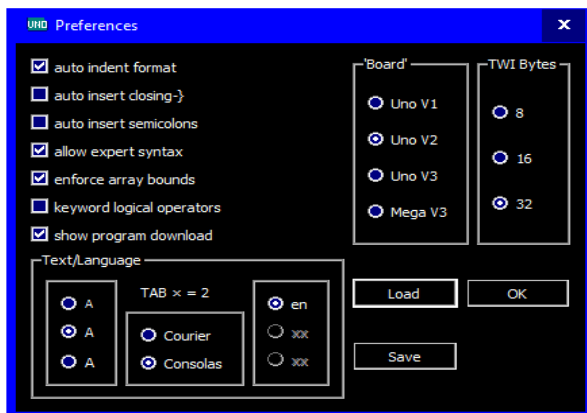
void setup()
{
  count=0;
}

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

Preferences



Configure | Preferences allows users to set program and viewing preferences (that a user will normally wish to adopt at the next session). These can therefore be saved and loaded from a '**myArduPrefs.txt**' file that resides in the same directory as the loaded 'Uno' or 'Mega' program ('**myArduPrefs.txt**' is automatically loaded if it exists).

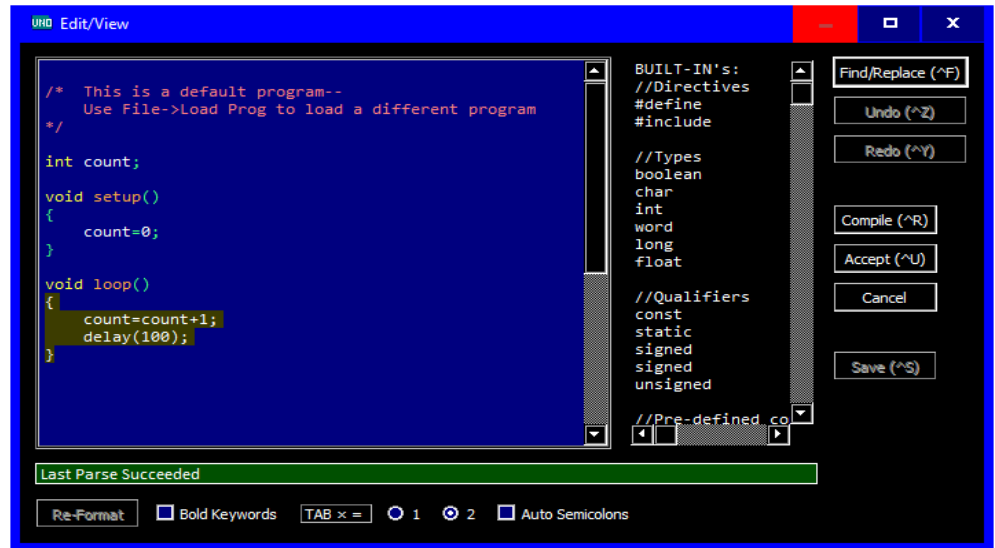
This dialog-box allows a choice between two mono-spaced fonts and three type sizes, and other miscellaneous preferences. As of V2.0, language choice is now included. – this always include English (**en**), plus one or two other user locale languages (where these exist) , and one override based on the two-letter ISO-639 language code on the very first line of the '**myArduPrefs.txt**' file (if one is provided

there). Choices only appear if a ".qm" translation file exists in the translations folder (inside the UnoArduSim.exe home directory).

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.

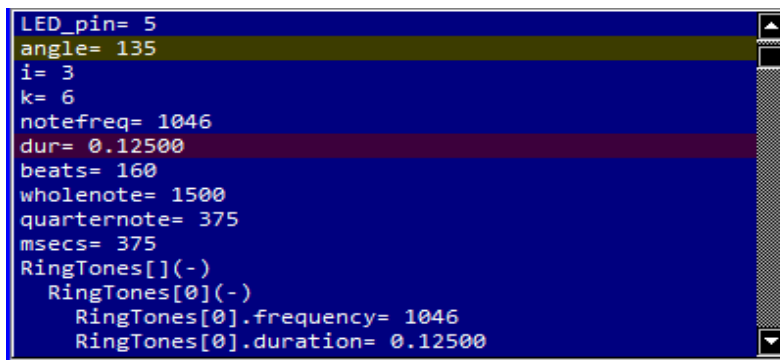
When **Auto Semicolons** is checked, pressing **Enter** to end a line automatically inserts the line-terminating semicolon.

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 .

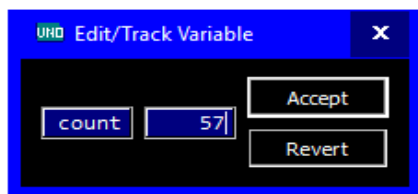


Arrays and **objects** are shown in either **un-expanded** or **expanded** format, with either a trailing plus `'(+)'` or minus `'(-)'` sign, respectively. The symbol for an array `x` shows as `'x[]'`. To expand it to show all elements of the array, just single-click on `'x[] (+)'` in the **Variables 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.



The **Edit/Track** window gives you **the ability to follow any variable value during execution**, or to **change its value in the middle of (halted) program execution** (so you can 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, **leave the dialog-box open** and then one of the **Run** or **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 value**, fill in the edit-box value, and **Accept**. Continue execution (using any of the **Step** or **Run** commands) to use that new value from that point forward (or you can **Revert** to the previous value).

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.

Lab Bench Pane

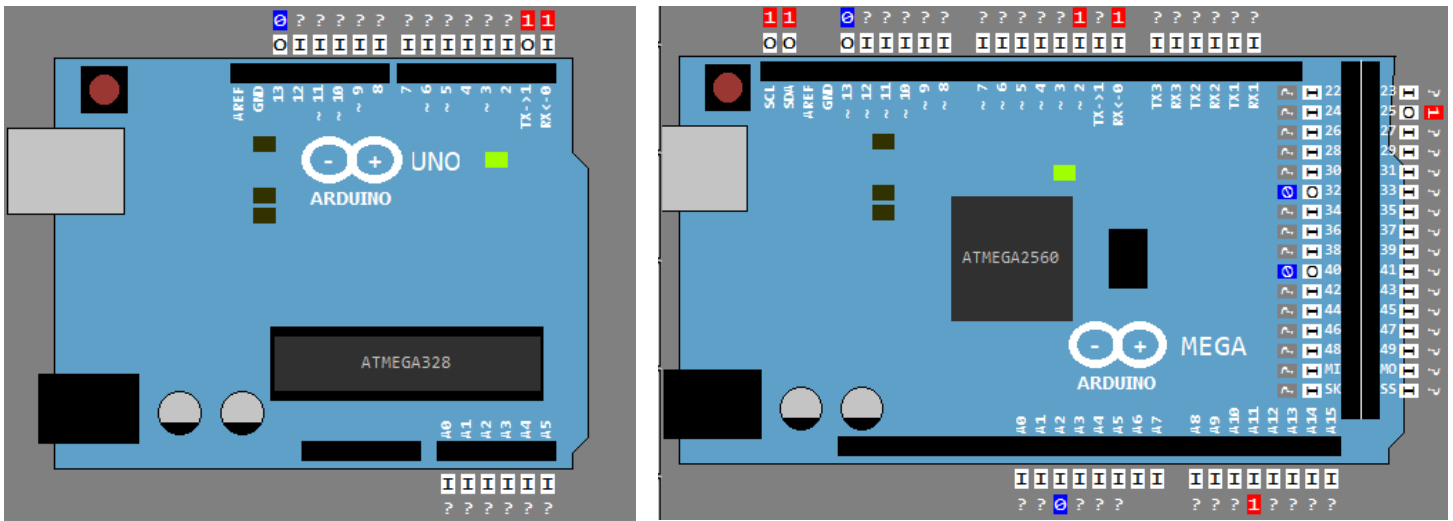
The Lab Bench Pane shows a 5-volt 'Uno' or 'Mega' board which surrounded by a set of 'I/O' devices that you can select/customize, and connect to your desired 'Uno' or 'Mega' pins.

The 'Uno' or 'Mega'

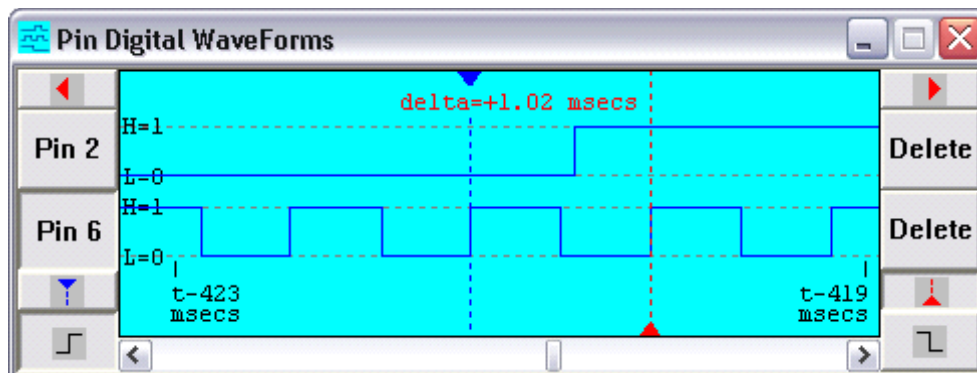
This is a depiction of the 'Uno' or 'Mega' board and its onboard LEDs. When you load a new program into UnoArduSim, if it successfully parses it undergoes a "simulated download" to the board that mimics the way an actual board behaves— you will see the serial RX and TX LED flashing (along with activity on pins 1 and 0 which are *hard-wired for serial communication with a host computer*). This is immediately followed by a pin 13 LED flash that signifies board reset and (and UnoArduSim automatic halt at) the beginning of your loaded program execution. You can avoid this display and associated loading lag by deselecting **Show Download** from **Configure | Preferences**.

The window allows you to visualize the digital logic levels on all 20 'Uno' pins or all 70 'Mega' 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', or '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 '^'.

Note that **Digital pins 0 and 1 are hard-wired through 1-kOhm resistors to the USB chip for serial communication with a host computer.**













Left-clicking on any 'Uno' or 'Mega' pin will open a **Pin Digital Waveforms** window that displays the past **one-second 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).



Click to page view left or right, or use keys Home, PgUp, PgDn, End

One of the displayed waveforms will be the **active pin** waveform, indicated by its "Pin" button being shown as

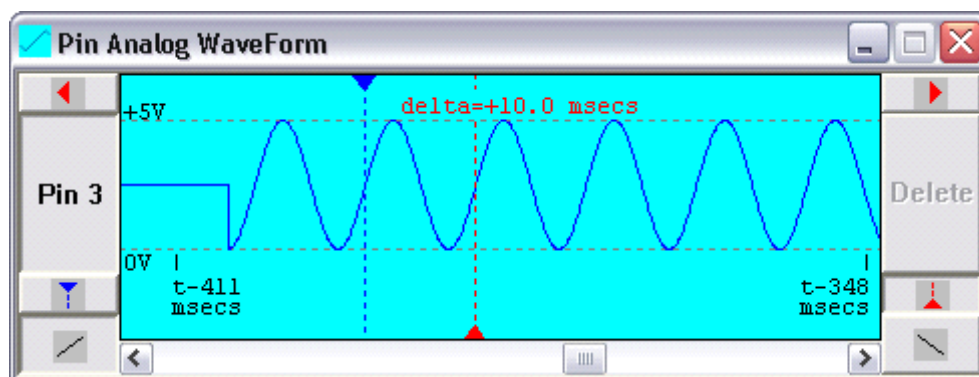
depressed (as in the above Pin Digital Waveforms screen capture). You can select a waveform by clicking its Pin number button, and then select the edge polarity of interest by clicking the appropriate rising/falling edge-polarity selection button, , or , or by using the shortcut keys **up-arrow** and **down-arrow**. You can then *jump* the active cursor (either the 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 cursor buttons, ,  or ,  (depending on which cursor was activated earlier with  or ), or simply use the keyboard keys **←** and **→** .

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







You can *jump* 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.

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.

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



Click to page view left or right, or use keys Home, PgUp, PgDn, End

You can *jump the* blue or red cursor lines to the next rising or falling "slope point" by using the forward or backward arrow buttons (,  or ,  , again depending on activated cursor, or use the **←** and **→** keys) in concert with the rising/falling slope selection buttons ,  (the "slope point" occurs where the analog voltage passes through the ATmega pin 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

Pressing **'Ctrl-S'** inside **either window allows you to save the waveform (X,Y) data** to a text file of your choice, where **X** is in microseconds from the left side, and **Y** is in volts.

'I/O' Devices

A number of different devices surround the 'Uno' or 'Mega' board on the perimeter of the **Lab Bench Pane**. "Small" 'I/O' devices (of which you are allowed up to 16 in total) reside along the left and right sides of the Pane. "Large" 'I/O' devices (you are allowed as many as will fit) have "active" elements and reside along the top and bottom of the **Lab Bench Pane**. The desired number of each type of available 'I/O' device can be set using the menu **Configure | 'I/O' Devices**.

Smaller 'I/O' Devices	Big 'I/O' Devices
Push Button: 2	Servo Motor: 1
Switched Resistor: 4	DC Motor: 1
Piezo Speaker: 2	Stepper Motor: 1
Coloured LED: 6	Pulsed Stepper Motor:
4-LED Row:	Digital Pulser: 1
7-Segment LED:	Function Generator: 1
Pin Jumper:	SFT Serial:
Analog Slider: 2	SR Slave:
Total (max 16): 16	1-Wire Slave:
	One-Shot Generator:
	SPI Slave: 1
	I2C Slave:
	Character LCD_SPI:
	Character LCD_I2C:
	Character LCD_D4:
	Port Expander SPI:
	Port Expander I2C:
	Mux LED SPI:
	Mux LED I2C:
	ProgIO UNO:
	Total (max 8): 7

Each 'I/O' device has one or more pin attachments shown as a **two-digit** pin number (00, 01, 02, ... 10,11,12, 13 and either A0-A5, or 14-19, after that) in a corresponding edit-box. For pin numbers 2 through 9 you can simply enter the single digit – the leading 0 will be automatically provided, but for pins 0 and 1 you must first enter the leading 0. Inputs are *normally* on the left side of an 'I/O' device, and outputs are *normally* on the right (*space permitting*). All 'I/O' devices will respond directly to pin levels and pin-level changes, so will respond to either library functions targeted to their attached pins, or to programmed `'digitalWrite()'` (for "bit-banged" operation).

You can connect multiple devices to the same ATmega pin as long as this does not create an **electrical conflict**. Such a conflict can be created either by an 'Uno' or 'Mega' pin as `'OUTPUT'` driving against a strong-conduction (low-impedance) connected device (for example, driving against a 'PUSH' output or a 'MOTOR' **Enc** output), or by two connected devices competing with each other (for example both a 'PULSER' and a 'PUSH' - button attached to the same pin). Any such conflict would be disastrous in a real hardware implementation and so are disallowed, and will be flagged to the user via a pop-up message-box).

The dialog-box can be used to allow the user to choose the types, and numbers, of desired 'I/O' devices. From this dialog-box you can also **Save 'I/O' devices** to a text file, and/or **Load 'I/O' devices** from a previously saved (or edited) text file (*including all pin connections, and clickable settings, and any typed-in edit-box values*).

Note that the values in the period, delay, and pulse-width edit-boxes in the relevant IO devices can be given the suffix 'S' (or 's') . That indicates that they should be scaled according to the position of a global **'I/O ____S'** slider control that appears in the Main window **Tool-Bar**. With that slider fully to the right, the scale factor is 1.0 (unity), and with the slider fully to the left the scale factor is 0.0 (subject to minimum values enforced by each particular 'I/O' device). You can scale more than one edit-box value **simultaneously** using this slider. This feature allows you to drag the slider while executing to easily emulate changing pulse widths, periods and delays for those attached 'I/O' devices.

The remainder of this section provides descriptions for each type of device.

Several of these devices **support scaling of their typed-in values** using the slider on the main window **Tool-Bar**. If the device value has the letter 'S' as its suffix, its value will be multiplied by a scale factor (between 0.0 and 1.0) that is determined by the slider-thumb position, subject to the device minimum value constraint (1.0 is fully to the right, 0.0 is fully to the left) --see **'I/O ____S'** under each of those devices detailed below.



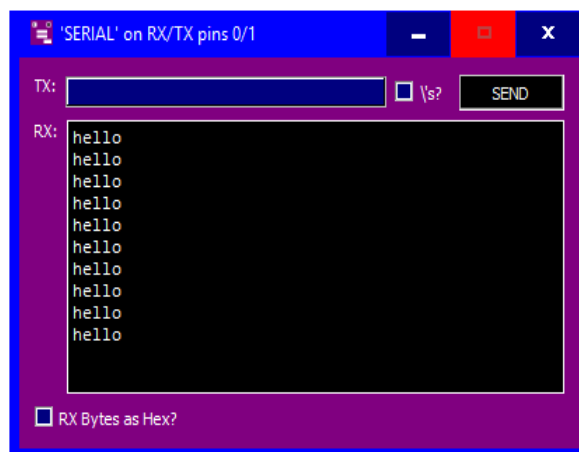
Serial Monitor ('SERIAL')



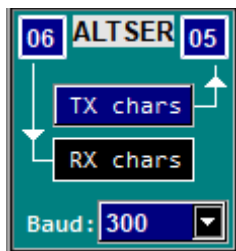
This 'I/O' device allows for ATmega hardware-mediated serial input and output (through the 'Uno' or 'Mega' USB chip) on pins 0 and 1. The baud-rate is set using the drop-down list at its bottom – the selected baud-rate **must match** the value your program passes to the `'Serial.begin()'` function for proper transmission/reception. *The serial communication is fixed at 8 data bits, 1 stop bit, and no parity bit.* You are allowed to **disconnect** (blank) **but not replace** TX pin 00, but not RX pin 01.

To send keyboard input to your program, type one or more characters in the upper (TX chars) edit window and then hit the **'Enter'** keyboard key. (characters become italicized to indicate transmissions have begun) – or if already in progress, added typed characters will be in italics. You can then use the `'Serial.available()'` and `'Serial.read()'` functions to read the characters in the order in which they were received into the pin 0 buffer (the leftmost typed character will be sent first). Formatted textual and numeric printouts, or un-formatted byte values, can be sent to the lower console output (RX chars) window by calling the Arduino `'print()'`, `'println()'`, or `'write()'` functions.

Additionally, **a larger window for setting/viewing TX and RX characters can be opened by double-clicking (or right-clicking) on this 'SERIAL' device.** This new window has a larger TX chars edit-box, and a separate 'Send' button which may be clicked to send the TX characters to the 'Uno' or 'Mega' (on pin 0). There is also a check-box option to re-interpret backslash-escaped character sequences such as `'\n'` or `'\t'` for non-raw display.



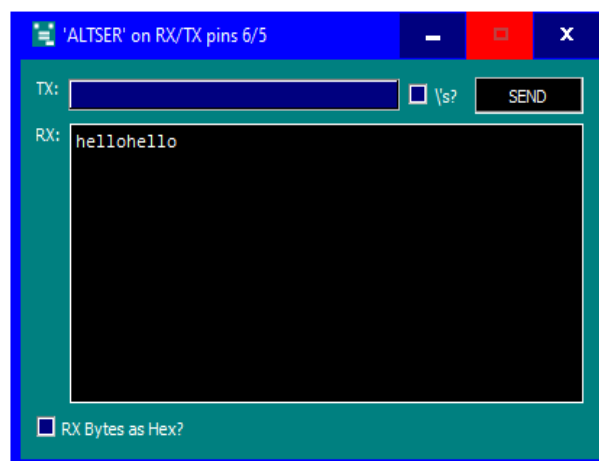
Alternate Serial ('ALTSER')



This 'I/O' device allows for library, or, alternatively, user "bit-banged", serial input and output on any pair of 'Uno' or 'Mega' pins you choose to fill in (**except for** pins 0 and 1 which are dedicated to hardware `'Serial'` communication). Your program must have an `'#include <SoftwareSerial.h>'` line near the top if you wish to use the functionality of that library. As with the 'SERIAL' device, the baud-rate for 'ALTSER' is set using the drop-down list at its bottom – the selected baud-rate must match the value your program passes to the `'begin()'` function for proper transmission/reception. *The serial communication is fixed at 8 data bits, 1 stop bit, and no parity bit.*

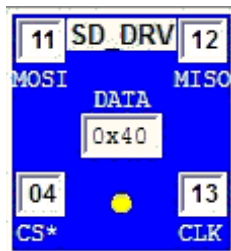
Also, as with 'SERIAL', **a larger window for TX and RX setting/viewing can be opened by double-clicking (or right-clicking) on the ALTSER device.**

Note that unlike the hardware implementation of `'Serial'`, in `'SoftwareSerial'` there is no provided TX buffer supported by internal ATmega interrupt operations (only an RX buffer), so that `'write()'` (or `'print'`) calls are blocking (that is, your program will not proceed until they are complete).



SD Disk Drive('SD_DRV')

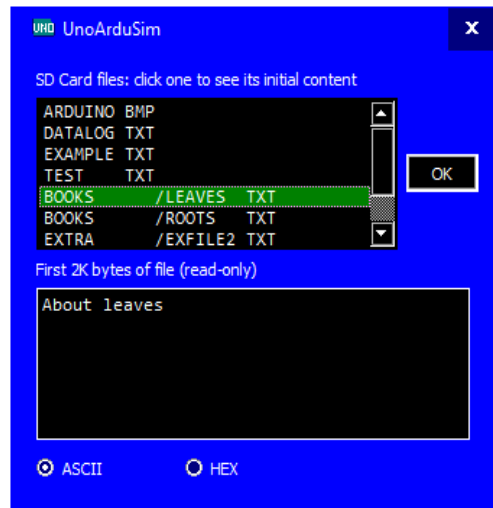
This 'I/O' device allows for library software-mediated (but **not** "bit-banged") file input and output operations on the 'Uno' or 'Mega' **SPI** pins (you can choose which **CS*** pin you will use). Your program can simply `'#include <SD.h>'` line near the top, and you can use `<SD.h>` functions OR directly call `'SdFile'` functions yourself.



A larger window displaying directories and files (and content) can be opened by double-clicking (or right-clicking) on the 'SD_DRV' device. All disk content is **loaded from** an **SD** sub-directory in the loaded program directory (if it exists) at `'SdVolume::init()'`, **and is mirrored to** that same **SD** sub-directory on file `'close()'`, `'remove()'`, and on

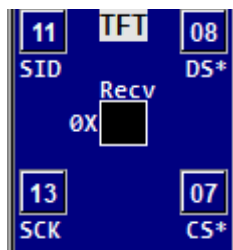
`'makeDir()'` and `'rmDir()'`.

A yellow LED flashes during SPI transfers, and 'DATA' shows the last 'SD_DRV' **response** byte. All SPI signals are accurate and can be viewed in a **Waveform window**.



TFT Display ('TFT')

This 'I/O' device emulates an Adafruit™ TFT display of size 1280by-160 pixels (in its native rotation=0, but when using the **'TFT.h'** library, `'TFT.begin()'` initialization sets for rotation=1 which gives a "landscape" view of 160-by-128 pixels). You can drive this device by calling the functions of the **'TFT.h'** library (which first requires `'#include <TFT.h>'`), or you can use the SPI system to send you own sequence of bytes to drive it.

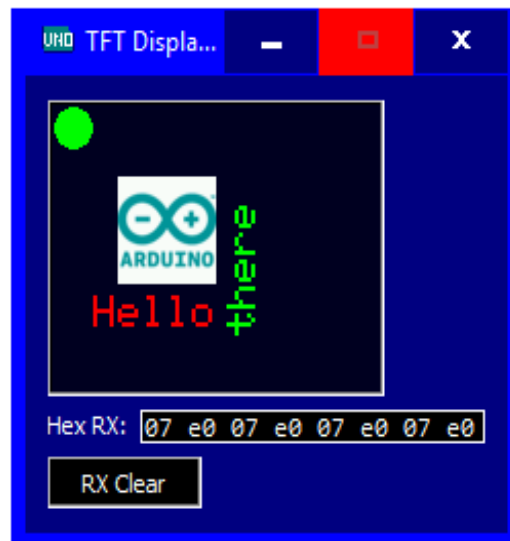


The TFT is always connected to 'SPI' pins 'MOSI' (for 'SID') and 'SCK' (for 'SCK') – those cannot be changed. The 'DS*' pin is for data/command select ('LOW' selects data mode), and the 'CS*' pin is the active-low chip-select

There is no Reset pin provided so you cannot do a hardware reset of this device by driving a pin low (as the `'TFT::begin()'` function attempts to do when you have passed a valid 'reset' pin number as the third parameter to the `'TFT(int cs, int ds, int rst)'` constructor). The device does however have a hidden connection to the system Reset line so it

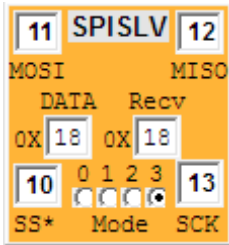
resets itself every time you click the main UnoArduSim Reset toolbar icon, or the 'Uno' or 'Mega' board reset button.

By **double-clicking** (or **right-clicking**) on this device, a larger window is opened to show that full 160-by-128 pixel LCD display, along with the most recently received 8 bytes (as shown below)



Configurable SPI Slave ('SPISLV')

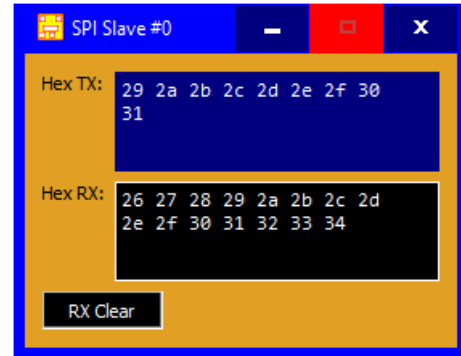
This 'I/O' device emulates a chosen-mode SPI slave with an active-low **SS*** ("slave-select") pin controlling the **MISO** output pin (when **SS*** is high, **MISO** is not driven). Your program must have an `'#include <SPI.h>'` line if you wish to use the functionality of the built-in SPI Arduino object and library. Alternatively, you may choose to create your own "bit-banged" **MOSI** and **SCK** signals to drive this device.



The device senses edge transitions on its **CLK** input according to the selected mode (`'MODE0'`, `'MODE1'`, `'MODE2'`, or `'MODE3'`), which must be chosen to match the programmed SPI mode of your program.

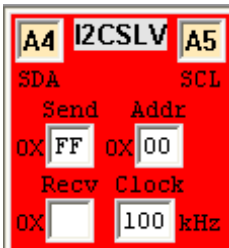
By double-clicking (or right-clicking) on the device you can open a larger companion window that instead allows you to fill in 32-byte-maximum buffer (so as to

emulate SPI devices which automatically return their data), and to see the last 32 received bytes (all as hex pairs). **Note that** the next TX buffer byte is automatically sent to 'DATA' only **after** a full `'SPI.transfer()'` has completed!



Two-Wire I2C Slave ('I2CSLV')

This 'I/O' device only emulates a *slave-mode* device. The device may be assigned an I2C bus address using a two-hex-digit entry in its 'Addr' edit-box (it will only respond to I2C bus transactions involving its assigned address). The device sends and receives data on its open-drain (pull-down-only) **SDA** pin, and responds to the bus clock signal on its open-drain (pull-down-only) **SCL** pin. Although the 'Uno' or 'Mega' will be the bus master responsible to generating the **SCL** signal, this slave device will also pull **SCL** low during its low phase in order to extend (if it needs to) the bus low time to one appropriate to its internal speed (which can be set in its 'Clock' edit-box).

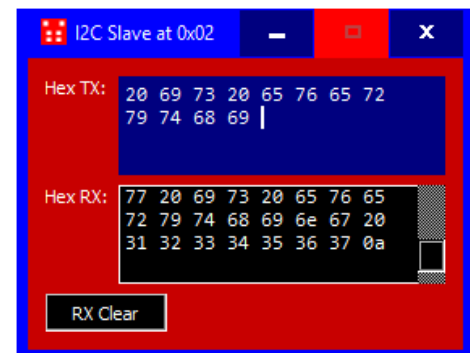


Your program must have an `'#include <Wire.h>'` line if you wish to use the functionality of the `'TwoWire'` library to interact with this device. Alternatively, you may choose to create your own bit-banged data and clock signals to drive this slave device.

A single byte for transmission back to the 'Uno' or 'Mega' master can be set into the 'Send' edit-box, and a single (most-recently-received) byte can be viewed in its (read-only) 'Recv' edit-box. **Note that** the 'Send' edit-box value always reflects the **next** byte for transmission from this device internal data buffer.

By double-clicking (or right-clicking) on the device you can open a larger companion window that instead allows you to fill in a 32-byte-maximum FIFO buffer (so as to emulate TWI devices with such

functionality), and to view (up to a maximum of 32) bytes of the most recently received data (as a two-hex-digit display of 8 bytes per line). The number of lines in these two edit-boxes corresponds to the chosen TWI buffer size (which can be selected using **Configure | Preferences**). This has been added as an option since the Arduino `'Wire.h'` library uses **five** such RAM buffers in its implementation code, which is RAM memory expensive. By editing the Arduino installation `'Wire.h'` file to change defined constant `'BUFFER_LENGTH'` (and also editing the companion `'utility/twi.h'` file to change TWI buffer length) both to be instead either 16 or 8, a user *could* significantly reduce the RAM memory overhead of the 'Uno' or 'Mega' in a targeted **hardware implementation** – UnoArduSim therefore mirrors this real-world possibility through **Configure | Preferences**.



Text LCD I2C ('LCDI2C')

This 'I/O' device emulates a 1,2, or 4-line character-LCD, in one of three modes:

- a) backpack type 0 (Adafruit style port expander with hardware having I2C bus address 0x20-0x27)
- b) backpack type 1 (DFRobot style port expander with hardware having I2C bus address 0x20-0x27)
- c) no backpack (Native mode integrated I2C interface having I2C bus address 0x3C-0x3F)

Supporting library code for each device mode has been provided inside the 'include_3rdParty' folder of your UnoArduSIm installation directory: 'Adafruit_LiquidCrystal.h', 'DFRobot_LiquidCrystal.h', and 'Native_LiquidCrystal.h', respectively.



The device may be assigned any I2C bus address using a two-hex-digit entry in its 'Addr' edit-box (it will only respond to I2C bus transactions involving its assigned address). The device receives bus address and data (and responds with ACK=0 or NAK=1) on its open-drain (pull-down-only) **SDA** pin. You can only write LCD commands and DDRAM data – you **cannot** read back data from the written DDRAM locations..



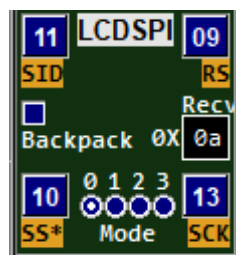
Double-click or **right-click** to open the LCD screen monitor window from which you can also set the screen size and character set.

Text LCD SPI ('LDCSPI')

This 'I/O' device emulates a 1,2, or 4-line character-LCD, in one of two modes:

- a) backpack (Adafruit style SPI port expander)
- b) no backpack (Native mode integrated SPI interface – as shown below)

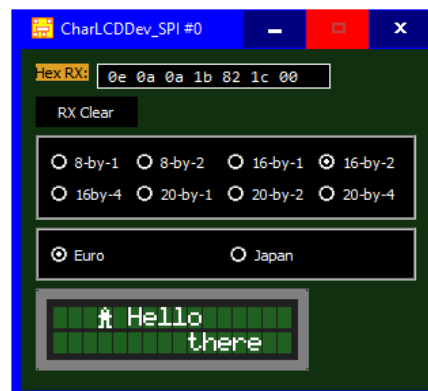
Supporting library code for each device mode has been provided inside the 'include_3rdParty' folder of your UnoArduSIm installation directory: 'Adafruit_LiquidCrystal.h', and 'Native_LiquidCrystal.h', respectively.



Pin 'SID' is serial data in, 'SS*' is the active-low device-select, 'SCK' is the clock pin, and 'RS' is the data/command pin. You can only write LCD commands and DDRAM data (all SPI transactions are writes)– you **cannot** read back data from the written DDRAM locations.

size and character set.

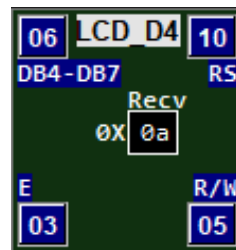
Double-click or **right-click** to open the LCD screen monitor window from which you can also set the screen



Text LCD D4 ('LCD_D4')

This 'I/O' device emulates a 1,2, or 4-line character-LCD having a 4-bit-parallel bus interface. Data bytes are written/read in **two halves** on its 4 data pins 'DB4-DB7' (where the edit box contains the *lowest numbered of its 4 consecutive pin numbers*),-- data is clocked on falling edges on the 'E' (enable) pin, with data direction controlled by the 'R/W' pin, and LCD data/command mode by the 'RS' pin.

Supporting library code has been provided inside the 'include_3rdParty' folder of your UnoArduSIm installation directory: 'Adafruit_LiquidCrystal.h', and 'Native_LiquidCrystal.h' both work.



Double-click or **right-click** to open the LCD screen monitor window from which you can also set the screen size and character set.



Multiplexer LED I2C ('MUXI2C')

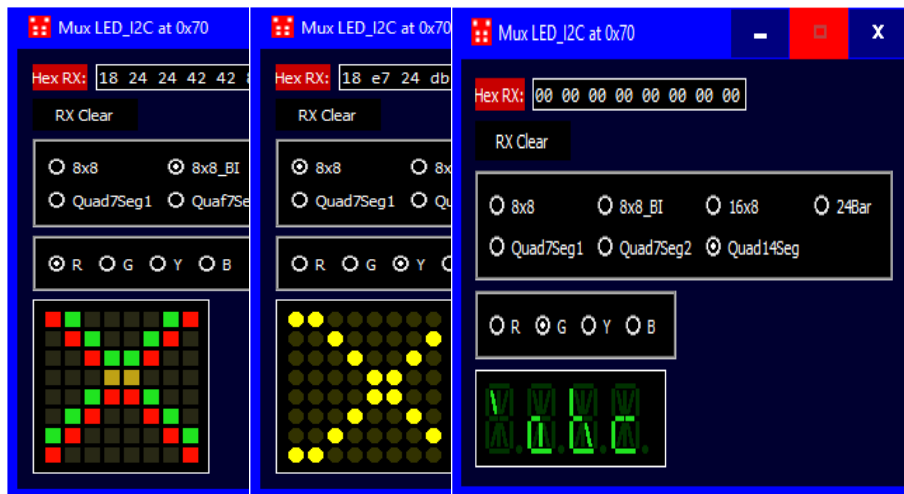
This 'I/O' device emulates an I2C-interfaced HT16K33 controller (having I2C bus address 0x70-0x77) to which one of several different types of multiplexed LED displays can be attached



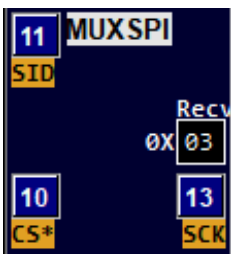
- a) 8x8, or 16x8, LED array
- b) 8x8 bi-color LED array
- c) 24-bi-color-LED bar
- d) two styles of 4-digit 7-segment displays
- e) one 4-digit 14-segment alphanumeric display

All are supported by the '**Adafruit_LEDBackpack.h**' code provided inside the 'include_3rdParty' folder:

Double-click (or right-click) to open a larger window to choose and view one of the several colored LED displays.



Multiplexer LED SPI ('MUXSPI')

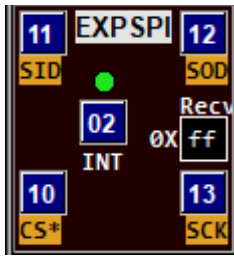


A *multiplexed-LED controller* based on the MAX6219, with supporting '**MAX7219.h**' code provided inside the 'include_3rdParty' folder to drive up to eight 7-segment digits.

Double-click (or right-click) to open a larger window to view the colored 8-digit 7-segment display.



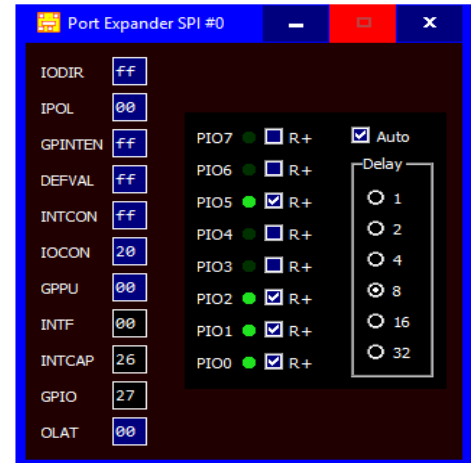
Expansion Port SPI ('EXPSPi')



An 8-bit port expander based on the MCP23008, with supporting 'MCP23008.h' code provided inside the 'include_3rdParty' folder. You can write to MCP23008 registers, and read back the GPIO pin levels. Interrupts can be enabled on each GPIO pin change – a triggered interrupt will drive the 'INT' pin.

Double-click (or right-click) to open a larger window to see the 8 GPIO port lines, and the attached pull-up resistors. You can change pull-ups manually by clicking, or attach a counter that will periodically change them in a up-count

manner. The rate at which the count increments is determined by the scale-down delay factor chosen by the user (a 1x factor corresponds to one increment approximately every 30 milliseconds; higher delay factors give a slower up-count rate)

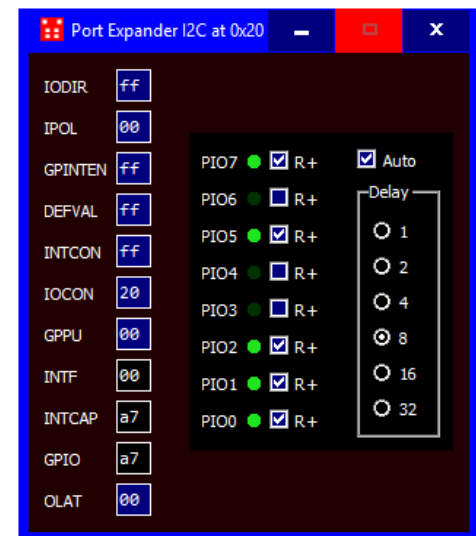


Expansion Port I2C ('EXPI2C')



An 8-bit port expander based on the MCP23008, with supporting 'MCP23008.h' code provided inside the 'include_3rdParty' folder. Capabilities match the 'EXPSPi' device.

Double-click (or right-click) to open a larger window as from the 'EXPSPi' device.



'1-Wire' Slave ('OWISLV')

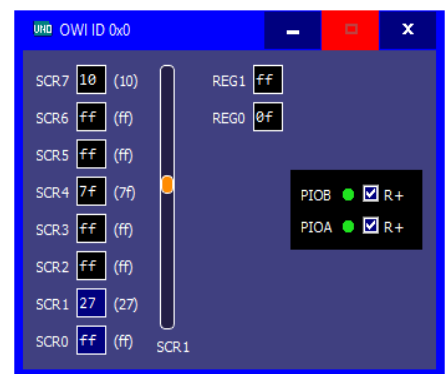
This 'I/O' device emulates one of a small set of '1-Wire' bus devices connected to pin OWIO. You can create a '1-Wire' bus (with one or more of these slave '1-Wire' devices) on the 'Uno' or 'Mega' pin of your choice. This device can be used by calling the '`OneWire.h`' library functions after placing an '`#include <OneWire.h>`' line at the top of your program. Alternatively you can also use bit-banged signals on OWIO to this device (although that's very tricky to do properly without causing an electrical conflict – such a conflict is still possible even when using the '`OneWire.h`' functions, but such conflicts are reported in UnoArduSim).



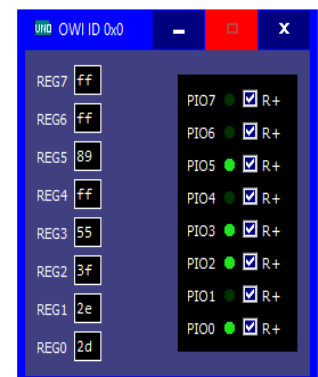
Each real-world OWISLV device must have a unique 8-byte (64-bit!) internal serial number – in UnoArduSim this is simplified by the user providing a short 1-byte hexadecimal '**ID**' value (which is assigned sequentially by default at device load/addition), plus the '**Fam**' Family code for that device. UnoArduSim recognizes a small set of Family codes as of V2.3 (0x28, 0x29, 0x3A, 0x42) covering temperature-sensor, and parallel IO (PIO) devices (an unrecognized Family code sets the device to become a generic 8-byte scratchpad device with generic sensor).

If the device Family has no PIO registers, registers **D0** and **D1** represent the first two scratchpad bytes, otherwise they represent the PIO “status” (actual pin levels) register and PIO pin latch data register, respectively.

By **double-clicking** (or **right-clicking**) on the device, a larger **OWIMonitor** window is opened. From that larger window you can inspect all the device registers, change scratchpad locations SCR0 and SCR1 using edits and a slider (again, SCR0 and SCR1 only correspond to **D0** and **D1** if no PIO is present), or set external pin PIO pull-ups. When SCR0 and SCR1 are edited, UnoArduSim remembers these edited values as the user “preference” representing an initial (starting from Reset) value representing the signed value output from the device's sensor – the Slider is reset to 100% (a scale factor of 1.0) at the time of the edit. When the Slider is subsequently moved, the '**signed**' value in SCR1 is scaled down according to the slider position (scale factor from 1.0 down to 0.0) – this feature allows you to easily test the response of your program to smoothly changing sensor values. .

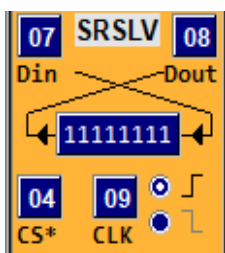


For a device with PIO pins, when you set the pin-level check-boxes, UnoArduSim remembers these checked values as the current pull-ups applied externally to the pins. These external pull-up values then are used along with the pin latch data (register **D1**) to determine the final actual pin levels, and light or extinguish the green LED attached to the PIO pin (the pin only goes '**HIGH**' if an external pull-up is applied, **and** the corresponding **D1** latch bit is a '**1**').



Shift Register Slave ('SRSLV')

This 'I/O' device emulates a simple shift-register device with an active-low **SS*** ("slave-select") pin controlling the '**Dout**' output pin (when **SS*** is high, '**Dout**' is not driven). Your program could use the functionality of the built-in SPI Arduino object and library. Alternatively, you may choose to create your own “bit-banged” '**Din**' and **CLK** signals to drive this device.



The device senses edge transitions on its **CLK** input which trigger shifting of its register – the polarity of sensed **CLK** edge may be chosen using a radio-button control. On every **CLK** edge (of the sensed polarity), the register captures its **Din** level into the least-significant bit (LSB) position of the shift register, as the remaining bits are simultaneously shifted left one position toward the MSB position. Whenever **SS*** is low, the current value in the MSB position of the shift register is driven onto '**Dout**'.

Programmable 'I/O' Device ('PROGIO')



This 'I/O' device is actually a bare 'Uno' board that you can program (with a separate program) in order to emulate an 'I/O' device whose behaviour you can completely define. You can choose up to four pins (IO1, IO2, IO3, and IO4) that this slave 'Uno' will share in common with the master (main 'Uno' or 'Mega') that appears in the middle of your **Lab Bench Pane**. As with other devices, any electrical conflict between this 'Uno' slave and the master board will be detected and flagged. Note that all connections are **directly wired**, **except for pin 13** (where a series R-1K resistor is assumed between the two pins in order to prevent an electrical conflict at Reset). As of V2.8, the connection between master and slave pins are **mapped**: if the master is also an 'Uno',

the mapping defaults to an identity (except that pin 1 is mapped to pin 0, and vice-versa to allow 'Serial' communications); if the master is a 'Mega', pins 1 and 0 are again flipped, and **all SPI and TWI pins are mapped** for direct connection between the corresponding master and slave subsystems. This default mapping can be overridden by specifying in your **IODevs.txt** file an explicit list of 4 master pin numbers that follows the PROGIO program file name – if any of these values are -1, that is the same as the default mapping. For this device the pin numbers typed in **are those of the slave 'Uno'**. The picture at the left shows the 4 slave pins specified to its SPI system pins (SS*, MISO, MOSI, SCK) – irrespective of whether the master were an 'Uno' or a 'Mega', this would allow you to program this slave as a generic SPI slave (or master) whose behaviour you can define programmatically.

By **double-clicking** (or **right-clicking**) on this device, a larger window is opened to show that this 'Uno' slave has its own **Code Pane** and associated **Variables Pane**, just like the master has. It also has its own **Tool-Bar**, which you can use to **load** and **control execution** of a slave program – the icon actions have the same keyboard shortcuts as those in the Main window. (**Load** is **Ctrl-L**, **Save** is **Ctrl-S** etc.). Once loaded, you can execute from **either** the Main UnoArduSim window, or from inside this Slave Monitor window – in either case the Main program and Slave 'Uno' program remain locked in synchronization to the passage of real time as their executions progress forward. **To choose the Code Pane that will have the load, search, and execution focus, click on its parent window title bar – the non-focus Code Pane then has its tool bar actions grayed out.**

Some possible ideas for slave devices that could be programmed into this 'PROGIO' device are listed below. For serial, I2C, or SPI device emulation, you can use appropriate program coding with arrays for Send and Receive buffers, in order to emulate complex device behaviour, **such as GPS devices, serial EEPROM devices, etc.**

a) An SPI master or slave device. UnoArduSimV2.4 has extended the '**SPI.h**' library to allow slave mode SPI operation through an optional '**mode**' parameter in '**SPI.begin(int mode = SPI_MASTR)**'. Explicitly pass '**SPI_SLV**' to select slave mode (instead of relying on the default master mode). You can also now define a user interrupt function (let us call it '**onSPI**') in either master or slave program to transfer bytes by calling another added extension '**SPI.attachInterrupt(user_onSPI)**'. **Now** calling '**rxbyte=SPI.transfer(tx_byte)**' from inside your '**user_onSPI**' function will clear the interrupt flag, and will **return immediately** with the just-received byte in your variable '**rxbyte**'. Alternatively, you can avoid attaching an SPI interrupt, and instead simply call '**rxbyte=SPI.transfer(tx_byte)**' from inside your main program -- that call will **block execution** until an SPI byte has been transferred, and will then **return** with the newly received byte inside '**rxbyte**'.

b) A generic **serial 'I/O'** device. You can communicate with the Master board using either '**Serial**', or a '**SoftwareSerial**' defined inside your slave program – for '**SoftwareSerial**' you must define '**txpin**' and '**rxpin**' oppositely to those of the Master so that the slave receives on the pin on which the master transmits (and vice-versa), but for '**Serial**' only, the 1 and 0 pins are already flipped for you.

c) A generic 'I2C' master or slave device. Slave operation has now been added to complete UnoArduSim's implementation of the '**Wire.h**' library (functions '**begin(address)**', '**onReceive()**' and '**onRequest**' have now been implemented in order to support slave mode operations).

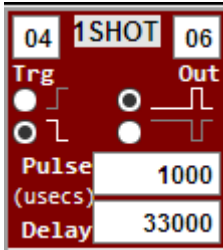
d) A generic digital **Pulser**. Using '**delayMicroseconds()**' and '**digitalWrite()**' calls inside '**loop()**' in your 'PROGIO' program, you can define the '**HIGH**' and '**LOW**' intervals of a pulse train. By adding a separate '**delay()**' call inside your '**setup()**' function, you can delay the start of this pulse train. You can even vary the pulse widths as time progresses by using a counter variable. You could also use a separate '**IOx**' pin as a trigger to start the timing of an emulated '1Shot' (or double-shot, triple-shot etc.) device, and you can control the produced pulse width to change in whatever manner you desire as time progresses.

e) A random signaller. This is a variation on a digital **Pulser** that also uses calls to '**random()**' and

'`delayMicroseconds()`' to generate random times at which to '`digitalWrite()`' a signal on any chosen pin shared with the master. Using all four '**IOx**' pins would allow four simultaneous (and unique) signals.

One-Shot ('1SHOT')

This 'I/O' device emulates a digital one-shot that can generate a pulse of chosen polarity and pulse-width on its 'Out' pin, occurring after a specified delay from a triggering edge received on its **Trg** (trigger) input pin. Once the specified triggering edge is received, timing begins and then a new trigger pulse will not be recognized until the 'Out' pulse has been produced (and has completely finished).



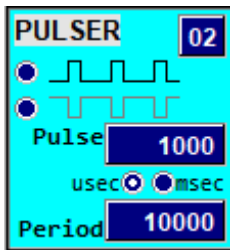
One possible use of this device is to simulate ultrasound ranging sensors that generate a range pulse in response to a triggering pulse. It can also be used wherever you wish to generate a pin input signal synchronized (after your chosen delay) to a pin output signal created by your program.

'Pulse' and 'Delay' values can be scaled from the Main window **Tool-Bar** 'I/O ____S' scale-factor slider control by adding the suffix 'S' (or 's') to either one (or both).

Another use for this device is in testing a program that uses interrupts, and you would like to see what happens if a **specific program instruction** gets interrupted. Temporarily disconnect the 'I/O' Device you have connected to pin 2 (or pin 3) and replace it by a '1SHOT' device whose 'Out' pin is connected to to pin 2 (or pin3, respectively), You can then trigger its 'Trg' input (assuming rising-edge sensitivity is set there) by inserting the instruction pair { '`digitalWrite(LOW)`' , '`digitalWrite(HIGH)`' } **just prior** to the instruction inside which you wish the interrupt to occur. Set the 1SHOT's 'Delay' to time the pulse produced on 'Out' to happen inside the program instruction that follows this triggering instruction pair. Note that some instructions mask interrupts (such as '`SoftwareSerial.write(byte)`' , and so cannot be interrupted.

Digital Pulser ('PULSER')

This 'I/O' device emulates a simple digital pulse waveform generator which produces a periodic signal that can be applied to any chosen 'Uno' or 'Mega' pin.



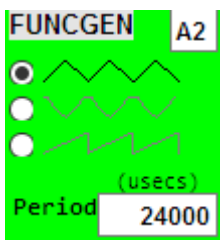
The period and pulse widths (in microseconds) can be set using edit-boxes—the minimum allowed period is 50 microseconds, and the minimum pulse width is 10 microseconds. You can choose between timing values in microseconds ('usec') and milliseconds ('msec'), and this choice will be saved along with the other values when you 'Save' from **Configure | I/O Devices**.

The polarity can also be chosen: either positive leading-edge pulses (0 to 5V) or negative leading-edge pulses (5V to 0V).

'Pulse' and 'Period' values can be scaled from the main **Tool-Bar** 'I/O ____S' scale-factor slider control by adding as a suffix 'S' (or 's') to either one (or both). This allows you to then change the 'Pulse' or 'Period' value **dynamically** during execution.

Analog Function Generator ('FUNCGEN')

This 'I/O' device emulates a simple analog waveform generator which produces a periodic signal that can be applied to any chosen 'Uno' or 'Mega' pin.



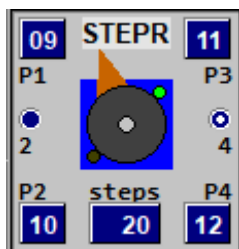
The period (in microseconds) can be set using the edit-box—the minimum allowed period is 100 microseconds. The waveform it creates can be chosen to be sinusoidal, triangular, or sawtooth (to create a square wave, use a 'PULSER' instead). At smaller periods, fewer samples per cycle are used to model the produced waveform (only 4 samples per cycle at period=100 microseconds).

The 'Period' value can be scaled from the Main window **Tool-Bar** 'I/O ____S' scale-factor slider control by adding as a suffix the letter 'S' (or 's'). This allows you to then change the

'Period' value **dynamically** during execution.

Stepper Motor ('STEPR')

This 'I/O' device emulates a 6V bipolar or unipolar Stepper Motor with an integrated driver controller driven by **either two** (on **P1,P2**) or **four** (on **P1,P2,P3,P4**) control signals. The number of steps per revolution can also be set. You can use the '**Stepper.h**' functions '**setSpeed()**' and '**step()**' to drive the 'STEPR'. Alternatively, 'STEPR' will *also* respond to your own '**digitalWrite()**' "bit-banged" drive signals.



The motor is accurately modeled both mechanically and electrically. Motor-driver voltage drops and varying reluctance and inductance are modeled along with a realistic moment of inertia with respect to holding torque. The motor rotor winding has a modeled resistance of $R=6$ ohms, and an inductance of $L=6$ milli-Henries which creates an electrical time constant of 1.0 millisecond. Because of the realistic modeling you will notice that very narrow control pin pulses *do not get* the motor to step – both due to the finite current rise time, and the effect of rotor inertia. This agrees with what is observed when driving a real stepper motor from an 'Uno' or 'Mega' with, of course, an appropriate (**and required**) motor driver chip in between the

motor wires and the 'Uno' or 'Mega' !

An unfortunate bug in the Arduino '**Stepper.h**' library code means that at reset the Stepper motor will not be in Step position 1 (of four steps). To overcome this, the user should use '**digitalWrite()**' in his/her '**setup()**' routine to initialize the control pin levels to the '**step(1)**' levels appropriate to 2-pin (0,1) or 4-pin (1,0,1,0) control, and allow the motor 40-100 milliseconds to move to the 12-noon reference initial desired motor position.

Note that **gear reduction is not directly supported** due to lack of space, but you can emulate it in your program by implementing a modulo-N counter variable and only calling '**step()**' when that counter hits 0 (for gear reduction by factor N).

AS of V2.6, this device now includes a 'sync' LED (green for synchronized, or red when off by one or more steps). In addition, in addition to the number of steps per revolution, two extra (hidden) values may optionally be specified in the IODevs.txt file to specify the mechanical load– for example, values 20, 50, 30 specifies 20 steps per revolution, a load moment of inertia 50 times that of the motor rotor itself, and a load torque of 30 percent of full motor hold torque.

Pulsed Stepper Motor ('PSTEPR')

This 'I/O' device emulates a 6V **micro-stepping** bipolar Stepper Motor with an integrated driver controller driven by a **pulsed 'Step'** pin, an active-low '**EN***' (enable) pin, and a '**DIR**' (direction) pin. The number of full steps per revolution can also be set directly, along with the number of micro-steps per full step (1,2,4,8, or 16). In addition to these settings, two extra (hidden) values may optionally be specified in the IODevs.txt file to specify the mechanical load– for example, values 20, 4, 50, 30 specifies 20 steps per revolution, 4 micro-steps per full step, a load moment of inertia 50 times that of the motor rotor itself, and a load torque of 30 percent of full motor hold torque.

You must write code to drive the control pins appropriately.

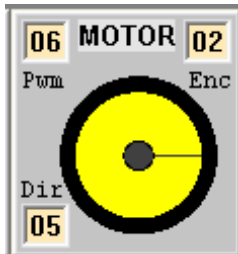


The motor is accurately modeled both mechanically and electrically. Motor-driver voltage drops and varying reluctance and inductance are modeled along with a realistic moment of inertia with respect to holding torque. The motor rotor winding has a modeled resistance of $R=6$ ohms, and an inductance of $L=6$ milli-Henries which creates an electrical time constant of 1.0 millisecond.

This device includes a yellow 'STEP' activity LED, and a 'sync' LED (GREEN for synchronized, or RED when off by one or more steps).

DC Motor ('MOTOR')

This 'I/O' device emulates a 6-volt supply 100:1 geared DC motor with an integrated driver controller driven by a pulse-width-modulation signal (on its **Pwm** input), and a direction control signal (on its **Dir** input). The motor also has a wheel encoder output which drives its **Enc** output pin. You can use '**analogWrite()**' to drive the **Pwm** pin with a 490 Hz (on 'Uno' pins 3,9,10,11) or 980 Hz (on 'Uno' pins 5,6) PWM waveform of duty cycle between 0.0 and 1.0 ('**analogWrite()**' values 0 to 255). Alternatively, 'MOTOR' will *also respond* to your own '**digitalWrite()**' "bit-banged" drive signals.



The motor is accurately modeled both mechanically and electrically. Accounting for (low) MOS-motor-driver transistor voltage drops and realistic no-load gear torque gives a full speed of almost 3 revs per second, and stall torque of just under 10 kg-cm (occurring at a steady PWM duty cycle of 1.0), with a mechanical time constant of approximately 40 milliseconds (which is increased by the inertia of any specified load). Three (optional) parameter values can be specified in a user 'IODevs.txt' file – 'F' or 'B' (for freewheel-coast or Brake mode when **Pwm** is LOW), then a constant load torque as a percentage of stall torque, then load moment of inertia as a integer multiple of intrinsic motor rotor inertia (these values default to 'F', 0, and 1 if none were specified). In addition there is a built-in constant opposing gearbox torque of 10% of stall torque (which adds to the load torque).

The motor rotor winding has a modeled resistance of $R=2$ ohms, and an inductance of $L=300$ micro-Henries which creates an electrical time constant of 150 microseconds. Because of the realistic modeling you will notice that very narrow PWM pulses *do not get* the motor to turn – both due to the finite current rise time, and the significant off-time after each narrow pulse. These combine to cause insufficient rotor momentum to overcome the gearbox spring-like lash-back under static friction. The consequence is when using '**analogWrite()**', a duty cycle below about 0.125 will not cause the motor to budge – this agrees with what is observed when driving a real gear-motor from an 'Uno' or 'Mega' with, of course, an appropriate (**and required**) motor driver module in between the motor and the 'Uno' or 'Mega' !.

The emulated motor encoder is a shaft-mounted optical-interruption sensor that produces a 50% duty cycle waveform having 8 complete high-low periods per wheel revolution (so your program can sense wheel rotational changes to a resolution of 22.5 degrees).

Servo Motor ('SERVO')

This 'I/O' device emulates a position-controlled PWM-driven 6-volt supply DC servo motor. Mechanical and electrical modeling parameters for servo operation will closely match those of a standard HS-422 servo. The servo has a maximum rotational speed of approximately 60 degrees in 180 milliseconds. If the lower-left checkbox is checked, the servo becomes a **continuous-rotation** servo with the same maximum speed, but now the PWM pulse-width sets the **speed** rather than the angle



Your program must have an '**#include <Servo.h>**' line before declaring your '**Servo**' instance(s) *if you choose to use the 'Servo.h' library functionality*, e.g. '**Servo.write()**', '**Servo.writeMicroseconds()**'. Alternatively, 'SERVO' also responds to '**digitalWrite()**' "bit-banged" signals. Due to the internal implementation of UnoArduSim, you are limited to 6 'SERVO' devices.

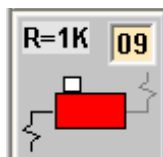
Piezo Speaker ('PIEZO')



This device allows you to "listen" to signals on any chosen 'Uno' or 'Mega' pin, and can be a useful adjunct to LEDs for debugging your program operation. You can also have a bit of fun playing ringtones by appropriately `'tone()'` and `'delay()'` calls (although there is no filtering of the rectangular waveform, so you will not hear "pure" notes) .

You can also listen to a connected 'PULSER' or 'FUNCGEN' device by hooking a 'PIEZO' to the pin that device drives.

Slide Resistor ('R=1K')



This device allows the user to connect to an 'Uno' or 'Mega' pin either a 1 k-Ohm pull-up resistor to +5V, or a 1 k-Ohm pull-down resistor to ground. This lets you simulate electrical loads added to a real hardware device. By left-clicking on the slide switch **body** you can toggle your desired pull-up or pull-down selection. Using one, or several, of these devices would allow you to set a single (or multi)-bit "code" for your program to read and respond to.

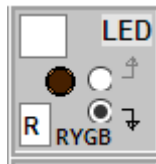
Push Button ('PUSH')



This 'I/O' device emulates a normally-open **momentary OR latching** single-pole, single-throw (SPST) push-button with a 10 k-ohm pull-up (or pull-down) resistor. If a rising-edge transition selection is chosen for the device, the push-button contacts will be wired between the device pin and +5V, with a 10 k-Ohm pull-down to ground. If a falling-edge transition is chosen for the device, the push-button contacts will be wired between the device pin and ground, with a 10 k-Ohm pull-up to +5V.

By left-clicking on the button, or pressing any key, you close the push-button contact. In **momentary** mode, it stays closed for as long as you hold down the mouse button or key, and in **latch** mode (enabled by clicking on the 'latch' button) it stays closed (and a different colour) until you press the button again. Contact bouncing (for 1 millisecond) will be produced each time you use the **space-bar** to press the push-button.

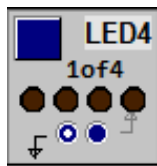
Coloured LED ('LED')



You can connect an LED between the chosen 'Uno' or 'Mega' pin (through a built-in hidden series 1 k-Ohm current-limiting resistor) to either ground or to +5V – this gives you the choice of having the LED light up when the connected 'Uno' or 'Mega' pin is **'HIGH'**, or instead, when it is **'LOW'** .

The LED colour can be chosen to be either red ('R'), yellow ('Y'), green ('G') or blue ('B') using its edit-box.

4-LED Row ('LED4')



You can connect this row of 4 coloured LEDs between the chosen set of 'Uno' or 'Mega' pins (each has a built-in hidden series 1 k-Ohm current-limiting resistor) to either ground or to +5V – this gives you the choice of having the LEDs light up when the connected 'Uno' or 'Mega' pin is **'HIGH'**, or instead, when it is **'LOW'** .

The **'1of4'** pin edit box accepts a single pin number which will be taken to mean **the first of four consecutive** 'Uno' or 'Mega' pins that will connect to the 4 LEDs.

The LED colour ('R', 'Y', 'G', or 'B') is a **hidden option** that can be **only be chosen by editing the IODevices.txt file** (which you can create using **Save** from the **Configure | I/O Devices** dialog-box).

7-Segment LED Digit ('7SEG')



You can connect this 7-Segment Digit LED display to a chosen set of **four consecutive 'Uno' or 'Mega' pins that give the hexadecimal code** for the desired displayed digit, ('0' through 'F'), and turn this digit on or off using the CS* pin (active-LOW for ON).

This device includes a built-in decoder which uses the **active-HIGH** levels on the four consecutive '1of4' pins to determine the requested hexadecimal digit to be displayed. The level on the lowest pin number (the one displayed in the '1of4' edit box) represents the least-significant bit of the 4-bit hexadecimal code.

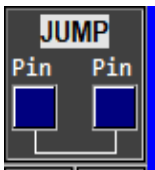
The colour of the LED segments ('R', 'Y', 'G', or 'B') is a **hidden option** that can be **only be chosen by editing the IODevices.txt file** you can create using **Save** from the **Configure | I/O Devices** dialog-box.

Analog Slider

A slider-controlled 0-5V potentiometer can be connected to any chosen 'Uno' or 'Mega' pin to produce a static (or slowly changing) analog voltage level which would be read by 'analogRead()' as a value from 0 to 1023. Use the mouse to drag, or click to jump, the analog slider.



Pin Jumper ('JUMP')



You can connect two 'Uno' or 'Mega' pins together using this device (if any electrical conflict is detected when you fill in the second pin number, the chosen connection is disallowed, and that pin is disconnected).

This jumper device has a limited usefulness, and is most useful when combined with interrupts, for program testing, experimentation, and learning purposes. **As of UnoArduSim V2.4 you may find using a 'PROGIO' device instead offers more flexibility than the interrupt-driven methods**

below.

Three possible uses for this device are as follows:

- 1) You can **create a digital input for testing your program** that has more complex timing than can be produced using by any of the set set of supplied standard 'I/O' devices, as follows:

Define an interrupt function (let us call it 'myIntr') and do 'attachInterrupt(0, myIntr, RISING)' inside your 'setup()'. Connect a **Pulser** device to pin2 – now 'myIntr()' will execute every time a **Pulser** rising edge occurs. Your 'myIntr()' function can be an algorithm you have programmed (using global counter variables, and perhaps even 'random()') to produce a waveform of your own design on any available 'OUTPUT' pin (let us say that is pin 9). Now **JUMP** pin 9 to your desired 'Uno' or 'Mega' 'INPUT' pin to apply that generated digital waveform to that input pin (in order to test your program's response to that particular waveform). You can generate a sequence of pulses, or serial characters, or simply edge transitions, all of arbitrary complexity, and varying intervals. Please note that if your main program calls 'micros()' (or calls any function that relies on it), its 'return' value **will be increased** by the time spent inside your 'myIntr()' function every time the interrupt fires. You can produce a quick burst of accurately timed edges using calls to 'delayMicroseconds()' from inside 'myIntr()' (perhaps to generate an entire **byte** of a high-baud-rate transfer), or simply generate one transition per interrupt (perhaps to generate **one bit** of a low-baud-rate transfer) with the **Pulser** device 'Period' chosen appropriate to your timing needs (recall that **Pulser** limits its minimum 'Period' to 50 microseconds).

- 2) You can **experiment with sub-system loop-backs:**

For example, disconnect your 'SERIAL' 'I/O' device TX '00' pin (edit it to a blank), and then **JUMP** 'Uno' or 'Mega' pin '01' back to 'Uno' or 'Mega' pin '00' to emulate a hardware loop-back of the ATmega 'Serial' subsystem.

Now in your test program, inside `'setup()'` do a **single** `'Serial.print()'` of a word or character, and inside your `'loop()'` echo back any characters received (when `'Serial.available()'`) by doing a `'Serial.read()'` followed by a `'Serial.write()'`, and then watch what happens. You could observe that a similar `'SoftwareSerial'` loop-back **will fail** (as it would in real life -- the software cannot do two things at once).

You can also try out **SPI** loop-back by using a **JUMP** to connect pin 11 (MOSI) back to pin 12 (MISO).





3) You can **count the number, and/or measure the spacing of, specific level transitions on any 'Uno' or 'Mega' output pin X** that occur as a result of a complex Arduino instruction or library function (as examples: `'analogWrite()'`, or `'OneWire::reset()'`, or `'Servo::write()'`), as follows:

JUMP pin **X** to interrupt pin **2** and inside your `'myIntr()'` use a `'digitalRead()'` and a `'micros()'` call, and compare to saved levels and times (from previous interrupts). You can change the edge-sensitivity for the next interrupt, if needed, using `'detachInterrupt()'` and `'attachInterrupt()'` from **inside** your `'myIntr()'`. Note that you will not be able to track pin transitions that occur too closely together (closer than the total execution time of your `'myIntr()'` function), such as those that happen with I2C or SPI transfers, or with high baud-rate `'Serial'` transfers (even though your interrupt function would not disturb the inter-edge timing of these hardware-produced transfers). Also note that software-mediated transfers (like `'OneWire::write()'` and `'SoftwareSerial::write()'`) are deliberately protected from interruption (by their library code temporarily disabling all interrupts, in order to prevent timing disruptions), so you cannot measure inside those using this method.






Although you can instead make these same edge-spacing measurements **visually** in a **Digital Waveforms** window, if you are interested in the minimum or maximum spacing over a large number of transitions, or in counting transitions, doing so using this `'myIntr()'`-plus-**JUMP** technique is more convenient. And you can measure, for example, variations in spacing of main-program produced transitions (due to the effect of your software taking varying execution paths of different execution times), to do a kind of program "profiling".

Menus





File:

<u>Load INO or PDE Prog (ctrl-L)</u> 	Allows the user to choose a program file having the selected extension. The program is immediately given a Parse
<u>Edit/View (ctrl-E)</u>	Opens the loaded program for viewing/editing.
<u>Save</u> 	Save the edited program contents back to the original program file.
<u>Save As</u>	Save the edited program contents under a different file name.
<u>Next ('#include')</u> 	Advances the Code Pane to display the next ' #include ' file
<u>Previous</u> 	Returns the Code Pane display to the previous file
<u>Exit</u>	Exits UnoArduSim after reminding user to save any modified file(s).

Find:

<u>Ascend Call Stack</u> 	Jump to the previous caller function in the call-stack – the Variables Pane will adjust to show the local variables for that function
<u>Descend Call Stack</u> 	Jump to the next called function in the call-stack – the Variables Pane will adjust to show local variables for that function
<u>Set Search Text (ctrl-F)</u> 	Activate the Tool-Bar Find edit-box to define the text to be searched for next (and adds the first word from the currently highlighted line in the Code Pane or Variables Pane if one of those has the focus).
<u>Find Next Text</u> 	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).
<u>Find Previous Text</u> 	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:

<u>Step-Into (F4)</u>		Steps execution forward by one instruction, or <i>into a called function</i> .
<u>Step-Over (F5)</u>		Steps execution forward by one instruction, or <i>by one complete function call</i> .
<u>Step-Out-Of (F6)</u>		Advances execution by <i>just enough to leave the current function</i> .
<u>Run-To (F7)</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-Till (F8)</u>		Runs the program until a write occurs to the variable that had the current highlight in the Variables Pane (click on one to establish the initial highlight).
<u>Run (F9)</u>		Runs the program.
<u>Halt (F10)</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.

Options:

<u>Step-Over Structors/Operators</u>	Fly right through constructors, destructors, and operator overload functions during any stepping (i.e. it will not stop inside these functions).
<u>Register-Allocation</u>	Assign function locals to free ATmega registers instead of to the stack (generates somewhat reduced RAM usage).
<u>Error on Uninitialized</u>	Flag as a Parse error anywhere your program attempts to use a variable without having first initialized its value (or at least one value inside an array).
<u>Added 'loop()' Delay</u>	Adds 1000 microseconds of delay every time 'loop()' is called (in case there are no other program calls to 'delay()' anywhere) – useful to try avoiding falling too far behind real-time.
<u>Allow Nested Interrupts</u>	Allow re-enabling with 'interrupts()' from inside a user interrupt service routine.

Configure:

<u>'I/O' Devices</u>	Opens a dialog-box to allow the user to choose the type(s), and numbers, of desired 'I/O' devices. From this dialog-box you can also Save 'I/O' devices to a text file, and/or Load 'I/O' devices from a previously saved (or edited) text file (including all pin connections and clickable settings and typed-in values)
<u>Preferences</u>	Opens a dialog-box to allow the user to set preferences including automatic indentation of source program lines, allowing Expert syntax, choice of font typeface, opting for a larger font size, enforcing of array bounds, permitting of logical operator keywords, showing program download, choice of board version, and TWI buffer length (for I2C devices).

VarRefresh:

<u>Allow Auto (-) Contract</u>	Allow UnoArduSim to contract displayed expanded arrays/objects when falling behind real-time.
<u>Minimal</u>	Only refresh the Variables Pane display 4 times per second.
<u>Highlight Changes</u>	Highlight changed variable values when running (can cause slowdown).

Windows:

<u>Serial Monitor</u>	Connect a serial I/O device to pins 0 and 1 (if none) and pull up a larger ' Serial ' monitor TX/RX text window.
<u>Restore All</u>	Restore all minimized child windows.
<u>Pin Digital Waveforms</u>	Restore a minimized Pin Digital Waveforms window.
<u>Pin Analog Waveform</u>	Restore a minimized Pin Analog Waveform window.

Help:

<u>Quick Help File</u>	Opens the UnoArduSim_QuickHelp PDF file.
<u>Full Help File</u>	Opens the UnoArduSim_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.

'Uno' or 'Mega' Board and 'I/O' Devices

The 'Uno' or 'Mega' and attached 'I/O' devices are all accurately modeled electrically, and you will be able to get a good idea at home of how your programs will behave with the actual hardware, and all electrical pin conflicts will be flagged.

Timing

UnoArduSim 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 (such as '`print()`' or '`SPI.transfer()`' etc.) that will naturally keep it synchronized to real time (see below).

To accomplish this, UnoArduSim 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 when the user has created tight loops **with no added delay**, or 'I/O' devices are configured for operation with very high 'I/O' device frequencies (and/or baud-rate) which would generate an excessive number of pin-level change events and associated processing overload. UnoArduSim copes with this overload by skipping some timer intervals to compensate, and this then slows down program progression to **below real-time**.

In addition, programs with large arrays being displayed, or again 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 – UnoArduSim automatically reduces variable refresh frequency to try to keep up, but when even more reduction is needed, choose **Minimal** , from the **VarRefresh** menu to specify only four refreshes per second.

Accurately modeling 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 the simulated performance of your program will be very close to real-time (again, barring blatantly excessive high-frequency pin-level change events or 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 menu **Execute**.

'I/O' Device Timing

These virtual devices receive real-time signalling of changes that occur on their input pins, and produce corresponding outputs on their output pins which can then be sensed by the 'Uno' or 'Mega' – they are therefore inherently synchronized to program execution. Internal 'I/O' device timing is set by the user (for example through baud-rate selection or Clock frequency), and simulator events are set up to track real-time internal operation.

Sounds

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

As of V2.0, sound has now been modified to use the Qt audio API –unfortunately its QAudioOutput '`class`' does not support looping the sound buffer to avoid running out of sound samples (as can happen during longer OS windowing operational delays). Therefore, in order to avoid the great majority of annoying sound clicks and sound breakup during OS delays, sound is now muted according to the following rule:

Sound is muted for as long as UnoArduSim is not the "active" window (except when a new child window has just been created and activated), **and even** when UnoArduSim is the "active" main window but the mouse pointer is **outside** of its main window client area.

Note that this implies that sound will be temporarily muted as long as the mouse pointer is hovering **over a child window**, and will get muted **if that child window is clicked to activate it** (until the main UnoArduSim window is clicked again to re-activate it).

Sound can always be un-muted by clicking anywhere inside the client area of the UnoArduSim main window.

Due to buffering, sound has a real-time lag of up to 250 milliseconds from the corresponding event time on the pin of the attached 'PIEZO'.

Limitations and Unsupported Elements

Included Files

A '<>' - bracketed '#include' of '<Servo.h>', '<Wire.h>', '<OneWire.h>', '<SoftwareSerial.h>', '<SPI.h>', '<EEPROM.h>' and '<SD.h>' is supported but these are only emulated – the actual files are not searched for; instead their functionality is directly "built into" UnoArduSim, 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', 'SPI.h', 'Wire.h', 'OneWire.h', 'Servo.h', 'Stepper.h', 'SD.h', 'TFT.h' and 'EEPROM.h' for the **Arduino V1.8.8** release are currently supported in UnoArduSimV2.6 . 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 UnoArduSim 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 macro*** '#define'. The '#pragma' directive and conditional inclusion directives ('#ifdef', '#ifndef', '#if', '#endif', '#else' and '#elif') are also ***not supported***. The '#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 – from V2.5 onward

Bug Fixes

V2.8.2 September 2020

- 1) 'analogRead()' was not accepting 'A5' as a valid analog pin number.
- 2) Since V2.6, 'PULSER' radio buttons were not always showing the selected state (but were otherwise working correctly).
- 3) Since V2.4, flipping from a steady 'HIGH' pin level to 'analogWrite(pin, 0)' caused the 0-level change to be missed.
- 4) Since V2.8.1, execution (non-error) warning pop-ups were failing to highlight the problem code-line.
- 5) Stopping an ongoing periodically flipping output pin with 'digitalWrite()' or 'digitalRead()' caused loss of the preceding PWM flips on that pin in the Waveforms window.
- 6) Fixed issues with connecting two 'INPUT' pins via a 'JUMP' device : 'I/O' device pin changes are no longer simply appear instead on the jumpered-to pin (they now appear on both pins), and attached 'PULSER' or 'FUNCGEN' periodic signals are now extended as well to the jumpered-to pin.
- 7) Momentary-close 'PUSH' devices were not reverting to their open state when the mouse pointer left the device.

V2.8.1– June 2020

- 8) Extra blank lines removed by the auto-formatting Preference caused the initially highlighted line in Edit/View to be offset downward by the number of blank lines that were removed above it.
- 9) 'analogRead()' was only accepting the full digital pin number.
- 10) Classes that contained function overloads that differed only in the 'unsigned' attribute of a function argument could have the wrong overload function called. This affected LCD 'print(char/int/long)' where the 'unsigned' base-10 print overload function would get called instead, and this caused LCD 'parseFloat()' to print '46' instead of the '.' decimal point.
- 11) Auto-completion insertion (upon 'Enter') of an already matched built-in did not work after backtracking to correct a typing mistake on the line.
- 12) A 'TFT' device with an empty 'RS' pin could cause a crash on execution.

V2.8.0– June 2020

- 1) When a Parse warning (but not a Parse error) occurred, V2.7 could attempt to highlight the problematic line in the wrong buffer (in the most recent '**#include**' buffer instead), and that would cause a silent crash (even before the warning pop-up appears) if the line number was beyond the end of that '**#include**' buffer.
- 2) The received bytes in the larger monitor windows of 'I2CSLV', 'SPISLV', 'TFT', 'LCDI2C', and 'LCDSPI' devices were still not all reported correctly (this did not affect their functioning).
- 3) Variables of type '**unsigned char**' were promoted to type '**int**' in arithmetic expressions, but improperly maintained their '**unsigned**' status, leading to a fault '**unsigned**' resulting expression.
- 4) Changing (or blanking) the pin of an 'LED4' or '7SEG' device from an initial valid pin setting failed to detach its upper 3 pins and could result in unexplained crashes -- in addition, changes made in V2.7 to harmonize the handling of all 'LED' types completely broke the 'LED4' device.
- 5) An error in changes made for Version 2.6 to support 'LOW' interrupts caused 'FALLING' interrupts attached to pin 3 to corrupt interrupt-level-change sensing on pin 2.

- 6) 'SoftwareSerial' was improperly disabling user interrupts during each character it was receiving.
- 7) When a 'PROGIO' program containing a Parse error was loaded, the error was flagged but that program had already been replaced by a default 'PROGIO' program inside the 'PROGIO' Monitor window.
- 8) Since version V2.4, pin changes on Pin 1 and Pin 0 were not seen during downloading.
- 9) Looping read or write operations on an open SD file caused execution sequencing to fail, resulting in an eventual internal UnoArduSim error due to scope level depth.
- 10) Attempting to change the 'MISO' pin on an 'SD_DRV' device corrupted the MOSI pin.
- 11) All previous versions failed to warn that using '`analogWrite()`' on pins 9 or 10 would corrupt 'Servo' timing for all active 'Servo' devices.
- 12) Typing a different number on top of an already valid 'EN' pin on an 'LCD_D4' device could cause a crash (a partially-typed number would return -1 for the pin value).

V2.7– Mar. 2020

- 1) When the (Windows-default) light OS theme was adopted, the **Code Pane** was not showing the colour-highlighting introduced in V2.6 (instead, only a grey highlight resulting from a system override).
- 2) Version 2.6 inadvertently broke auto-indent-tab formatting at the first '`switch()`' construct.
- 3) The new call-stack navigation feature introduced in Version 2.6 showed **incorrect values** for local variables when not inside the currently executing function, and failed with nested member function calls.
- 4) '`TFT::text()`' was working, but '`TFT::print()`' functions were not (they simply blocked forever). In addition, '`TFT::loadImage()`' failed if '`Serial.begin()`' had been done earlier (which is the normal case, and is now required).
- 5) Version 2.6 introduced a bug that displayed the incorrect value for present and past 'RX' bytes for 'I2CSLV', 'SPISLV', 'TFT', 'LCDI2C' and 'LCDSPI' devices (and their Monitor windows).
- 6) A change made in V2.4 caused **Execute|Animate** highlighting to skip over many executed code-lines.
- 7) Since V2.4, de-asserting 'SS*' or 'CS*' on a 'I/O' device in the instruction immediately following an '`SPI.transfer()`' would cause that device to fail to receive the transferred data byte. In addition, byte reception in '`SPI_MODE1`' and '`SPI_MODE3`' was not flagged until the start of the next byte sent by the master (and the byte was completely lost if the device 'CS*' was de-selected before then).
- 8) In the new '`SPI_SLV`' mode allowed since V2.4, '`bval = SPI.transfer()`' only returned the correct value for '`bval`' if the byte transfer was already complete and waiting when '`transfer()`' was called.
- 9) The 'DATA' edit box on 'SPISLV' devices now gets the default 0xFF when there are no more bytes to reply with.
- 10) The synchronization LED state was incorrect for 'PSTEPR' devices having more than 1 micro-step per full step.
- 11) Electrical conflicts caused by 'I/O' devices reacting to transitions on the 'SPI' clock, a 'PWM' signal, or a '`tone`' signal, were not reported, and could lead to unexplained (corrupted) data reception.
- 12) When the interval between interrupts was too small (less than 250 microseconds), a (faulty) change in V2.4 altered the timing of built-in functions that use either system timers, or instruction loops, to generate delays (examples of each are '`delay()`' and '`delayMicroseconds()`'). A subsequent change in V2.5 caused mis-alignment of '`shiftOut()`' data and clock signals when an interrupt happened between bits.
- 13) Accepting a built-in function auto-completion text via the Enter key failed to strip out parameter types from the text of the inserted function call.
- 14) Loading a new (user-interrupt-driven) program when a previously running program still had an interrupt pending could cause a crash during downloading (due to faulty attempted execution of the new interrupt routine).
- 15) Object-member auto-completions (accessible via 'ALT'-right-arrow) for objects inside '`#include`' files are now accessible as soon as their '`#include`' file is successfully parsed.
- 16) A function declaration having an inadvertent space inside a function parameter name caused an unclear Parse error message.

- 17) When execution halted in a module different from the main program, the File | Previous action failed to become enabled.
- 18) Single-quoted braces (' { ' and ' } ') were still counted (incorrectly) as scope brackets in the Parse, and also confused auto-tab-indent formatting.
- 19) 'OneWire::readBytes(byte* buf, int count)' had been failing to immediately update the displayed 'buf' contents in the Variables Pane.
- 20) Octal-latch 'OWISLV' devices showed output pin levels that lagged by one latch-register write.

V2.6.0– Jan 2020

- 1) A bug introduced in V2.3 led to a crash when the added Close button was used in the **Find/Replace** dialog (rather than its **Exit** title-bar button).
- 2) If a user program did '#include' of other user files, the **Save** button inside **Edit/View** would have failed to actually save a modified file if there was an existing Parse or Execution error flagged inside a different file.
- 3) A **Cancel** after a **Save** could also be confusing -- for these reasons the **Save** and **Cancel** button functionalities have been changed (see **Changes and Improvementts**).
- 4) Unbalanced brackets inside a 'class' definition could cause a hang since V2.5.
- 5) Direct logical testing on 'long' values returned 'false' if none of the 16 lowest bits were set.
- 6) UnoArduSim had been flagging an error when a pointer variable was declared as the loop variable inside the brackets of a 'for()' statement.
- 7) UnoArduSim had been disallowing logical tests that compared pointers to 'NULL' or '0'.
- 8) UnoArduSim had been disallowing pointer arithmetic involving an integer variable (only integer constants had been allowed).
- 9) Interrupts set using 'attachInterrupt(pin, name_func, LOW)' had only been sensed on a transition to 'LOW'.
- 10) When using more than one 'I2CSLV' device, a non-addressed slave could interpret later bus data as matching to its bus (or global-call 0x00) address, and so falsely signal ACK, corrupting the bus ACK level and hanging a 'requestFrom()'.
- 11) Passing numeric value '0' (or 'NULL') as a function argument to a pointer in a function call is now allowed.
- 12) The tabbing indent level after a nesting of 'switch()' constructs was too shallow when the 'auto-indent formatting' choice of **Configure | Preferences** was used.
- 13) Subtraction of two compatible pointers now results in a type of 'int'.
- 14) UnoArduSim had been expecting a user-defined default constructor for a member object even if it had not been declared as 'const'.
- 15) At an execution break, the drawn position of a 'STEPR', 'SERVO', or 'MOTOR' motor could lag by up to 30 milliseconds of motion behind its actual last-computed position.
- 16) 'Stepper::setSpeed(0)' was causing a crash because of a divide-by-zero.
- 17) Single-line 'if()', 'for()', and 'else' constructs no longer cause one too many auto-indentation tabs.

V2.5.0– Oct 2019

- 1) A bug introduced into V2.4 broke 'SD' card initialization (caused a crash).
- 2) Using the 'SPI' subsystem in the new 'SPI_SLV' mode worked improperly in 'SPI_MODE1' and 'SPI_MODE3'.
- 3) Auto-completion pop-ups (as requested by 'ALT-right=arrow') were fixed for functions having object parameters; the pop-ups list now also include inherited (base-)class members, and auto-completions now also appear for 'Serial'.
- 4) Since V2.4 the return value for 'SPI.transfer()' and 'SPI.transfer16()' was incorrect if a user

interrupt routine fired during that transfer.

5) In V2.4, fast periodic waveforms were shown as having a longer duration than their actual duration evident when viewed at higher zoom.

6) The constructor `'File::File(SdFile &sdf, char *fname)'` was not working, so `'File::openNextFile()'` (which relies on that constructor) was also not working.

7) UnoArduSim was incorrectly declaring a Parse error on object-variables, and object-returning functions, declared as `'static'`.

8) Assignment statements with a `'Servo'`, `'Stepper'`, or `'OneWire'` object variable on the LHS, and an object-returning function or constructor on the RHS, caused an internal miscount of the number of associated objects, leading to an eventual crash.

9) Boolean tests on objects of type `'File'` were always returning `'true'` even if the file was not open.

V2.4– May 2019

1) Run-Till watch-points could be falsely triggered by a write to an adjacent variable (1 byte lower in address).

2) Baud rate selections could be sensed when the mouse was inside a `'SERIAL'` or `'SFTSER'` device baud drop-down list-box (even when no baud rate was actually clicked).

3) Since V2.2, serial reception errors occurred at a baud rate of 38400.

4) A change made in V2.3 caused **'SoftwareSerial'** to sometimes erroneously report a disabled interrupt (and so fail upon first RX reception).

5) A change made in V2.3 caused SPISLV devices to misinterpret hex values directly entered into their `'DATA'` edit-box.

6) A change made in V2.3 caused SRSLV devices to sometimes falsely, and silently, detect an electrical conflict on their `'Dout'` pin, and so disallow a pin assignment there. Repeated attempts to attach a pin to `'Dout'` could then lead to an eventual crash once the device was removed.

7) Attempting to attach an `'LED4'` device past pin 16 would cause a crash.

8) A bug triggered by rapid repeated calls to `'analogWrite(255)'` for **'MOTOR'** control in the user program caused resulting **'MOTOR'** speeds to be incorrect (much too slow).

9) Since V2.3, SRSLV devices could not be assigned a `'Dout'` pin due to a faulty electrical conflict detection (and so disallow a pin assignment).

10) SPI slaves now reset their transmitter and receiver logic when their **'SS'** pin goes **'HIGH'**.

11) Calling `'Wire.h'` functions `'endTransmission()'` or `'requestFrom()'` when interrupts are currently disabled now generates an execution error (`'Wire.h'` needs interrupts enabled in order to work).

12) `'Ctrl-Home'` and `'Ctrl-End'` now work as expected in Edit/View.

13) The `'OneWire'` bus command `0x33 ('ROM_READ')` was not working, and hung the bus.

V2.3– Dec. 2018

1) A bug introduced in V2.2 made it impossible to edit the value of an array element inside **Edit/Track**.

2) Since version 2.0, the text in the **'RAM free'** Tool-Bar control was only visible if using a dark Windows-OS theme.

3) On **File| Load**, and on I/O device file **Load**, the file filters (like `'*.ino'` and `'*.txt'`) were not working --files of all types were instead shown.

4) The "modified" state of a file was lost after doing **Accept** or **Compile** in a subsequent **File | Edit/View if no further edits were made** (**Save** became disabled, and there was no automatic prompt to **Save** on program **Exit**).

5) Operators `'/='` and `'%='` only gave correct results for `'unsigned'` left-hand-side variables

6) The ternary conditional `'(bval) ? S1:S2'` gave an incorrect result when `'S1'` or `'S2'` was a local

expression instead of a variable.

- 7) Function `'noTone()'` has been corrected to become `'noTone(uint8_t pin)'`.
- 8) A long-standing bug caused a crash after proceeding from a **Reset** when that Reset was done in the middle of a function that was called with one of its parameters missing (and so receiving a default initializer value).
- 9) Member expressions (e.g. `'myobj.var'` or `'myobj.func()'`) were not inheriting the `'unsigned'` property of their right-hand-side (`'var'` or `'func()'`) and so could not be directly compared or combined with other `'unsigned'` types – an intermediate assignment to an `'unsigned'` variable was first required.
- 10) UnoArduSim was insisting that if a function's definition had any parameter with a default initializer, that the function have an earlier prototype declared.
- 11) Calls to `'print(byte bvar, base)'` mistakenly promoted `'bvar'` to an `'unsigned long'`, and so printed out too many digits.
- 12) `'String(unsigned var)'` and `'concat(unsigned var)'` and operators `'+=(unsigned)'` and `'+(unsigned)'` incorrectly created `'signed'` strings instead.
- 13) An 'R=1K' device loaded from an **IODevices.txt** file with position `'U'` was mistakenly drawn with its slider (always) in the **opposite position** from its true electrical position.
- 14) Attempting to rely on the default `'inverted=false'` argument when declaring a `'SoftwareSerial()'` object caused a crash, and passing `'inverted=true'` only worked if the user program did a subsequent `'digitalWrite(txpin, LOW)'` in order to first establish the required idle `'LOW'` level on the **TX** pin.
- 15) 'I2CSLV' devices did not respond to changes in their pin edit boxes (the A4 and A5 defaults remained in effect).
- 16) 'I2CSLV' and 'SPISLV' devices did not detect and correct partial edits when the mouse left their borders
- 17) Pin values for devices that followed an SPISLV or SRSLV were improperly saved to the **IODevs.txt** file as hexadecimals.
- 18) Trying to connect more than one SPISLV device MISO to pin 12 always generated an Electrical Conflict error.
- 19) Switching a pin from `'OUTPUT'` back to `'INPUT'` mode failed to reset the pin data latch level to `'LOW'`.
- 20) Using `'sendStop=false'` in calls to `'Wire.endTransmission()'` or `'Wire.requestFrom()'` caused a failure.
- 21) UnoArduSim improperly allowed a `'SoftwareSerial'` reception to occur simultaneous with a `'SoftwareSerial'` transmission.
- 22) Variables declared with `'enum'` type could not be assigned a new value after their declaration line, and UnoArduSim was not recognizing 'enum'-members when referred to with a (legal) `'enumname::'` prefix.

V2.2– Jun. 2018

- 1) Calling a function with fewer arguments than was needed by its definition (when that function was “forward-defined”, that is, when it had no earlier prototype declaration line) caused a memory violation and crash.
- 2) Since V2.1, **Waveforms** had not been getting updated during **Run** (only at **Halt**, or after a **Step**) – in addition, **Variables Pane** values were not getting updated during lengthy **Step** operations.
- 3) Some minor **Waveform** scrolling and zooming issues that have existed since V2.0 have now been fixed.
- 4) Even in early versions, the Reset at $t = 0$ with a PULSER or FUNCGEN, whose period would make its very last cycle prior to $t = 0$ be only a *partial* cycle, resulted in its **Waveform** after $t = 0$ being shifted from its true position by this (or the remaining) fractional cycle amount, either to the right or to the left (respectively).
- 5) Fixed some issues with syntax-colour highlighting in **Edit/View**.
- 5) Since V2.0, clicking to expand one object in an array of objects did not work properly.
- 6) `'delay(long)'` has been corrected to be `'delay(unsigned long)'` and `'delayMicroseconds(long)'` has been corrected to be `'delayMicroseconds(unsigned int)'`.
- 7) As of V2.0, functions attached using `'attachInterrupt()'` were not getting checked as being valid functions for that purpose (i.e. `'void'` return, and having no call parameters).
- 8) The impact of `'noInterrupts()'` on functions `'micros()'`, `'mills()'`, `'delay()'`,

'pulseInLong()' ; its impact upon 'Stepper::step()' and upon 'read()' and 'peek()' timeouts; upon all RX serial reception, and upon 'Serial' transmission, is now accurately reproduced.

9) Time spent inside user interrupt routines is now accounted for in the value returned by 'pulseIn()', the delay produced by 'delayMicroseconds()', and in the position of edges in displayed **Pin Digital Waveforms**.

10) Calls to **object-member** functions that were part of larger complex expressions, or were themselves inside function calls having multiple complex arguments, e.g, 'myobj.memberfunc1() + count/2' or 'myfunc(myobj.func1(), count/3)', would have incorrect values computed/passed at execution time due to faulty stack-space allocations.

11) Arrays of pointer variables worked properly, but had faulty displayed values shown in the **Variables Pane**.

12) When dynamic arrays of simple type were created with 'new', only the first element had been getting a (default) initialization to value 0 – now all elements do.

13) 'noTone()', or the end of a finite tone, no longer resets the pin (it remains 'OUTPUT' and goes 'LOW').

14) Continuous-rotation 'SERVO' devices are now perfectly stationary at 1500 microseconds pulse width.

15) Calling on SdFile::ls() (SD card directory listing) worked properly, but improperly showed some duplicated block SPI transfers in the Waveforms window.

V2.1.1– Mar. 2018

- 1) Fixed inconsistencies in non-English locales with the language saved to 'myArduPrefs.txt', with displayed radio language buttons in the **Preferences** dialog-box, and with matching to translated lines in 'myArduPrefs.txt'.
- 2) Allocations with 'new' now accept an integer array dimension that is not a constant.
- 3) Clicking in the **Variables Pane** to expand a multi-dimensional array would show a superfluous empty '[]' bracket-pair .
- 4) Array-element references with trailing superfluous characters (e.g. 'y[2]12') were not caught as errors at Parse time (the extra characters were simply being ignored).

V2.1– Mar. 2018

- 1) A bug in the new versions V2.0.x caused the Windows heap to grow with each update in the **Variables Pane** -- after millions of updates (many minutes worth of execution), a crash could result.
- 2) Calls to 'static' member functions using double-colon '::' notation failed to Parse when inside 'if()', 'while()', 'for()', and 'switch()' brackets, and when inside expressions used as function-call arguments or array indices.

V2.0.2 Feb. 2018

- 1) A bug introduced in V2.0 caused a **FileLoad** crash if an '#include' referred to a missing or empty file
- 2) Inside an **IODEVS.txt** file, the 'I/O' name 'One-Shot' was expected instead of the older 'Oneshot' ; both are now accepted.

V2.0.1– Jan. 2018

- 3) In non-English language locales, 'en' was incorrectly shown as selected in **Preferences**, making reverting to English awkward (requiring de-selection then re-selection).
- 4) It had been possible for the user to leave a Device pin edit box value in an incomplete state (like 'A_'), and to leave the 'DATA' bits of an 'SRS:V' incomplete.
- 5) The maximum number of Analog Sliders had been limited to 4 (corrected now to be 6).
- 6) UnoArduSim no longer insists on '=' appearing in an array aggregate initialization.
- 7) UnoArduSim had insisted the "inverted_logic" argument be provided to 'SoftwareSerial()' .
- 8) Bit-shift operations now allow shifts longer than the size of the shifted variable.

V2.0– Dec. 2017

- 1) All functions that were declared as '**unsigned**' had nevertheless been returning values as if they were '**signed**'. This had no effect if the '**return**' value was assigned to an '**unsigned**' variable, but would have caused an improper negative interpretation if it had MSB==1, and it was then assigned to a '**signed**' variable, or tested in an inequality.
- 2) Analog Sliders were only reaching a maximum '**analogRead()**' value of 1022, not the correct 1023.
- 3) A bug inadvertently introduced back in V1.7.0 in logic used to speed up the handling of the SPI system SCK pin caused SPI transfers for '**SPI_MODE1**' and '**SPI_MODE3**' to fail after the first byte transferred (a spurious extra SCK transition followed each byte). Also updates to an 'SPISLV' edit 'DATA' box for bytes transferred were delayed.
- 4) The Coloured LED device was not listing 'B' (for Blue) as a colour option (even though it was accepted).
- 5) Settings for 'SPISLV' and 'I2CSLV' devices were not being saved to the user '**I/O Devices**' file.
- 6) Copying '**Servo**' instances failed due to a faulty '**Servo::Servo(Servo &toCopy)**' copy-constructor implementation.
- 7) Out of range '**Servo.writeMicroseconds()**' values were properly detected as an error, but the stated limit values accompanying error message text were wrong.
- 8) A legal baud-rate of 115200 was not accepted when loaded from an '**I/O Devices**' text file.
- 9) Electrical pin conflicts caused by an attached Analog Slider device were not always detected.
- 10) In rare instances, passing a faulty string pointer (with the string 0-terminator missing) to a '**String**' function could cause UnoArduSim to crash.
- 11) The **Code Pane** could highlight the current Parse error line in the **wrong** program module (when '**#include**' was used).
- 12) Loading an '**I/O Devices**' file that had a device that would (improperly) drive against 'Uno' pin 13 caused a program hang at the error message pop-up.
- 13) UnoArduSim had mistakenly allowed the user to do pasting of non-hex characters into the expanded TX buffer windows for SPISLV and I2CSLV.
- 14) Declaration-line initializations failed when the right-hand-side value was the '**return**' value from an object member-function (as in '**int angle = myservol.read();**').
- 15) '**static**' member variables having explicit '**ClassName::**' prefixes were not recognized if they appeared at the very start of a line (for example, in an assignment to a base- 'class' variable),
- 16) Calling '**delete**' on a pointer created by '**new**' was only recognized if function parenthesis notation was used, as in '**delete(pptr)**'.
- 17) UnoArduSim implementation of '**noTone()**' was incorrectly insisting that a pin-argument be supplied.
- 18) Changes that increased global 'RAM' bytes in a program that used '**String**' variables (via **Edit/View** or **File | Load**), could lead to corruption in that 'Uno' global space due to heap deletion of the '**String**' objects belonging to the old program while using (incorrectly) the heap belonging to the new program. In some circumstances this could lead to a program crash. Although a second Load or Parse solved the problem, this bug has at last been fixed.
- 19) The return values for '**Wire.endTransmission()**' and '**Wire.requestFrom()**' had both been stuck at 0 -- these have now been fixed.

Changes/Improvements

V2.8.2- September 2020

- 1) To avoid confusion, UnoArduSim now allows '<>>' triangle brackets and double-quotes to be used interchangeably around the file name in '#include' statements for both user-local and '3rdParty' files.
- 2) The 'MOTOR' device can now accept three hidden values from an 'IODevs.txt' file: 'F' or 'B' (freewheel- coast, or Braking mode), then constant-load-torque as a percentage of stall torque, then load moment of inertia as a multiple of the intrinsic 'MOTOR' rotor inertia.
- 3) UnoArduSim now accepts 'long int' as synonymous with 'long'.

V2.8.0- June 2020

- 4) Added new 'Mega2560' board **Preferences** option (Board number==10). This features many more 'I/O' pins, 4 more external interrupts (pins 18, 19, 20, 21), three more '**HardwareSerial**' ports (on pins 22-27), and RAM memory increased from 2 KBytes to 8 Kbytes..
- 5) The 'SFTSER' device was renamed to 'ALTSER' (since it can now also be used with 'Serial1', 'Serial2', and 'Serial3').
- 6) Clicking inside a device pin edit box now selects the whole number to make entry easier, and clicking inside a devices-number edit box in Configure'I/O' Devices does the same.
- 7) Added orange highlighting of the relevant source line for Parse and Execution warning pop-ups.
- 8) Auto-formatting now deletes blank lines that occur after keyword lines, or that follow preceding blank lines.
- 9) Added support for '**digitalPinToInterrupt()**' calls.
- 10) Classes now always receive a default constructor if they have no user-specified constructor (even if they do not have a base class, or object members).

V2.7 Mar. 2020

- 1) In addition to the current code-line (green if ready to run, red if error), UnoArduSim now maintains, for each module, the last user-clicked or stack-navigation code-line (highlighted with a dark olive background), making it easier to set and find temporary breakpoint lines (one per module is now allowed, but only the one in the currently displayed module is in effect at a 'Run-To').
- 2) Added new 'I/O' devices (and supporting 3rd-party library code), including 'SPI' and 'I2C' **Expansion Ports**, and 'SPI' and 'I2C' **Multiplexer LED** Controllers and displays (LED arrays, 4-alphanumeric, and 4-digit or 8-digit 7-segment displays).
- 3) '**Wire**' operations are no longer disallowed from inside user interrupt routines (this supports external interrupts from an 'I2C' Expansion Port).
- 4) Digital Waveforms now show an intermediate level (between '**HIGH**' and '**LOW**') when the pin is not being driven.
- 5) To avoid confusion when stepping through over single '**SPI.transfer()**' instructions, UnoArduSim now makes sure that attached 'I/O' devices now receive their (logic-delayed) final 'SCK' clock edge before the function returns.
- 6) When the auto-tab-formatting **Preference** is enabled, typing a closing brace '}' in **Edit/View** now causes a jump to the tab indent position of its matching opening-brace '{' partner.
- 7) A **Re-Format** button has been added to **Edit/View** (to cause immediate auto-tab-indent re-formatting) -- this button is only enabled when the auto-tab-indent Preference is enabled.
- 8) A clearer error message now occurs when a Prefix keyword (like '**const**', '**unsigned**', or 'PROGMEM') follows an identifier in a declaration (it needs to precede the identifier).
- 9) Initialized global variables, even when never used later, are now always assigned a memory address, and so will

appear visible.

V2.6.0 Jan. 2019

- 1) Added **Character-LCD** display devices having 'SPI', 'I2C', and 4-bi-parallel interface. Supporting library source code has been added to the new installation 'include_3rdParty' folder (and can be accessed by using a normal `'#include'` directive) – users may alternatively choose to instead write their own functions to drive the LCD device.
- 2) **Code Pane** highlighting has been improved, with separate highlight colours for a ready code-line, for an error code-line, and for any other code-line.
- 3) The **Find** menu and tool-bar 'func' actions (previous-up and next-down) no longer jump to the previous/next function starting line, and instead now ascend (or descend) the call-stack, highlighting the relevant code-line in the caller (or called) function, respectively, where the **Variables Pane** content is adjusted to show variables for the function containing the currently highlighted code-line.
- 4) To avoid confusion, a **Save** done inside **Edit/View** causes an immediate re-**Compile**, and if the Save was successful, using a subsequent Cancel or Exit will now only revert text to that last-saved text.
- 5) Added a pulsed-input Stepper Motor ('PSTEPR') with 'STEP' (pulse), 'EN*' (enable), and 'DIR' (direction) inputs, and a micro-steps-per-step setting of (1,2,4,8, or 16).
- 6) Both 'STEPR' and 'PSTEPR' devices now have a 'sync' LED (GREEN for synchronized, or RED when off by one or more steps.)
- 7) 'PULSER' devices now have a choice between microseconds and milliseconds for 'Period' and 'Pulse'.
- 8) Built-in-function auto-completions no longer retain the parameter type in front of the parameter name.
- 9) When switching back to a previous **Code Pane**, its previously highlighted line is now re-highlighted.
- 10) As an aid to setting a temporary break-point, using Cancel or Exit from **Edit/View** leaves the highlight in the **Code Pane** on the line last visited by the cursor in **Edit/View**.
- 11) A user-defined (or 3rd party) **'class'** is now allowed to use **'Print'** or **'Stream'** as its base class. In support of this, a new 'include_Sys' folder has been added (in the UnoArduSim installation folder) that provides the source code for each base **'class'**. In this case, calls to such base-**'class'** functions will be treated identically to user code (that can be stepped into), instead of as a built-in function that cannot be stepped into (such as `'Serial.print()'`).
- 12) Member-function auto-completions now include the parameter name **instead of** its type.
- 13) The UnoArduSim Parse now allows an object name in a variable declaration to be prefaced by its optional (and matching) **'struct'** or **'class'** keyword, followed by the **'struct'** or **'class'** name.

V2.5.0 Oct 2019

- 1) Added support for **'TFT.h'** library (with the exception of `'drawBitmap()'`), and added an associated 'TFT' 'I/O' Device (128 by 160 pixels). Note that in order to avoid excessive real-time lags during large **'fillXXX()'** transfers, a portion of the 'SPI' transfers **in the middle of the fill** will be absent from the 'SPI' bus.
- 2) During large file transfers through 'SD', a portion of the 'SPI' transfers in the middle of the sequence of bytes will similarly be absent from the 'SPI' bus.
- 3) Decreased **'Stream'**-usage overhead bytes so that **'RAM free'** value more closely matches Arduino compilation.

- 4) UnoArduSim now warns the user when a `'class'` has multiple members declared on one declaration line.
- 5) Using 'File | Save As' now sets the current directory that that saved-into directory.
- 6) The two missing `'remove()'` member functions have been added to the `'String'` class.
- 7) UnoArduSim now disallows constructor base-calls in a constructor-function prototype unless the full function body definition immediately follows (so as to agree with the Arduino compiler).
- 8) Edge transition time of digital waveforms was reduced to support visualization of fast 'SPI' signals at highest zoom.
- 9) UnoArduSim now allows some constructors to be declared `'private'` or `'protected'` (for internal class use).

V2.4 May 2019

- 1) All 'I/O' device files are now saved in language-translated form, and along with the **Preferences** file, are all now saved in UTF-8 text encoding to avoid matching errors on subsequent read.
- 2) Added a new **'PROGIO'** Device which is a bare programmable slave 'Uno' board that shares up to 4 pins in common with the **Lab Bench Pane** master 'Uno', -- a slave 'Uno' can have no 'I/O' devices of its own.
- 3) You can now delete any 'I/O' device by clicking on it while pressing the 'Ctrl' key.
- 4) In **Edit/View**, text auto-completion has been added for global, built-ins and member variables and functions (use ALT-right-arrow to request a completion, or **Enter** if the Built-ins list is currently highlighting a matching selection).
- 5) In **Preferences**, a new choice allows automatic insertion of a line-terminating semicolon upon n **Enter** key-press (if the current line is an executable statement that seems self-contained and complete).
- 6) Pressing **'Ctrl-S'** from a **Waveform** window allows you to save to a file all **(X,Y)** points along the displayed section of each waveform (where X is microseconds from the leftmost waveform point, and Y is volts).
- 7) A 'SFTSER' device now has a hidden (optional) `'inverted'` value (*applies to both TX and RX*) that can be appended after its baud rate value at the end of its line in an **IODevs.txt** file.
- 8) Added the `'SPISettings'` class, functions `'SPI.transfer16()'`, `'SPI.transfer(byte* buf, int count)'`, `'SPI.beginTransaction()'`, and `'SPI.endTransaction()'`, as well as `'SPI.usingInterrupt()'` and `'SPI.notUsingInterrupt()'`.
- 9) Added SPI library functions `'SPI.detachInterrupt()'` along with an SPI library extension `'SPI.attachInterrupt(void myISRfunc)'` (instead of the actual library function `'SPI.attachInterrupt(void)'` (so as to avoid needing to recognize generic `'ISR(int vector)'` low-level vectored interrupt function declarations).
- 10) The SPI system can now be used in slave mode, either by making the **'SS'** pin (pin 10) an **'INPUT'** pin and driving it **'LOW'** after `'SPI.begin()'`, or by specifying `'SPI_SLV'` as the optional `'mode'` parameter in `'SPI.begin(int mode=SPI_MSTR)'` (another UnoArduSim extension to `'SPI.h'`). Received bytes can then be collected by using `'rxbyte = SPI.transfer(tx_byte)'` either inside a non-SPI-interrupt function or inside a user interrupt service function previously attached by `'SPI.attachInterrupt(myISRfunc)'`. In slave mode, `'transfer()'` waits until a data byte is ready in SPDR (so will normally block waiting for a complete byte reception, but in an interrupt routine it will **return** immediately because the received SPI byte is already there). In either case, `'tx_byte'` gets placed into the SPDR, so will be received by the attached master SPI at its next `'transfer()'`.

- 11) Slave mode support has been added to the UnoArduSim implementation of 'Wire.h'. Function `'begin(uint8_t slave_address)'` is now available, as are `'onReceive(void*)'` and `'onRequest(void*)'`.
- 12) `'Wire.end()'` and `'Wire.setClock(freq)'` can now be called; the latter to set the SCL frequency with a `'freq'` value of either 100,000 (the default standard-mode SCL frequency) or 400,000 (fast-mode).
- 13) 'I2CSLV' devices now all respond to the 0x00 general-call bus-address, and so 0x00 may no longer be chosen as a unique I2C bus address for one of those slaves.
- 14) The modeled execution delays of basic integer and assignment operations and array and pointer operations have been reduced, and 4 microseconds are now added for each floating point operation.

V2.3 Dec. 2018

- 1) Tracking has now been enabled on the **Tool-Bar** 'I/O ____S' **slider** for continuous and smooth scaling of 'I/O' device values that the user has added the suffix 'S'.
- 2) A new **'LED4'** 'I/O' device (row of 4 LEDs on **4 consecutive pin numbers**) has been added.
- 3) A new **'7SEG'** 'I/O' device (7-segment LED digit with hexadecimal code on **4 consecutive pin numbers**, and with active-low **CS*** select input), has been added.
- 4) A new **'JUMP'** 'I/O' device that acts like a wire jumper between two 'Uno' pins has been added. This allows an **'OUTPUT'** pin to be wired to an **'INPUT'** pin (see the device above for possible uses of this new feature).
- 5) A new **'OWISLV'** 'I/O' device has been added, and the third-party `'<OneWire.h>'` library can now be used with `'#include'` so that user programs can test interfacing to a small subset of '1-Wire' bus devices.
- 6) The **Execute** menu **Reset** command is now connected to the **Reset** button.
- 7) For more clarity, when **Artificial 'loop()' delay** is selected under the **Options** menu, an explicit `'delay(1)'` call is added to the bottom of the loop inside `'main()'` -- this is now a real delay that can be interrupted by user interrupts attached on 'Uno' pins 2 and 3.
- 8) Electrical pin conflicts with open-drain, or CS-selected, 'I/O' devices (e.g. I2CLV, or SPISLV) are now declared **only when a true conflict occurs at execution time**, rather than causing an immediate error when the device is first connected.
- 9) Function `'pulseInLong()'` is now accurate to 4-8 microseconds to agree with Arduino (the previous accuracy was 250 microseconds).
- 10) Errors flagged during initialization of a global variable now highlight that variable in the **Code Pane**.

V2.2 Jun. 2018

- 1) On **Save** from either the **Preferences** dialog-box, or from **Configure | 'I/O' Devices**, the **'myArduPrefs.txt'** file is now saved into the directory of the currently loaded program -- every subsequent **File | Load** then automatically loads the file, along with its specified IODEvs file, from that same program directory.
- 2) Function `'pulseInLong()'` **had been** missing, but has now been added (it relies on `'micros()'` for its measurements).
- 3) When a user program does an `'#include'` of a `'*.h'` file, UnoArduSim now also automatically tries to load the corresponding `'*.c'` file **if** a corresponding `'*.cpp'` file was not found.
- 4) Automatic insertion of a close-brace `'}'` (after each open-brace `'{'`) has been added to **Preferences**.
- 5) A new **Options** menu choice now allows `'interrupts()'` to be called from inside a user interrupt routine -- this is for educational purposes only, since nesting of interrupts should be avoided in practice.
- 6) Type-cast of pointers to an `'int'` value is now supported (but a warning pop-up message will appear).
- 7) UnoArduSim now supports labelled program lines (e.g. `'LabelName: count++;'` for user convenience (but `'goto'` is still **disallowed**)).

8) Execution warnings now occur when when a call to `'tone()'` could interfere with active PWM on pins 3 or 11, when `'analogWrite()'` would interfere with a Servo already active on the same pin, when a serial character arrival is missed because interrupts are currently disabled, and when interrupts would be coming so fast that UnoArduSim will miss some of them.

V2.1 Mar. 2018

- 1) Displayed **Variables Pane** values are now refreshed only every 30 milliseconds (and the Minimal option can still reduce that refresh rate further), but the **VarRefresh** menu option to disallow update reduction has been removed.
- 2) Operations that target only a portion of the bytes of a variable value (such as those made through pointers) now cause the change to that that variable value to be reflected in the **Variables Pane** display.

V2.0.1 Jan. 2018

- 1) Undocumented Arduino functions `'exp()'` and `'log()'` have now been added.
- 2) 'SERVO' devices can now be made continuous-rotation (so pulse width controls speed instead of angle).
- 3) In **Edit/View**, a closing brace `'}'` is now automatically added when you type an opening brace `'{'` if you have selected that **Preference**.
- 4) If you click the **Edit/View** window title bar `'x'` to exit, you are now given a chance to abort if you had modified, but not saved, the displayed program file.

V2.0 Sept. 2017

- 1) Implementation has been ported to QtCreator so the GUI has some minor visual differences, but no functional differences other than some improvements:
 - a) The status line messaging at the bottom of the Main window, and inside the **Edit/View** dialog-box, has been improved and a highlight colour-coding added.
 - b) The vertical space allocated between the **Code Pane** and **Variables Pane** is now adjustable through a drag-able (but not visible) splitter bar at their shared border.
 - c) 'I/O' device edit-box values are now only validated once the user has moved the mouse pointer outside the device – this avoids awkward automatic changes to enforce legal values while the user is typing.
- 2) UnoArduSim now supports multiple languages via **Configure | Preferences**. English can always be selected, in addition to the language for the user locale (as long as a custom *.qm translation file for that language is present in the UnoArduSim 'translations' folder).
- 3) Sound has now been modified to use the Qt audio API – this has required muting in certain circumstances in order to avoid annoying sound breakup and clicks during the longer OS windowing operational delays caused by normal user mouse clicks – see the -Sounds section for more detail on this.
- 4) As a user convenience, blanks are now used to represent a 0 value in the device-count edit-boxes in **Configure | 'I/O' Devices** (so you can now use the space-bar to remove devices).
- 5) The un-scaled (U) qualifier is now optional on 'PULSER', 'FUNCGEN' and '1SHOT' devices (it is the assumed default).
- 6) UnoArduSim now allows (in addition to literal numeric values) `'const'` integer-valued variables, and `'enum'`

