UnoArduSimV1.2 Help

Overview

Program Code Pane (and View/Edit)

  Code Pane

  View/Edit

Variables Pane

Lab Bench Pane

  The Uno

  I/O Devices

    Serial Monitor (SERIAL)
    Software Serial (SFTSER)
    SD Disk Drive(SD_DRV)
    Servo Motor (SERVO)
    DC Motor (MOTOR)
    Digital Pulser (PULSER)
    Analog Function Generator (FUNCGEN)
    Analog Slider
    SPI Slave (SPISLV)
    Two-Wire I^{2}C Slave (I2CSLV)
    Push Button (PUSH)
    Slide Switch Resistor (R=1K)
    Piezoelectric Speaker (PIEZO)
    Red LED (LED)

  Menus

    File menu commands:

```
    Load INO or PDE Prog
    View/Edit
    Save
    Save As
    Next (#include) file
    Previous
    Exit
```

Find menu commands:

Prompt
⏬ Find Next Function/Var
⏫ Find Previous Function/Var
🔭 Set Search Text
⏬ Find Next Text
⏫ Find Previous Text

Execute menu commands:

🔲 Step Into (F2
🔲 Step Over (F4)
🔲 Step Out Of
🔲 Run To
🔲 Run
🛑 Halt
⏬ Reset
Animate
Slow Motion

Options menu commands:

Step Over Structors/Operators
Register-Allocation Modelling
Error on Uninitialized
Show Program Download
Bigger Font

Configure menu commands:

I/O Devices
Preferences

VarUpdates menu commands:

Allow Reduction
Minimal Updates
HighLight Updates

Windows menu commands:

Serial Monitor
Restore All
Prompt
Pin Digital Waveforms
Pin Analog Waveform

Help menu commands:

Quick Help File
Full Help File
Bug Fixes
Change/Improvements
About

Overview

UnoArduSim is a freeware **real-time** 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** 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 in the **LabBench Pane**. -- you don't 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, Halt, and flexible Stepping in the **Code Pane**, with a simultaneous view of all global and currently-active local variables, arrays, and objects in the **Variables Pane.** Run-time array-bounds checking is provided, and 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 parsed, and "compiled" into a tokenized executable which is then ready for **simulated execution** (unlike Arduino.exe, a standalone binary executable is *not* created) Any parse error is detected and flagged by highlighting the line that failed to parse, and reporting the error on the **Status Bar** at the very bottom of the 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 mis-matched baud rates) are reported on the Status bar, and via a pop-up MessageBox.

UnoArduSim V1.2 is a substantially complete implementation of the Arduino programming language V1.0.6 **as documented at arduino.cc**.'s Language Reference web page, and with additions as noted in the version's Download page Release Notes. Although UnoArduSim does not support the full C++ implementation that Arduino.exe's underlying GNU compiler does, it is likely that only the most advanced programmers would find that some C/C++ element they wish to use is missing (and of course there are always simple coding work-arounds for such missing features). In general, I have supported only what I feel are the most useful C/C++ features for Arduino hobbyists and students -- for example, `enum`'s and `#define`'s are supported, but function-pointers are not. Even though user-defined objects (classes and structs) and (most) operator-overloads are supported, *multiple-inheritance is not*.

Because 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 Options menu).

As of V1.2, UnoArduSim has built-in automatic support for a limited subset of the Arduino provided libraries, these being: SD.h, Servo.h, SoftwareSerial.h, SPI.h, Wire.h, and EEPROM.h. For any `#include`'d 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 your are working on (subject of course to the limitation that the contents of any `#include`'d file(s) must be fully understandable to UnoArduSim's parser).

I developed UnoArduSIm in Microsoft Visual C++, and it is currently only available for Windows[TM]. Porting to Linux or MacOS, or to Java, 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-causing-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, Ph.D, P.Eng.
Associate Professor (retired)
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
Jun 2015

Program Code Pane (and View/Edit)

Code Pane

The Code Pane displays your user program, and highlighting tracks its execution.
After a loaded program is successfully parsed, 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 Execute menu and associated toolbar icons and keyboard shortcuts.
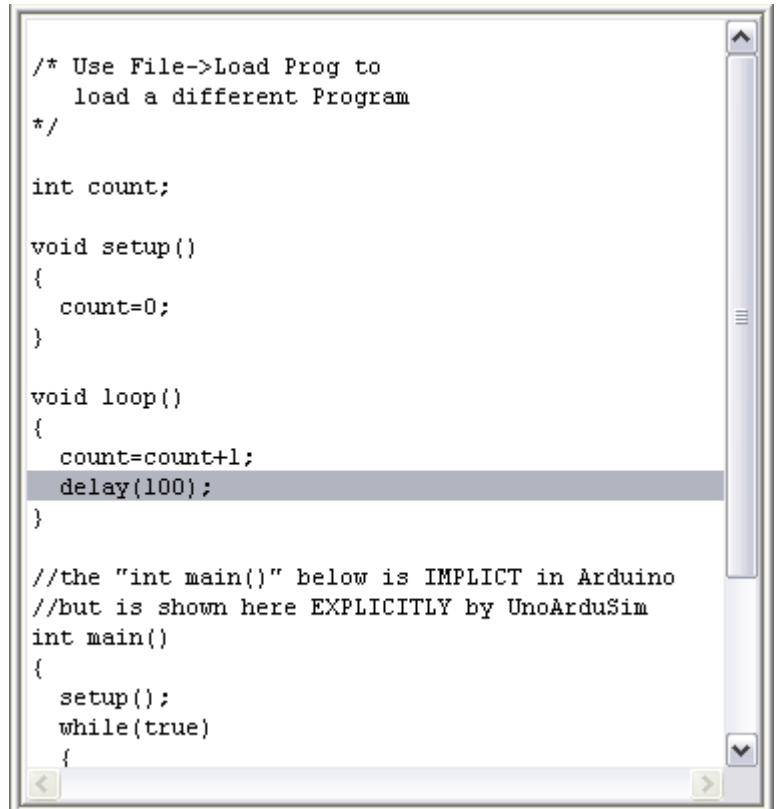
After stepping execution by one (or more) instructions, the next-to-be executed program line is then highlighted (the highlighted line is always the next line **ready to be executed using** ![icon], ![icon], ![icon], **or** ![icon]).

Similarly, when a running program hits a (temporary Run-To) breakpoint, execution is halted and the breakpoint line is highlighted (and is then ready for execution). When a program is running, UnoArduSim periodically highlights the current program line so you can see that some activity is happening. This highlighting will cause the **Code Pane** window contents to scroll to the keep highlighted line visible.

If program execution is currently halted, and you click in the **Code Pane** window, the line you just clicked becomes highlighted. This does *not*, however, change the current program line as far as program execution is concerned. But you can cause execution *to progress up to* the line you just highlighted by then clicking the **RunTo** ![icon] toolbar 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.

```
/* 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 IMPLICT in Arduino
//but is shown here EXPLICITLY by UnoArduSim
int main()
{
  setup();
  while(true)
  {
```

If your loaded program has **#include**'d files, you can move between them using the File menu items File→Prev File and File→ Next File for this, or by using the associated left and right blue arrow toolbar buttons ![icon], ![icon].

The Find menu (with shortcuts PgDn and PgUp or ![icon] and ![icon]) allows you to quickly jump between *functions* in the Code Pane (but you must first click a ine inside the Code Pane to give it focus). Or you can jump to specified text (after first using Find->Set Search text, or ![icon]) with the menu's text-search commands, or, more simply, the ![icon] and ![icon] toolbar icons.

View/Edit

By double-clicking on any line in the Code Pane (or using the File menu), a View/Edit window is opened to allow changes to your program file, with the Code Pane's **currently selected line** highlighted..
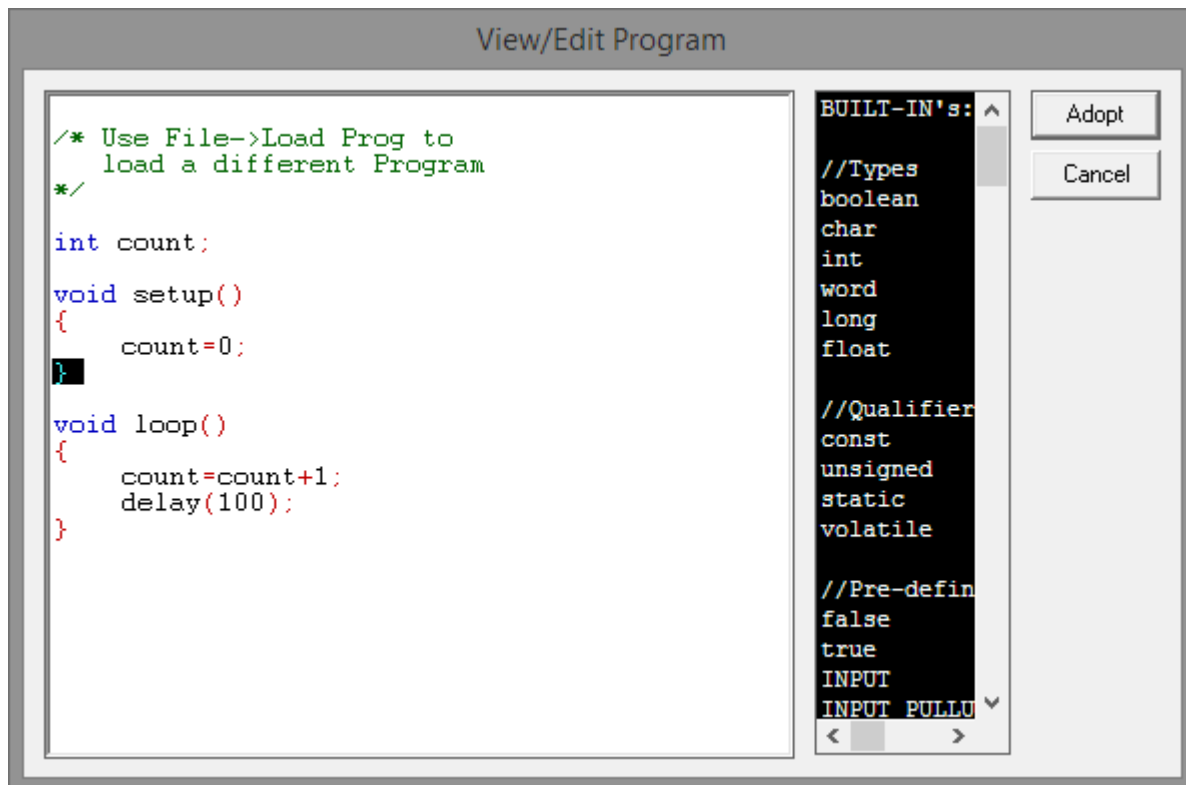
This window has full edit capability with dynamic syntax-highlighting (different highlight colours for C++ keywords, comments, etc.), and automatic indent-level formatting (assuming you have selected that using the Config→Preferences dialog). You can also conveniently select built-in function calls (or `#define`'d constants) to be added into your program from the provided ListBox (just double-click on the desired ListBox item to add it to your program at the current caret position – function-call variable types are just for information and are stripped out (leaving dummy placeholder variables) when added to your program).

The window has **Find** capability (use **ctrl-F**), and **Find/Replace** capability (use **ctrl-H**). The Edit/View window has an **UnDo** button (or use **ctrl-Z**) so you can sequentially undo any changes you have made since opening the Edit/View window (but there is *no* ReDo provided). All changes made on the same program line count as a single undo, as do any Replace-All operations. To discard all changes you made since you first opened the program for editing, hit the **Cancel** button. To adopt all changes, click the **Adopt** button and the program is automatically re-parsed, and the new status appears in the **Status Bar**.

On either **Cancel** or **Adopt** with no edits made, the Code Pane current line changes to become the *last View/Edit 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 **CTRLPgUp** 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). To avoid jumping to blank lines inside functions (or elsewhere), you can simply place a double-slash `//` empty-comment there so that the line is no longer "empty". You can use **CTRL-Home** and **CTRL-End** to jump to the program start, and end, respectively.

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

Variables Pane

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 also not displayed.
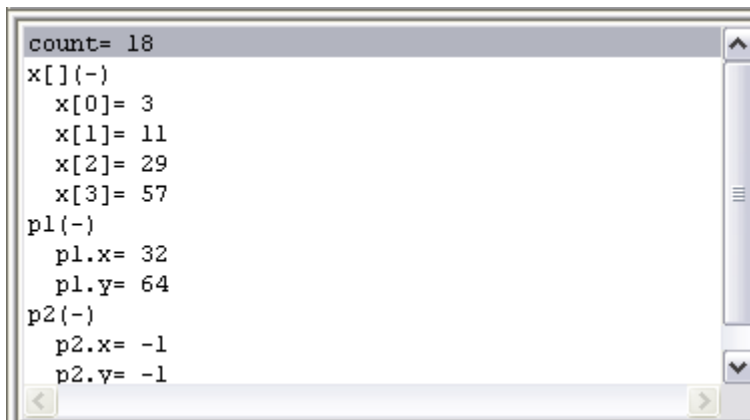
The Find menu (with shortcuts **PgDn** and **PgUp** or ⬇func and func⬆) allows you to quickly jump between *variables* in the variables Pane (but you must first click a line inside the Pane to give it focus). Or you can jump to specified text (after first using **Find→Set Search** text or 🔍) with the menu's text-search commands, or, more simply, the ⬇text and text⬆ toolbar icons.

Arrays and **objects** are shown in either **unexpanded** or **expanded** format, with either a trailing plus (+) or minus(-) sign, respectively. The expanded/unexpanded 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 collapse back to an unexpanded view, click on the **x[](-)**. The unexpanded 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 collapse back to an unexpanded view, single click on **p1(-)**.

When **Step-**ping or **Animate**-ing a program, changes to the value of a variable cause its displayed value in the Variables Pane to be updated immediately and it becomes highlighted-- this will cause the Variable pane to scroll (if needed) to the line that holds that variable. When **Run-**ning, variables updates are made according to user settings made under the **VarUpdates** menu – this allows a full range of behaviour from minmal periodic updates to full immediate updates – reduced or minimal updates are useful to reduce CPU load and may be needed to keep execution tracking real-time under what would otherwise be excessive window-update loads. And when **Run-**ning, highlighting is only done if that **VarUpdates** menu option was selected.

This window also gives you *the ability to change any variable to a new value in the middle of (halted) program execution* to test what would be the effect of continuing on ahead with that new value. Halt execution first, then *left-double-click* on the variable whose value you wish to change; fill in the Edit box, and close to adopt the new value. Resume program execution (Step or Run) to use the new value from that point forward.
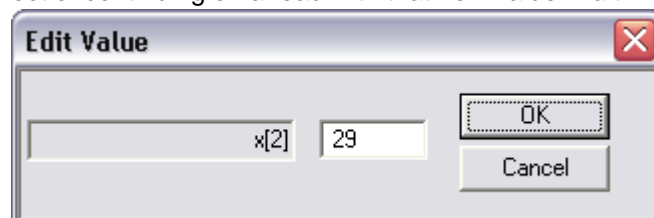

On program Load or Reset all *un-initialized value-variables are reset to value 0, and all un-initialized pointers to 0x0000.*

Lab Bench Pane

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

The Uno

This is a depiction of the Uno board and its onboard LEDs. When you load a new program into UnoArduSim, if it successfully parses it undergoes a "simulated download" to the Uno that mimics the way an actual Uno board behaves– you will see the serial RX and TX LED's 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's execution. You can avoid this display and associated loading lag by deselecting **Show DownLoad** form the **Options** menu.
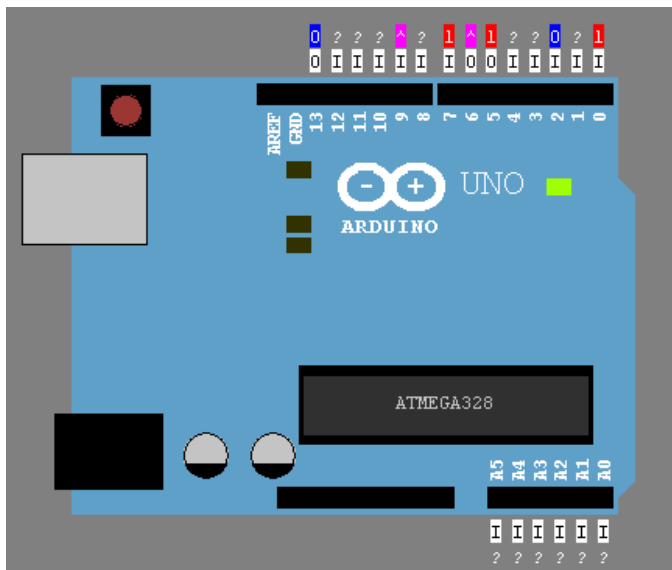
The window allows you to visualize the digital logic levels on all 20 Uno pins ('1' on red for HIGH, '0' on blue for LOW, and '?' on grey for an undefined indeterminate voltage), and programmed directions ('I' for INPUT, "O" for OUTPUT). For pins that are being pulsed using PWM via **analogWrite( )**, or by **Tone( )**, or by **Servo.write( ),** the colour changes to purple and the displayed symbol becomes '^'.

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.*

ASIDE: Digital pins 0-13 appear as simulator pins 0-13, and analog pins 0-5 appear as A0-A5 . To access an analog pin in your program, you can refer to the pin number by one of two equivalent sets of numbers: 14-19; or A0-A5 (A0-A5 are built-in-in const variables having values 14-19). And only when using **analogRead( )**, *a third option is made available – you can, for this one instruction, drop the 'A' prefix from the pin number and simply use 0-5.To access pins 14-19 in your program using* **digitalRead( )** *or* **digitalWrite( ),** *you can simply refer to that pin number, or you may instead use the A0-A5 aliases.*

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

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

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

cursor backward-to-edge (or use larrow key)

activate wave

activate BLUE cursor

sense rising edges

cursor forward-to-edge (or use rarrow key)

delete wave

activate RED cursor

sense falling edges

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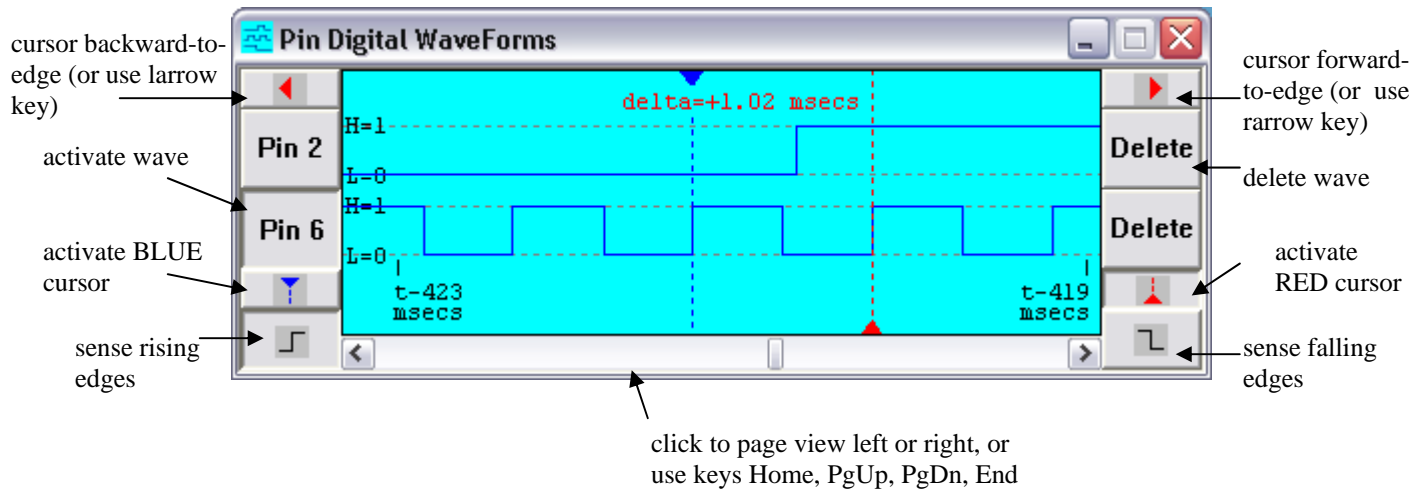
One of the displayed waveforms will be the ***active pin waveform***, indicated by its "Pin number" button being shown as depressed (for example Pin 6 is active in the above Pin Digital Waveforms screen capture). You can select a waveform by clicking its Pin number button, and then select the edge-polarity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

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

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

Doing instead a *right-click on any Uno pin* opens a **Pin Analog Waveform** window that displays the past one-second's worth of *analog-level activity* on that pin. Unlike the Pin Digital Waveforms window, you can only display one pin's worth of analog activity at any one time.
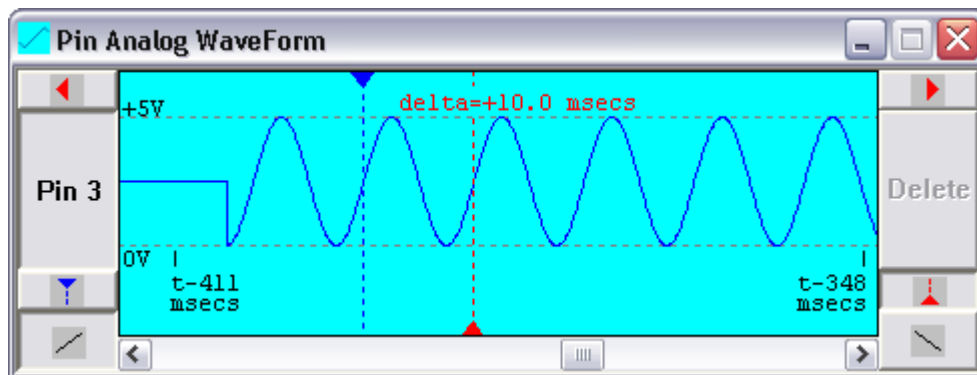
You can *jump-position* blue or red cursor lines to the next rising or falling "slope point" by using the forward/backward arrow buttons (◀ , ▶ or ◀ , ▶ , again depending on activated cursor, or *leftarrow* or *rightarrow*) in concert with the rising/falling slope selection buttons ╱ , ╲ (the "slope point" occurs where the analog voltage passes through the ATmega pin's high-digital-logic-level threshold). Alternatively, you can again click-to-jump, or drag these cursor lines similar to their behaviour in the Pin Digital Waveforms window

I/O Devices

A number of different devices surround the Uno 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 (of which you are allowed up to 8 in total) have "active" elements and reside along the top and bottom of the pane. The desired number of each type of available I/O device can be set using the **Config→ I/O Devices** menu selection.

Each IO 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( )**'s (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 OUTPUT pin driving against a high-drive (low-impedance) connected device (for example, driving against a FUNCGEN output, or a DCMOTOR encoder 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 conflicts 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 remainder of this section provides descriptions for each type of device.

Serial Monitor (SERIAL)

This I/O device allows for ATmega hardware-mediated Serial input and output (through the Uno's USB chip) on Uno 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.*

To send keyboard input to your program, type one or more characters in the upper (TX chars) edit window and then **hit Return**. 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 unformatted 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 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 (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

Software Serial (SFTSER)

This I/O device allows for library software-mediated, or ,alternatively, user "bit-banged", serial input and output on any pair of Uno 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 that library's functionality. As with the hardware-based SERIAL device, the baud rate for SFTSER 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.*
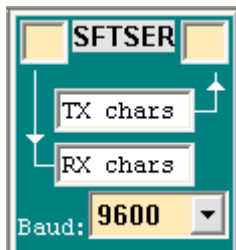
Also, as with the hardware based Serial, *a larger window for TX and RX setting/viewing can be opened by double-clicking on the SFTSER device*.

Note that unlike Serial's hardware implementation, there are no provided TX or RX buffers supported by internal ATmega interrupt operations, so that read( )'s and write( )'s are blocking (that is, your program will not proceed until they are completed).

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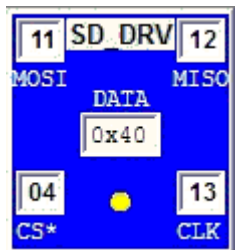
A *larger window displaying directories and files (and content) can be opened by double-clicking on the SD_DRV device*. All drive content is *loaded from* an **SD** sub-directory in the loaded program's directory (if it exists) at SdVolume `init()`, *and is mirrored to* that same **SD** sub-directory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

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 msecs.

Your program must have an **#include <Servo.h>** line before declaring your Servo instance(s) *if you choose to use the Servo library functionality* , e.g. **Servo.write( )**, **Servo.writeMicroseconds( )** Alternatively, SERVO also responds to **digitalWrite( )** bit-banged signals. Due to UnoArduSim's internal implementation, you are limited to 5 SERVO devices.

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 pins 3,9,10,11) or 980 Hz (on pins 5,6) PWM waveform of duty cycle between 0.0 and 1.0 (**analogWrite** values 0 to 255). Alternatively, DCMOTOR will *also respond* to your own digitalWrite( )-created bit-banged drive signals.

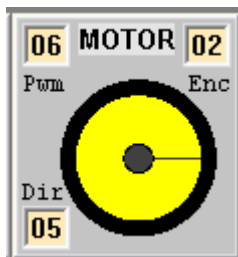The motor is accurately modelled both mechanically and electrically. Accounting for motor-driver transistor voltage drops and realistic no-load gear torque gives a full speed of approximately 2 revs per second, and stall torque of just over 5 kg-cm (occurring at a steady PWM duty cycle of 1.0), with a total motor-plus-load moment of inertia of 2.5 kg-cm. 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's spring-like lashback under static-friction. The consequence is that 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 with, of course, an appropriate (***and required)*** motor driver module in between the motor and the Uno!.

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).

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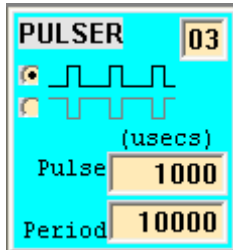
Analog Function Generator (FUNCGEN)

This I/O device emulates a simple analog waveform generator which produces a periodic signal that can be applied to any chosen Uno 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 usecs).
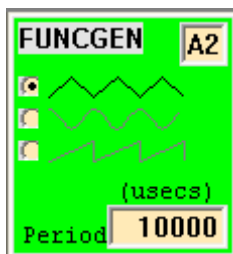
Analog Slider

A slider-controlled 0-5V potentiometer can be connected to any chosen Uno 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.
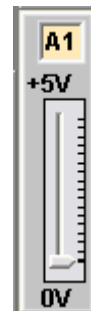
SPI Slave (SPISLV)

This I/O device emulates a simple shift-register device with an active-low SS* ("slave-select") pin controlling the drive on its Dout output pin (when SS* is high, Dout 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 data and clock signals to drive this device.
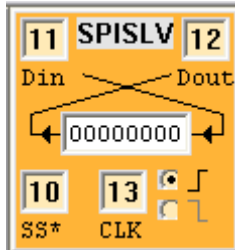
The device senses edge transitions on its CLK input which trigger shifting of its register – the polarity of sensed CLK edge may be chosen using a radio-button control. On every CLK edge (of the sensed polarity), the register captures its Din level into the least-significant-bit (LSB) position of the 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.

Two-Wire I^{2}C Slave (I2CSLV)

This I/O device emulates a *slave-mode-only* Two-Wire device. The device may be assigned an I^{2}C bus address using a two-hex-digit entry in its Addr edit box (it will only respond to I^{2}C bus transactions involving its assigned address). The device sends and receives data on its open-drain (pulldown-only) SDA pin, and responds to the bus clock signal on its open-drain (pulldown- only) SCL pin. Although the Uno 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).

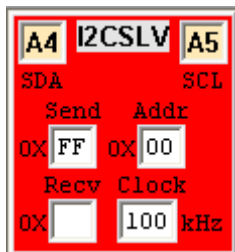Your program must have an **#include <Wire.h>** line if you wish to use the functionality of the TwoWire library to interact with this device. Alternatively, you may choose to create your own bit-banged data and clock signals to drive this slave device.

A single byte for transmission back to the Uno 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.

By double-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 the **Config->Preferences** menu dialog). 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_BUFFERLENGTH) both to be instead either 16 or 8, a user *could* significantly reduce the RAM memory overhead of the Uno in a targeted ***hardware implementation*** --. UnoArduSim therefore mirrors this real-world possibility through a Preferences menu option.

Push Button (PUSH)
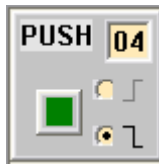
This I/O device emulates a normally-open momentary single-pole-single-throw (SPST) pushbutton with a 10 k-ohm pull-up (or pull-down) resistor. If a rising-edge transition selection is chosen for the device, the pushbutton contacts will be wired between the device pin and +5V, with a 10 k-Ohm pull-down to ground. If a falling-edge transition is chosen for the device, the pushbutton 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, you close the momentary pushbutton contact (it stays closed for as long as you hold down the mouse button).

Slide Switch Resistor (R=1K)
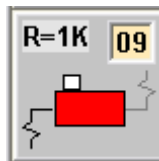
This device allows the user to connect to an Uno pin either a 1 k-Ohm pull-up resistor to +5V, or a 1 k-Ohm pull-down resistor to ground. This lets you simulate electrical loads added to a real hardware device. By left-clicking on the slide switch's *body* you can toggle your desired pull-up or pull-down 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.

Piezoelectric Speaker (PIEZO)

This device allows you to "listen" to signals on any chosen Uno pin, and can be a useful adjunct to LED's for debugging your program's 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.

Red LED (LED)

You can connect an LED between the chosen Uno 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 pin is HIGH, or instead, when it is LOW.

Menus

File menu commands:

Load INO or PDE Prog   Allows the user to choose a program file having the selected extension. The program is immediately parsed

View/Edit   Opens the loaded program for viewing/editing.

Save   Save the edited program contents back to the original program file.

Save As   Save the edited program contents under a different file name.

Next (#include) file   Advances the CodePane to display the next #include'd file

Previous   Returns the CodePane display to the previous file

Exit   Exits UnoArduSim.

Find menu commands:

Prompt   Click in either the Code Pane or the variables Pane to give it the active focus for this menu's commands.

Find Next Function/Var   Jump to the next Function in the Code Pane (if it has the active focus), or to the next variable in the Variables Pane (if instead it has the active focus).

Find Previous Function/Var   Jump to the previous Function in the Code Pane (if it has the active focus), or to the previous variable in the Variables Pane (if instead it has the active focus).

Set Search Text   Pull up a dialog box to edit your to-be-searched-for text..

Find Next Text   Jump to the next Text occurrence in the Code Pane (if it has the active focus), or to the next Text occurrence in the Variables Pane (if instead it has the active focus).

Find Previous Text   Jump to the previous Text occurence in the Code Pane (if it has the active focus), or to the previous Text occurrence in the Variables Pane (if instead it has the active focus).

Execute menu commands:

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

Options menu commands:

Step Over Structors/Operators	Fly right through constructors, destructors, and operator overload function during any stepping (i.e. it will not stop inside these functions).
Register-Allocation Modelling	Assign function locals to free ATmega registers instead of to the stack (generates somewhat-reduced RAM usage).
Error on Uninitialized	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).
Show Program Download	Show program download to the Uno board (with attendant delay).
Bigger Font	Use the next larger font size for the Code Pane, Variables Pane, and View/Edit window.

Configure menu commands:

I/O Devices   Opens a dialog to allow the user to choose the type(s), and numbers, of desired I/O devices. From this dialog 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

Preferences   Opens a dialog to allow the user to set preferences including auto-formatting of source program, allowing Expert syntax, enforcing of array bounds, Uno board version, and TWI buffer length (for I2C devices).

VarUpdates menu commands:

Allow Reduction   Allow reduced frequency of display updates in the Variables Pane to avoid flicker or reduce CPU load – then values shown are only updated periodically, ***but also whenever the program is halted***.

Minimal Updates   Only refresh the variables Pane display 4 times per second.

HighLight Updates   Highlight the last-changed variable value (will cause scrolling).

Windows menu commands:

Serial Monitor   Add Serial IO device (if none) and pull up a larger Serial monitor TX/RX text window.

Restore All   Restore all minimized child windows.

Prompt   Left-Click or Right-Click an Uno Pin to create a Waveform window:

Pin Digital Waveforms   Restore a minimized Pin Digital Waveforms window.

Pin Analog Waveform   Restore a minimized Pin Analog Waveform window.

Help menu commands:

Quick Help File   Opens the UnoArduSim_QuickHelp PDF file.

Full Help File   Opens the UnoArduSim_FullHelp PDF file.

Bug Fixes   View significant bug fixes since the previous release..

Change/Improvements   View significant changes and improvements since the previous release.

About   Displays version, copyright.

Modelling

Intro

The Uno and attached I/O devices are all modelled electrically, and you will be able to get a good idea at home of how your programs will behave with the actual hardware.
.

Timing

UnoArduSim executes rapidly enough on a PC or tablet that it can (in the majority of cases) model program actions in real-time. This is clearly useful for evaluating your programs.

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 time execution rate must fall behind real-time* is when the user has configured operation with multiple very-high IO device frequencies (and/or baud rate) which would generate an excessive number of pin-level-change events and associated processing overload. This overload is avoided by skipping some timer intervals to compensate, and this would slow down program progression to below real-time.

Programs with large arrays being displayed, or having very tight loops (with little inserted delay) that causing a high function call frequency can also generate a high Variables Pane display update load – this can be circumvented by allowing update reduction s in the **VarUpdates** menu, or by selecting **Minimal Updates** there when necessary.

Accurately modelling the sub-millisecond execution time for each program instruction or operation is *not done* – only very rough estimates for most have been adopted for simulation purposes. However, the timing of `delay( )`, and `delayMicroseconds( )` functions, and functions `millis( )` and `micros( )` are all perfectly accurate, and as long as you use at least one of the delay functions in a loop somewhere in your program, or you use a function that naturally ties itself to real-time operation (like `print( )` which is tied to the chosen baud rate), then your program's simulated performance will be very close to real-time (again, barring 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 w*hen running*, it may be desirable to be able to slow things down. A time-slowdown factor of 10 can be set by the user under the Options menu.

I/O Devices

These virtual devices receive real-time signaling of changes that occur on their input pins, and produce corresponding outputs on their output pins which can then be sensed by the Uno -- 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. UnoArduSim starts and stops an associated sound buffer as execution is started/halted to keep the sound buffer synchronized to program execution. Multiple PIEZO's are accommodated, and the DirectSound^{TM} API is used to look after all sound mixing.

Limitations and Unsupported Elements

Included Files

Bracketed-file `#include`'s `<Servo.h>`, `<Wire.h>`, `<SoftwareSerial.h>`, `<SPI.h>` and `<EEPROM.h>` are supported but 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-file `#include`'s (like `"supp.ino"` , `"myutil.cpp"`, or `"mylib.h"`) are supported but all such files must *reside in the same directory as the parent program file* that `#include'`s them (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 which are `#include`'d (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 parent program's directory).

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 option 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 modelling). 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 `'char'`, `'int'` and `'float'` variables/arrays and `String`'s are supported, but the only supported flash-memory `String` function is `strcpy_P( )`. It is recommend that you use the predefined types (`prog_char`, `prog_int16` etc.), but if you do directly use the `PROGMEM` variable modifier keyword, it must appear in *front* of the variable name.

Strings

The native `String` library is almost completely supported with a few very (and minor) exceptions. Direct String-character access through array notation bracketing `[ ]`-operator overloading is not supported – use the `charAt( )` member function instead, or alternatively, first get the `String`'s

buffer address into a "`const char * strptr`" by assigning to it the returned value of the `c_str( )`.member function – you could then use `strptr[ ]` for array-style (but read-only) access to the `string`'s characters.

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

Arudino Libraries

Only `Servo.h`, `SoftwareSerial.h`, `SPI.h`, `Wire.h`, and `EEPROM.h` currently have built-in support

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.

Classes and Structs

Although polymorphism, and inheritance (to any depth), is supported, classes and structs 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 non-static '`const`' 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' ("cast-like") functions are not supported.

Scope

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

All `typedef`'s, and all named `struct`'s and `class`'es (i.e. that may be used for future declarations), must be declared at *global* scope (*local* declarations of such items inside a function are not supported).

Const, Volatile, Static

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. 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 (classes/structs), for non-member functions, and for all function arguments.'

Compiler Directives

Regular **#define**'s are supported, but *not macro #defines*. The **#pragma** directive and conditional inclusion directives (**#ifdef**, **#ifndef**, **#if**, **#endif**, **#else** and **#elif**) are not supported. The **#line**, **#error** and predefined macros (like **_LINE_**, **_FILE_**, **_DATE_**, **_TIME_**) are 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's are not yet 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* modelled 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 **Modelling** above for more detail on limitations.

Release Notes

Bug Fixes

V1.2 – Jun 2015

1) The very simplest of user functions that made calls to either `digitalRead()` or to `analogRead()`could have corrupted their (very first) of any declared local variables due to those functions incorrectly returning a 4-byte value into a common scratchpad stack slot (correctly) allocated as only 2 bytes . Any arithmetic expression at all, or any call to a 4-byte-return function made inside a user function is sufficient to cause (at least) a 4-byte scratchpad allocation, and so avoids this issue. This unfortunate bug has been around since the original release V1.0 but is now fixed.

2) Void functions with an early explicit `return`, and non-void functions with more than one `return` statement, would see execution fall-through at the ***closing brace*** (if it was reached).

3) Return statements in un-braced `if( )` contexts led to a faulty return-to-caller target.

4) **Pulser** and **FuncGen** pulsewidths/periods of 0 could cause a crash (0 is now disallowed).

5) '`else`' continuations after an `if()` did not work following unbraced '`break`', '`continue`', or '`return`'s.

6) When multiple user `enum`'s were declared, only constants defined in the very first `enum` did not generate faulty "enum mismatch" Parse errors (the bug got introduced in V1.1).

7) A null identifier for the very last parameter of a function prototype caused a Parse error.

8) **Run-To** breakpoints set on complex lines were not always handled properly (and so could be missed).

9) HardwareSerial and SoftwareSerial used a private implementation TXPending buffer that did not get cleaned out on Reset's (so leftover characters from last time could appear).

10) Parse failed to check for illegal bit-flipping of float's, and pointer arithmetic attempted with illegal operators.

V1.1 – Mar 2015

1) Array indices that were 'byte' or 'char' sized variables caused incorrect array offsets (if an adjacent variable contained a non-0 high-byte).

2) Logical testing of pointers tested the pointed-to value for non-zero rather than the pointer value itself.

3) Return statements embedded inside for() or while() loops were mishandled.

4) Aggregate-initialization lists for arrays of objects, or objects containing other objects/arrays,

or completely empty initialization lists, were not being handled properly.

5) Access of enum member values using an '**enumname.**' prefix was not supported.

6) Declaration-line initialization of a **char [ ]** array with a quoted string literal was not working.

7) An array being passed to a function without prior initialization was improperly flagged with a "used but not initialized" error.

8) Pointer expressions involving array names were mishandled.

9) Function parameters declared as 'const' were not accepted.

10) The Pin Analog Waveform window did not display PWM signals (**servo.write()** and **analogWrite()**).

11) Member functions accessed.through an object-pointer gave faulty member accesses.

12) Waveforms were not being updated when a **RunTo** breakpoint was reached.

13) Register allocation modelling could fail when a function parameter was hen used directly as an argument to another function call.

V1.0.2 – Aug 2014
Fixed ordering of A0-A5 pins on the perimeter of the Uno board.

V1.0.1 – June 2014
Fixed bug that truncated edit "paste"'s that were longer than three times the number of bytes in the original program.

V1.0 -- first release May 2014

Changes/Improvements

V1.2 – Jun 2015

1) The SD library is now fully implemented and a (small) 8Mbyte SD Disk I/O device has been added (and functionality tested against all Arduino sample SD programs).

2) Like Arduino, UnoArduSim will now automatically convert a function argument to its address when calling a function expecting a pointer to be passed.

3) Parse error messages are now more appropriate when there are missing semicolons, and after unrecognized declarations.

4) Stale Variables Pane line highlights now get removed on function call/return.

V1.1 – Mar 2015

1) The main window can now be maximized/resized to make the Code Pane and Variables Pane wider (for larger screens).

2) A Find menu (and toolbar icons) have been added to allow quicker navigation in the Code and Variables Panes (PgUp and PgDown, or text-search with up-arrow, down-arrow).

3) The View/Edit window now allows ctrl-PgUp and ctrl-PgDn navigation jumps (to next empty-line), and has augmented Find/Replace functionality.

4) A VarUpdates menu item has been created to allow the user to select a computation-saving approach under heavy Variables Pane update loads.

5) Uno pins and attached LED's now reflect any changes made to I/O devices even when time is frozen (that is, even when execution is halted).

6) Other user functions can now be called from inside a user interrupt function (in accordance with update to Arduino 1.06).

7) A larger font can now be chosen from the Options menu.

V1.0.1 – June 2014

Waveform windows now label analog pins as A0-A5 instead of 14-19.

V1.0 -- first release May 2014