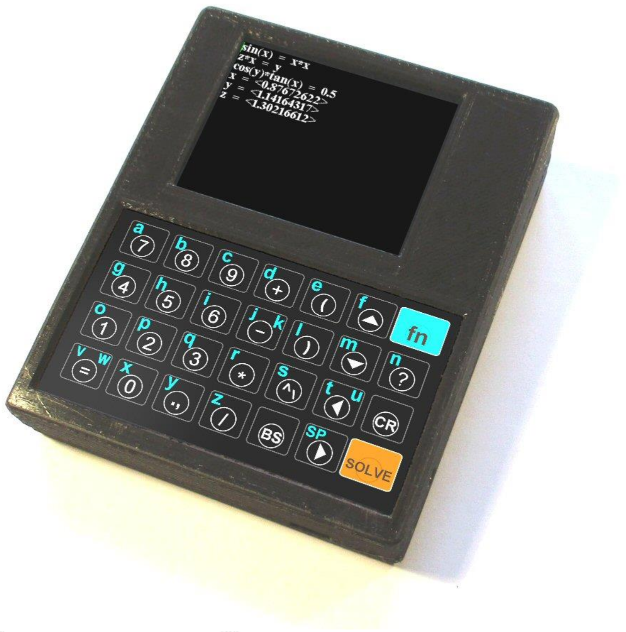
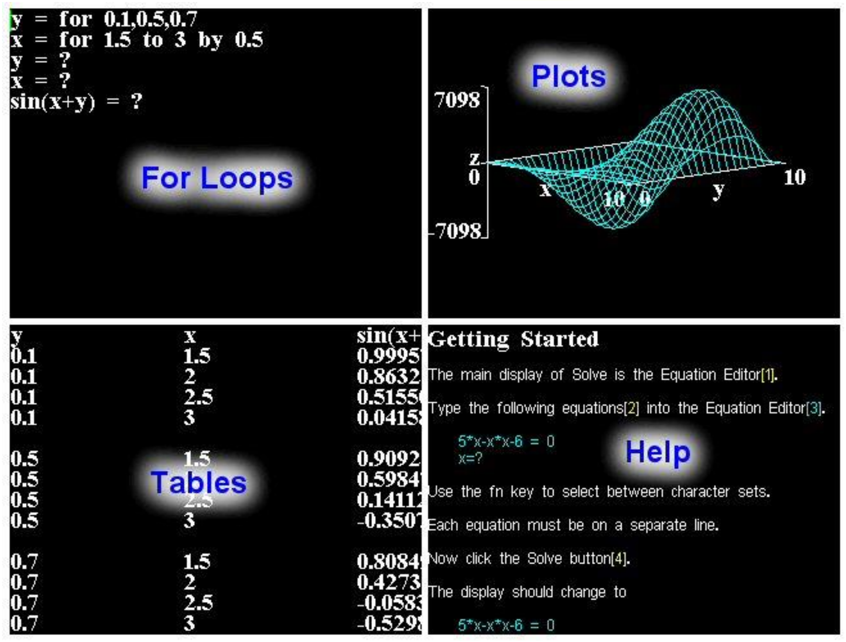
Introduction



```
sin(x) = x*x
z*x = y
cos(y)*tan(x) = 0.5
x = <0.87672622>
y = <1.14164317>
z = <1.30216612>
```



```
y = for 0.1,0.5,0.7
x = for 1.5 to 3 by 0.5
y = ?
x = ?
sin(x+y) = ?
```

```
y                  x                  sin(x+
0.1                1.5                0.9995
0.1                2                  0.8632
0.1                2.5                0.5155
0.1                3                  0.0415

0.5                1.5                0.9092
0.5                                   0.5984
0.5                2.5                0.1411
0.5                3                  -0.3507

0.7                1.5                0.8084
0.7                2                  0.4273
0.7                2.5                -0.0583
0.7                3                  -0.529
```

Getting Started

The main display of Solve is the Equation Editor[1].

Type the following equations[2] into the Equation Editor[3].

```
5*x-x*x-6 = 0
x=?
```

Use the fn key to select between character sets.

Each equation must be on a separate line.

Now click the Solve button[4].

The display should change to

```
5*x-x*x-6 = 0
```

The Solve pocket calculator solves equations numerically. You can build the calculator or you can use this Instructable as the basis of your own project that requires ESP32-WROOM-32 microprocessor, a screen and a home-made keyboard.

You can use Solve as an ordinary pocket calculator: type in an expression and it will show you the answer. Or you can type in several equations in any order and it will solve them. The equations can be non-linear and simultaneous If they contain "For Loops" then they will produce a table of answers or plot a graph.

The project uses

- a ESP32-WROOM-32 microprocessor module
- a 2.4" TFT screen with an ILI9341 controller
- 28 mini pushbuttons

Which cost me around £11.

You probably already have:

- an on/off switch

For power. I used

- a Lithium cell
- a USB Lithium charger module

Or you could use

- 3 AA or AAA cells

I built the keyboard on stripboard because I'm too impatient to wait for delivery of a PCB but I've included a PCB layout for those who dislike stripboard.

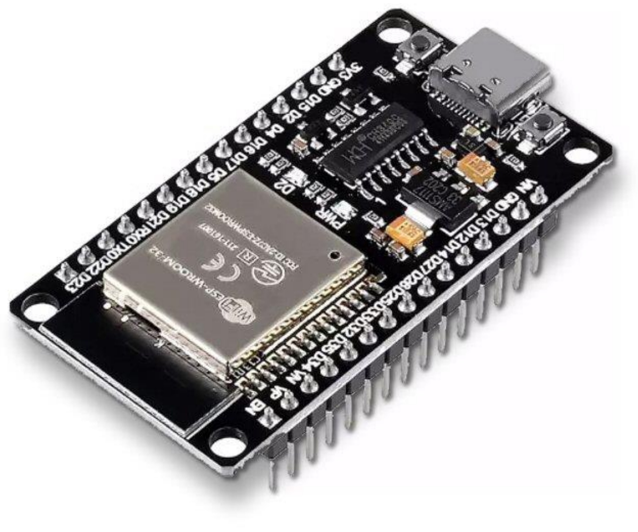
I 3D-printed a case but you could easily hand-build one using polystyrene sheet.

I suggest you test it in stages as described in the following sections:

- get the ESP32-WROOM-32 working
- connect and test the display
- build and test the keyboard
- breadboard and test the entire circuit
- program the solver code
- build and test a Lithium cell and charger
- OR connect and test AA cells
- build a case

I've explained here how I built it but no doubt you will make your own design decisions.

The Processor



The circuit uses an ESP32-WROOM-32. Search eBay (or your favourite supplier) for "ESP32 WROOM 32" and choose one like the photo above. I got a 30-pin module; other versions with more pins are available.

(I originally tried to use the ESP32 C3 "Super Mini" but I simply couldn't get it to work - two days of wasted effort. If you search the web you'll find lots of people having the same experience. The "Super Mini" implements USB-Serial in software and I suspect that the Windows drivers wouldn't work on my computer. The ESP32-WROOM-32 has a separate USB-Serial chip and it worked first time for me.)

Here's how I installed the ESP32 into the Arduino IDE:

- Open the Arduino IDE. I'm using version 1.8.16 (yes, I know there's a more recent version but I needed that version for a different project).
- Click on the File|Preferences menu item The Preferences dialog appears.
- Set Compiler Warnings to None - my code produces a lot of warnings which I find unhelpful.
- Enter https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json into the Additional Board Manager URLs field.
- Click OK.
- Click on the Tools|Board|BoardsManager menu item. The Boards Manager dialog appears.
- Select "ESP32" in the Type box. The Arduino ESP32 Boards listbox item appears.
- Select the "ESP32" item in the listbox (ignore the "Arduino ESP32" item).
- Select 3.0.4 from the Install drop-down box.
- Click Install.
- The Arduino IDE and Expressif may install USB device drivers. If your anti-virus software requests permission, say "Install".
- In the Tools|Board menu, select "ESP32 Arduino" then select "ESP32-WROOM-DA Module".
- In the Tools|Port menu, select the port that the new USB device driver has created. For me, it was Port 4.

My ESP32-WROOM-32 module came with a CH340 USB-to-serial convertor chip. That's the same one that a Nano uses. So if you can already program a Nano then you already have the CH340 driver installed and the ESP32-WROOM will work straight away. I think the ESP32-WROOM can come with a CP210x USB-to-serial convertor. In that case, you may need to install an additional driver which I believe you can get from here. (I haven't tested that.)

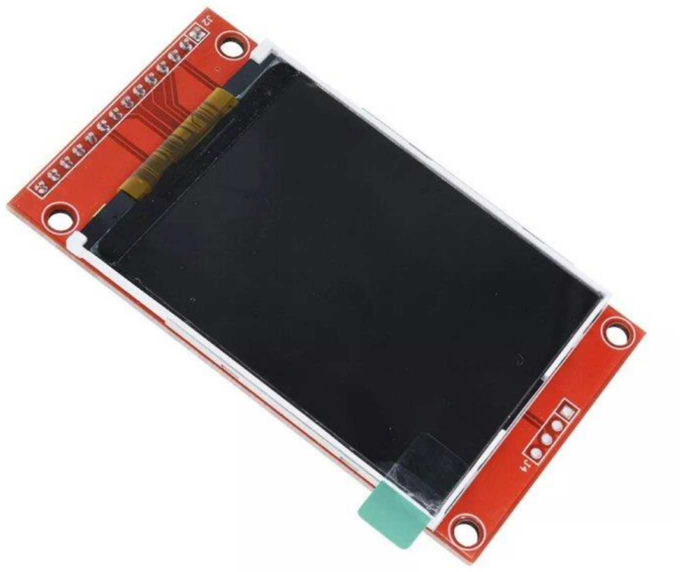
https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers?tab=downloads

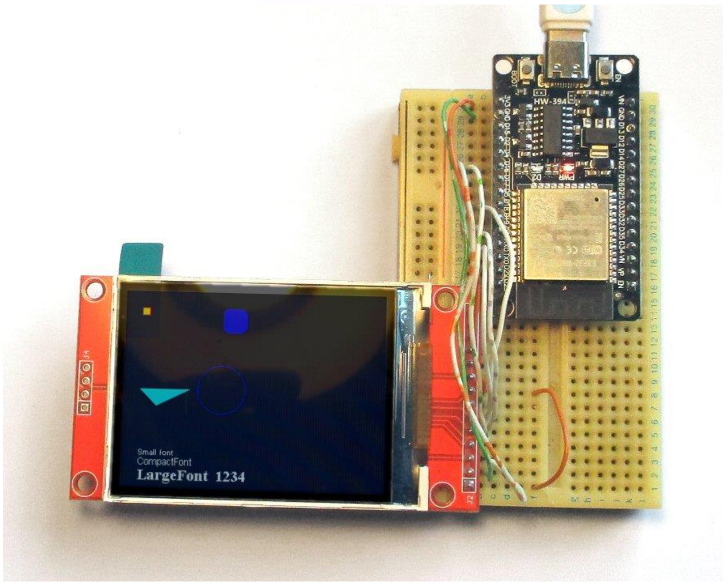
Write and upload a simple test program like this to make sure it all works:

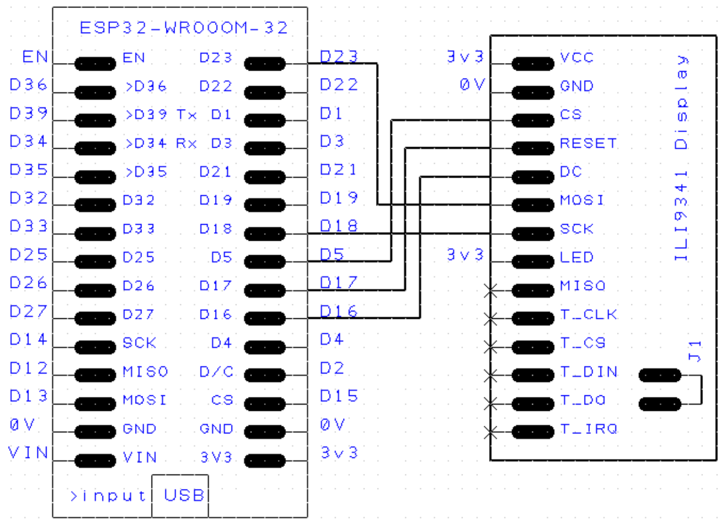
```
void setup() {
    Serial.begin(115200);
}

void loop() {
    Serial.println("Hello world");
    delay(1000);
}
```

The Display







The circuit uses a 2.4" TFT screen with an ILI9341 driver chip and "without touch". Search eBay (or your favourite supplier) for "2.4 TFT ILI9341" and choose one like the photo above. Make sure it has an SPI interface (it will have pins labelled MISO, MOSI, SCK, etc.) and it is 240x320 pixels.

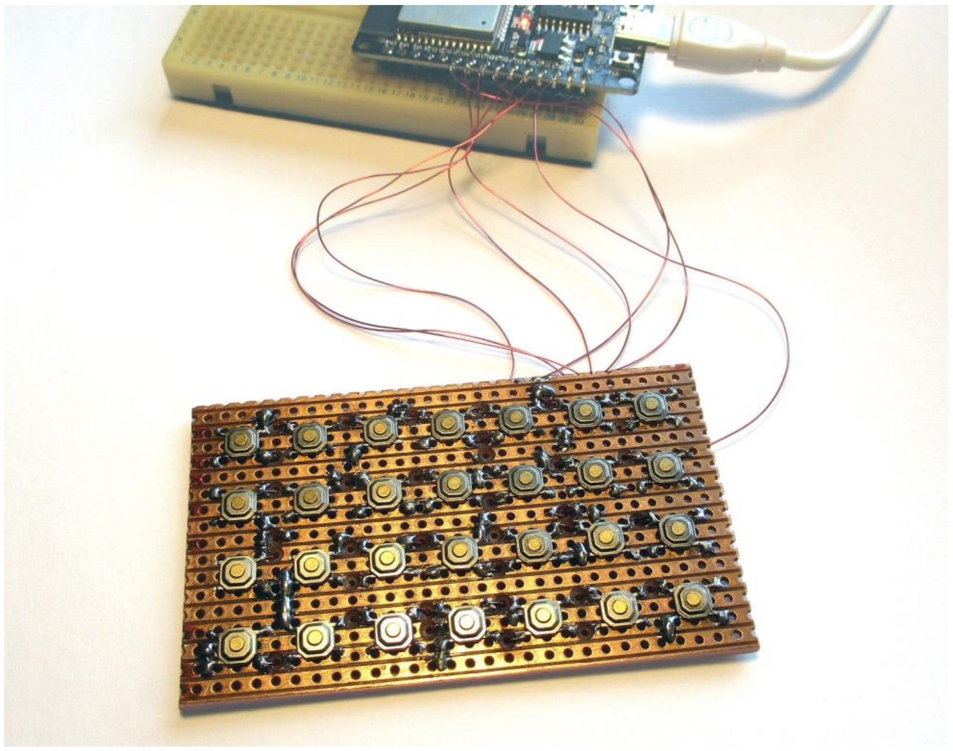
The ILI9341 display driver runs at 3.3V but the TFT display module expects to be powered by 5V. It includes a 5V-to-3.3V regulator. You can power the display by putting 3.3V into the "5V" pin - I've found that to be reliable. But it's better to bypass the regulator. The board should have a jumper labelled J1. Bridge it with a blob of solder.
https://randomnerdtutorials.com/esp32-tft-touchscreen-display-2-8-ili9341-arduino/

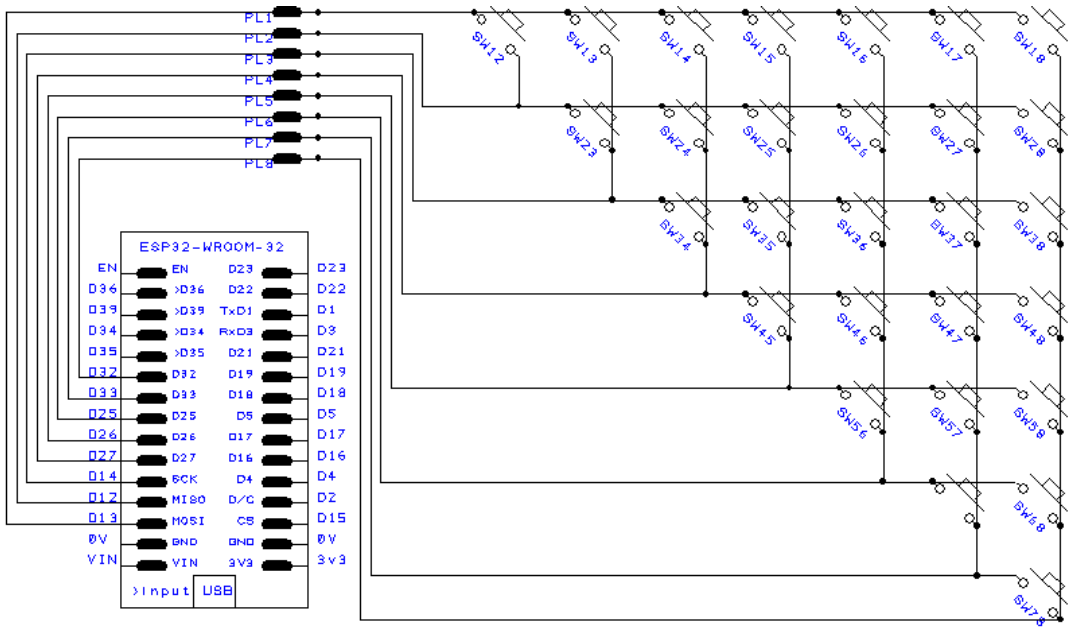
Test the display with circuit shown above. You can build it on a solderless-breadboard. Copy the following files into a directory called "TestTFT"

- TestTFT.ino
- SimpleILI9341.h
- SimpleILI9341cpp

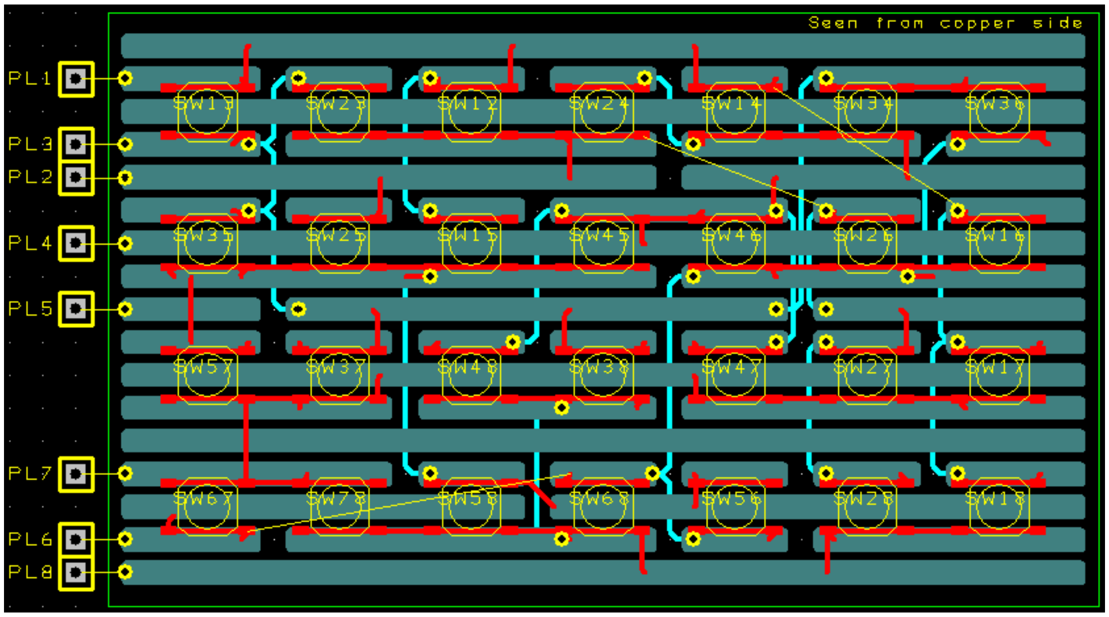
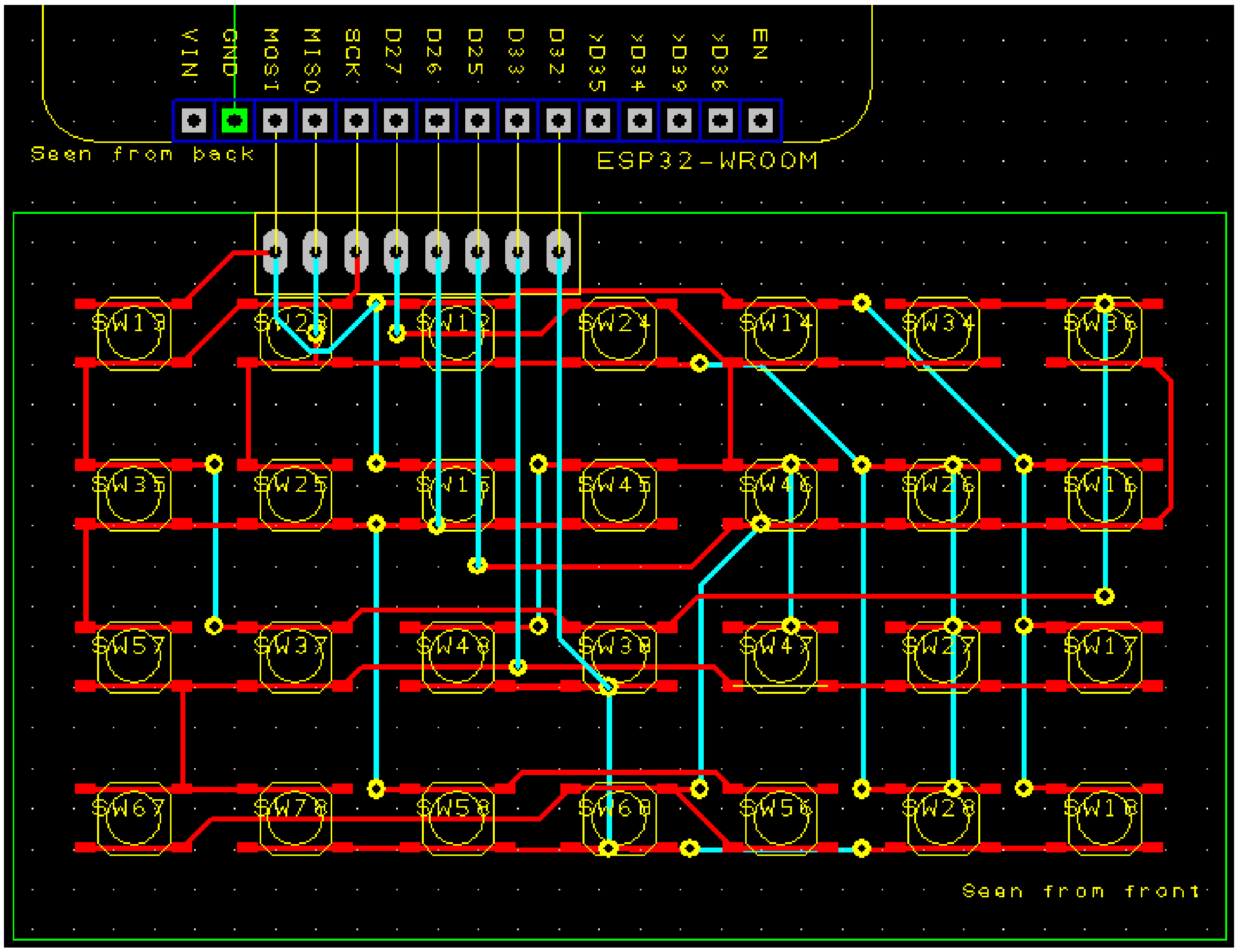
Compile and upload the sketch.; it shows some text, a circle, rectangle, triangle, etc. (In the File|Preferences dialog, set Compiler Warnings to None - my code produces a lot of warnings which I find unhelpful.)

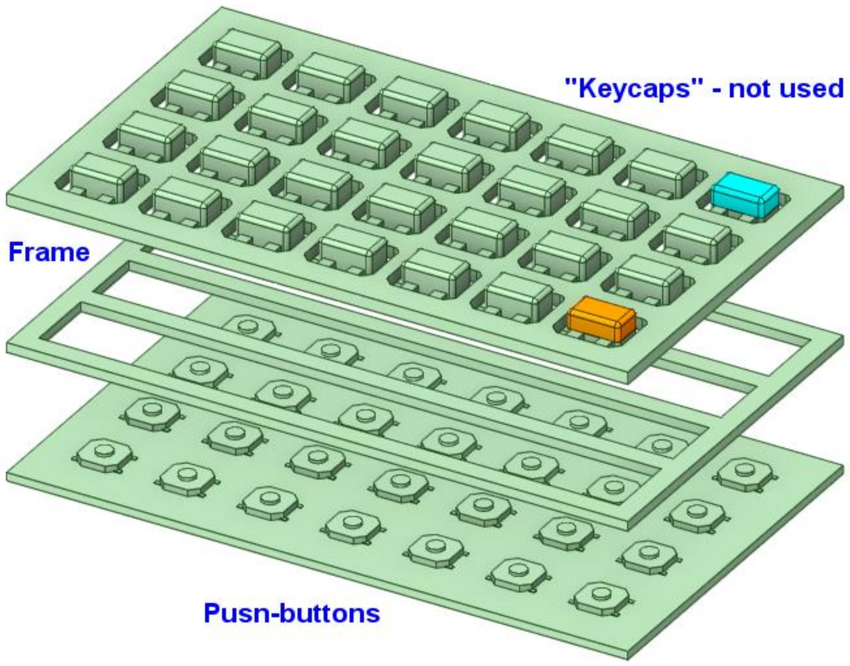
The Keyboard

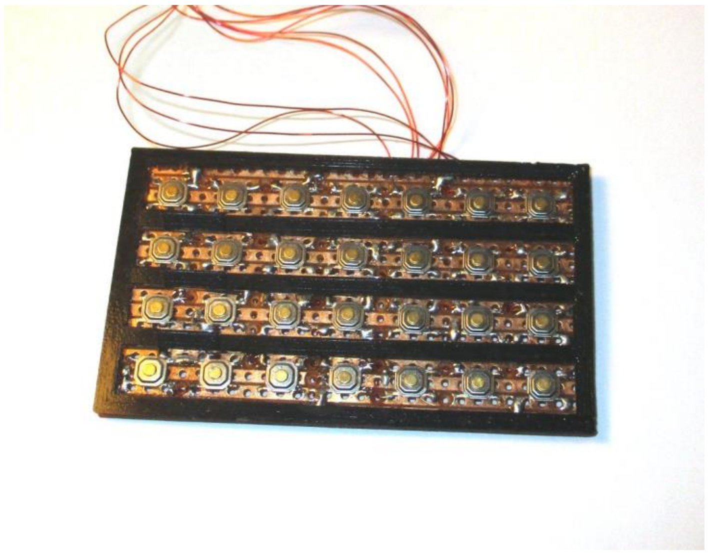


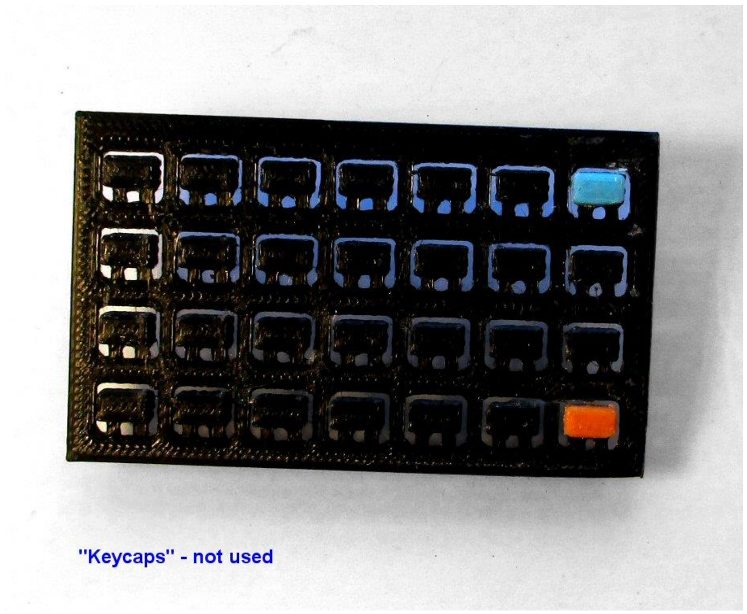


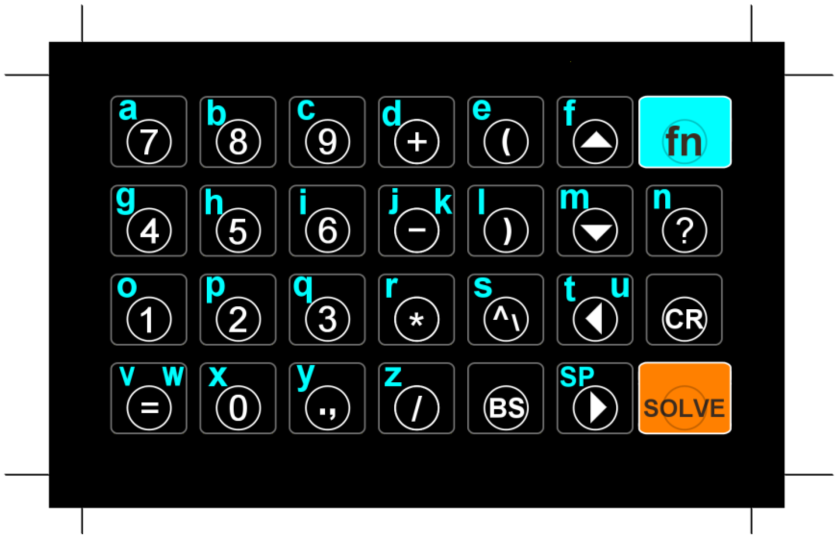
PL1
PL2
PL3
PL4
PL5
PL6
PL7
PL8
SW12
SW13
SW14
SW15
SW16
SW17
SW18
SW23
SW24
SW25
SW26
SW27
SW28
SW34
SW35
SW36
SW37
SW38
SW45
SW46
SW47
SW48
SW56
SW57
SW58
SW68
SW78
ESP32-WROOM-32
EN
D36
D39
D34
D35
D32
D33
D25
D26
D27
D14
D12
D13
ØV
VIN
>D36
>D39
>D34
>D35
SCK
MISO
MOSI
GND
D23
D22
TxD1
RxD3
D21
D19
D18
D5
D17
D16
D4
D/C
CS
3V3
D1
D3
D2
D15
3v3
>Input
USB


Seen from copper side
PL1
PL3
PL2
PL4
PL5
PL7
PL6
PL8
SW13 SW23 SW12 SW24 SW14 SW34 SW36
SW35 SW25 SW15 SW45 SW46 SW26 SW16
SW57 SW37 SW48 SW38 SW47 SW27 SW17
SW67 SW78 SW58 SW68 SW56 SW28 SW18

VIN
GND
MOSI
MISO
SCK
D27
D26
D25
D33
D32
>D35
>D34
>D39
>D36
EN
Seen from back
ESP32-WROOM
SW13 SW23 SW12 SW24 SW14 SW34 SW36
SW35 SW25 SW15 SW45 SW46 SW26 SW16
SW57 SW37 SW48 SW38 SW47 SW27 SW17
SW67 SW78 SW58 SW68 SW56 SW28 SW18
Seen from front


"Keycaps" - not used
Frame
Pusn-buttons




"Keycaps" - not used



It would be nice to be able to make your own calculator-style keyboards. I couldn't find any good examples on the web. It should be roughly the size of a calculator and look good. I collect old calculators and I particularly like the look of the TI-58 and HP-67 from the 1970s. And I want tactile-feedback.

The "keys" are sold on eBay as "5x5x1.5 Tactile Push Button" and I got 100 for £2 plus postage.They have a little button protruding from the top which I found works well. I also tried the"Tactile membrane switch" from eBay but I didn't like them as much - the "action" wasn't as positive and they couldn't be operated through a cover.

The ESP32-WROOM-32 module has few I/O pins so I use just 8 of them to scan the keyboard. The keys are charlieplexed without diodes which means there are 28 keys:

- 8*(8-1)/2 = 28

(If there were diodes, 8 pins could scan 56 keys but the circuit would be much more complicated.)

The ESP32-WROOM-32 enables pullups on the 8 pins then sets each pin low in turn and checks if any of the other pins have been pulled low. Imagine that each key has a "tag" byte. A bit in the byte is '1' if that pin was set low and another bit in the byte is '1' if that pin was pulled low. For instance "button 53" means that pin 3 was set low and pin 5 was pulled low "button 53" has tag 0x14.

The keys are arranged on the keyboard so as to minimise the total length of their interconnections. There is no "nice" ordering so the result looks random.

I soldered it all together on stripboard using self-fluxing enamelled copper wire for the interconnections. The result is not as pretty as a PCB but was quicker to build. (I've included PCB gerbers below if you'd prefer.) The keys are on a 0.4" grid. The PCB is 3"x1.8".

The cover for the keyboard was printed at a local photo printing shop. I find a "photo" is the best quality graphic printing I can produce quickly. A 7x5" print can contain several copies at slightly different scales - you can't be 100% sure how the printing machine will scale it. You can use the photo as it is or cover it with clear self-adhesive film.

I 3D-printed a frame which fits between the keys to make a surface to glue the photo onto. You can use UHU glue or PVA   glue ("white glue"). If you don't have a 3D printer, you could cut out thick cardboard or maybe even matchsticks to make the frame.

I tried two ways of printing the keyboard: with and without keys.

For the "with keys" keyboard, I printed a 3D matrix of keys. The matrix is glued onto the frame and the backs of the keys rest on the buttons protruding from the switches. I want the "Solve" and "fn" keys to be different colours so they were printed separately and glued on. It works well and has a nice feel. The captions for the keys are printed as a "photo". The problem is that it's hard to cut holes for the keys. I used a pin to make holes to mark the corners. I then cut the photo from the back with a scalpel. I couldn't get it to look nice. Maybe you'll do better. So I tried a different approach.

For the "without keys" keyboard, I simply glued a "photo" of the keys onto the frame. You can feel the switches under the card as you push them. It works well and is very much easier to make.

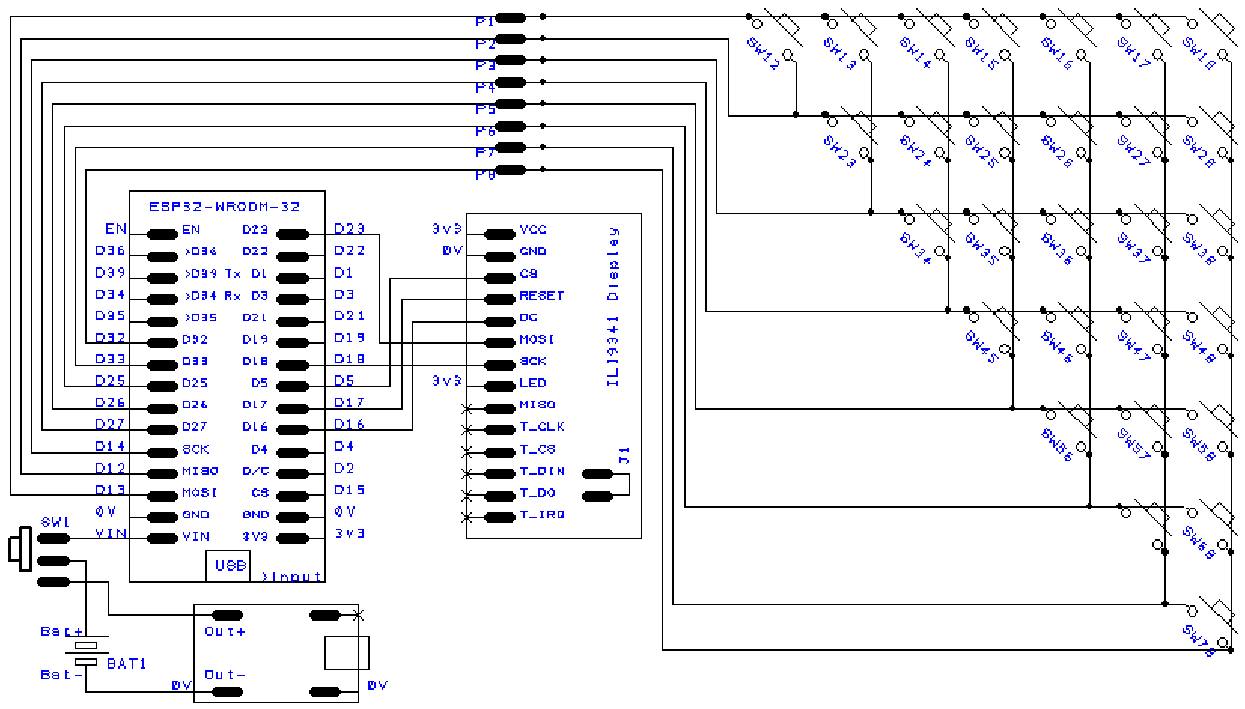
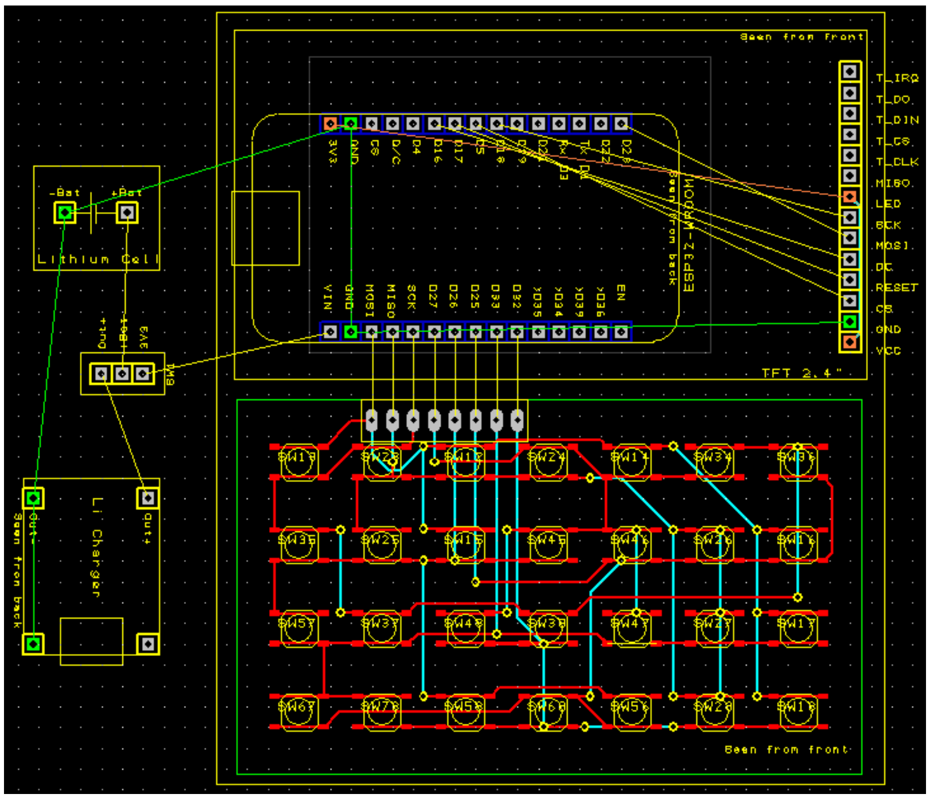
I think the resulting keyboard looks quite nice.

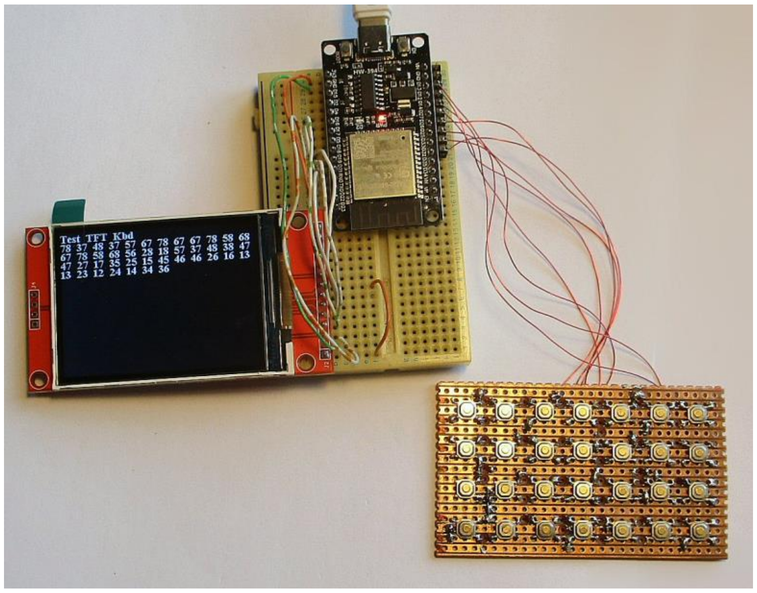
You could use different pins of the ESP32-WROOM-32 to scan the keyboard but note that D32, D34, D35,   D36, D39 are input only and D1, D3 are reserved for serial comms to the USB chip. Some pins have weak internal pullups; it's not clear from the documentation which ones. All I can say is that this circuit seems to work.

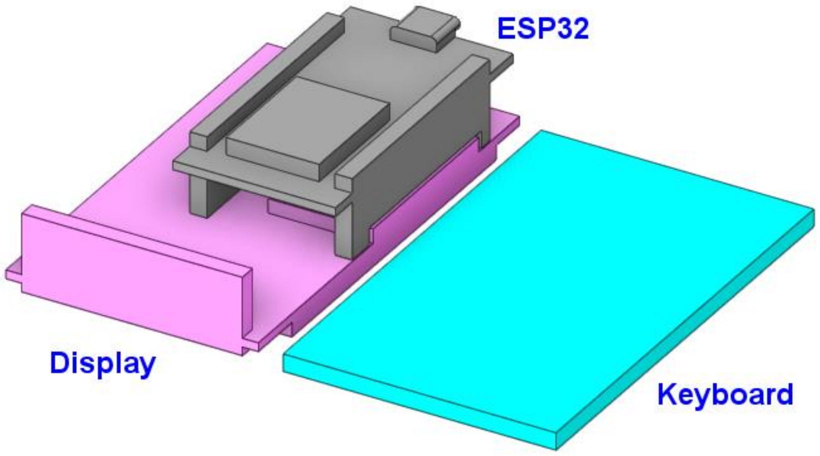
You can test the keyboard by downloading and running the TestKbd.ino sketch; when each key is pressed-and-released, it sends the tag value to the PC. If you have connected the TFT display, the TestTFT_KBD.ino sketch also prints the tag value on the TFT screen.

(A few other examples of mini keyboards can be found. LibreCalc no longer seems to exist and had no key captions. OpenRPNCalc uses stickers which I think will soon peel off.)
https://github.com/apoluekt/OpenRPNCalc

The Circuit


ESP32-WROOM-32
EN
D36
D39
D34
D35
D32
D33
D25
D26
D27
D14
D12
D13
0V
VIN
D23
D22
D1
D3
D21
D19
D18
D5
D17
D16
D4
D2
D15
3v3
>D36
>D39 Tx D1
>D34 Rx D3
>D35
SCK
MISO
MOSI
CS
GND
D/C
USB
>Input
ILI9341 Display
VCC
RESET
DC
LED
T_CLK
T_CS
T_DIN
T_DO
T_IRQ
J1
P1
P2
P3
P4
P5
P6
P7
P8
SW1
Bat+
Bat-
BAT1
Out+
Out-
SW12
SW13
SW14
SW15
SW16
SW17
SW18
SW23
SW24
SW25
SW26
SW27
SW28
SW34
SW35
SW36
SW37
SW38
SW45
SW46
SW47
SW48
SW56
SW57
SW58
SW67
SW68
SW78

Seen from front
T_IRQ
T_DO
T_DIN
T_CS
T_CLK
MISO
LED
SCK
MOSI
DC
RESET
CS
GND
VCC
TFT 2.4"
ESP32-WROOM
Seen from back
-Bat
+Bat
Lithium Cell
Out+
+Bat
3v3
SW9
Li Charger
Out-
Seen from back



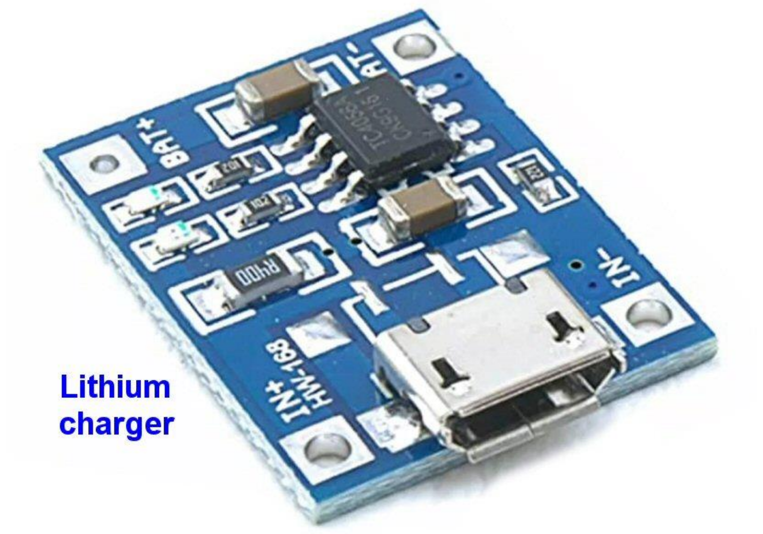


The circuit is very simple: an ESP32-WROOM-32, a TFT screen and the keypad. I didn't bother with a pcb (apart from the keypad): I just hooked up the connections with wires.

It would be nice if you could buy the ESP32-WROOM-32 and the display without pins but I couldn't. So I cut the pins short. Don't try to unsolder them - it's very easy to damage the PCB. I stuck the ESP32-WROOM-32 onto the back of the display with a hot-glue-gun. It gives a convenient solid unit. I positioned the ESP32-WROOM-32 so its USB connector is available in case I need to re-program it.

I used Easy-PC to design the circuit and PCB. The Easy-PC source files are available. It's such a simple circuit I expect you could reproduce it in under an hour using your own favourite design software.
github

Power Supply



The ESP32-WROOM can be powered from a 3.3V supply connected to the 3V3 pin or by, say, three AA cells connected to the "5V" pin.

Of course, during development, you just power it from the USB cable.

Copy what I did for this project and you'll be OK.

I used a a Lithium cell and a USB Lithium charger module. The charger module cost under £1 and I got a bag of Lithium cells for £0.50 each when a local electronics shop closed. I just soldered onto the terminals of the Lithium cell - yes, I know you're not meant to because the heat might damage it so be quick and careful.

I hot-glued the cell and charger module onto the back of my 3D-printed case and filed a notch for the USB cable. I fitted a transparent "window" so you can see the red/blue LEDs of the charger. Your arrangement might be different depending on what cell and module you use.

The calculator has a slide switch which connects the cell to either the ESP32-WROOM-32 or to the charger. (So you can't ue the calculator while it's charging.) With the switch "Off", the cell is connected to the charger; when the USB is disconnected, the charger draws around 2uA from the cell (i.e. far less than the self-discharge of the cell).

The cell is connected to the VIN pin of the ESP32-WROOM-32 (when the switch is On). The cell produces between 3.6V and 4.4V. The maximum supply of the ESP32-WROOM-32 is 3.6V and the minimum is 3.0V. The minimum dropout of the regulator at the current we're using seems to be around 0.5V. Those are the values I've measured. So the circuit I use works just fine even though it's "out of spec" according to the data sheets. (If this were a commercial project, I'd probably play safe and use a very low dropout regulator like the XC6203 into the 3V3 pin or a boost regulator into the VIN pin. There is no nice solution.)

If you're considering other projects, you might be interested to know that the "5V" or "VIN" pin of the ESP32-WROOM-32 connects to an AMS1117 or NCP1117 regulator chip (or maybe a different chip depending on your supplier). According to the data sheet, the absolute maximum input voltage is 12V and the minimum dropout is 1V. The maximum current is 1A. But don't believe those "12V" and "1A" maxima: the maximum power disspation of the regulator with no heatsink is maybe half a Watt so at 12V you can only draw 60mA. The TFT display takes around 120mA which will be well within the regulator's ability so long as the battery voltage is below 6V. My experience is that the TFT display works well at 3.3V but the touchscreen becomes unreliable below 4V. This project doesn't use the touchscreen. (A display without a touchscreen is a few pounds cheaper.)

Three AA cells give around 4.5V which will be ideal for the "VIN" pin.

The Software

I wrote the first version of this software in BASIC on a Sharp Pocket Computer around 40 years ago. Then I rewrote it in Turbo Pascal, then in Delphi-1, then in Delphi-4 and then in C for this project. So the code isn't as pretty as I'd like but it seems to be reliable.

Upload the code to the ESP32-WROOM-32. Set "Compiler Warnings" to None - my code produces a lot of warnings which I find unhelpful. Even with "Compiler Warnings" set to None, the gcc compiler still produces warnings like "". I wish it would just shut up. I'm not a fan of gcc.

Hold down the '?' key to get a Help page. Help pages can contain numbered links shown in yellow or cyan. Click the number key to display that page or load the example code that follows. There is a page of samples. Download one of them and press the "Solve" button.

The main display of Solve is the Equation Editor. Type the following equations into the Equation Editor.

- 5*x-x*x-6 = 0
- x=?

Use the fn key to select between character sets. Each equation must be on a separate line. Click the Solve button. The display should change to

- 5*x-x*x-6 = 0
- x=<2>

Solve has found a value for x which satisfies the first equation. The ? in the second equation has been replaced by the answer: <2>.

In the Equation display, holding down a key selects special commands:

- ? Display Help pages
- BS Clear the whole equations screen
- CR Duplicate current line
- SP Delete current line
- Left Load previous saved file
- Right Load next saved file

Some keys can type two characters, for instance 'j' and 'k'. Double-clicking toggles between the two.

If you include a "For Loop" then the solver will produce a table of answers. If you also include variables called "x" and "y" (and no "?") then the solver will draw a graph. If there is also a variable called "z" then the solver will draw a 3D graph.

You shouldn't need to touch the code but if you're interested, the sections are:

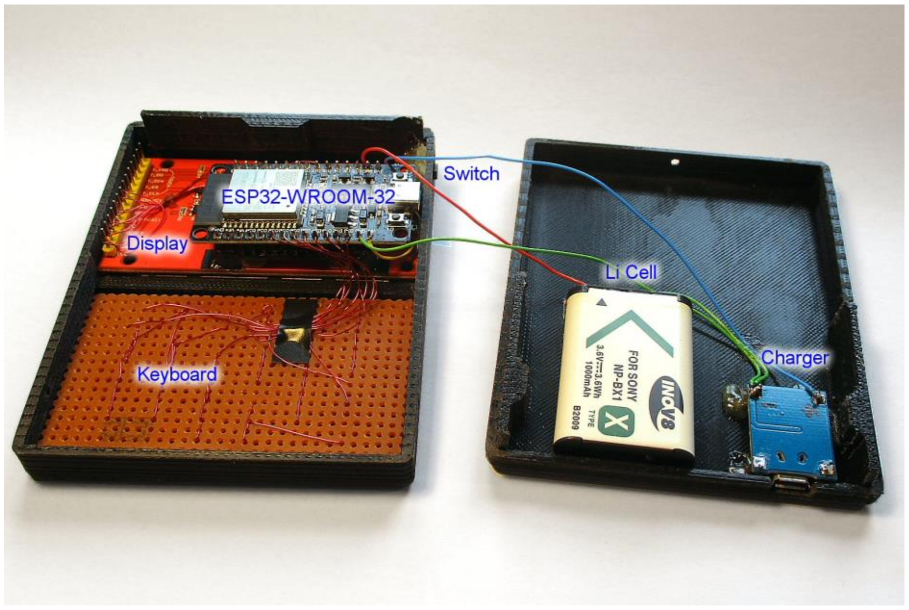
- Solver.ino: the "main program"; scans the keyboard
- calcdecl.cpp: useful declarations
- SimpleILI9341.cpp: the display driver
- VDU.cpp: the keyboard and screen that you type on
- parse.cpp: parses the text and produces a parse-tree
- solver.cpp: solves the parse-tree
- calc.cpp: calculates (constant) values from the parse-tree

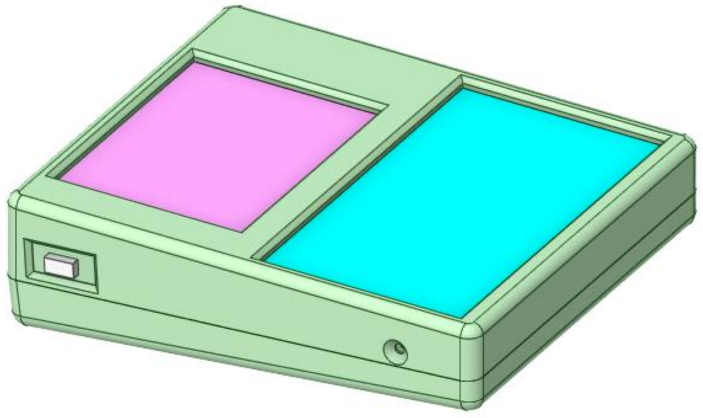
- gauss.cpp: gaussian elimination when solving simultaneous equations
- memory.cpp: a "heap" for the parse-tree, etc.
- Table.cpp: shows the table
- Plotter.cpp: shows the plots
- Help.cpp: shows the help pages
- Storage.cpp: saves previous pages of text in EEPROM

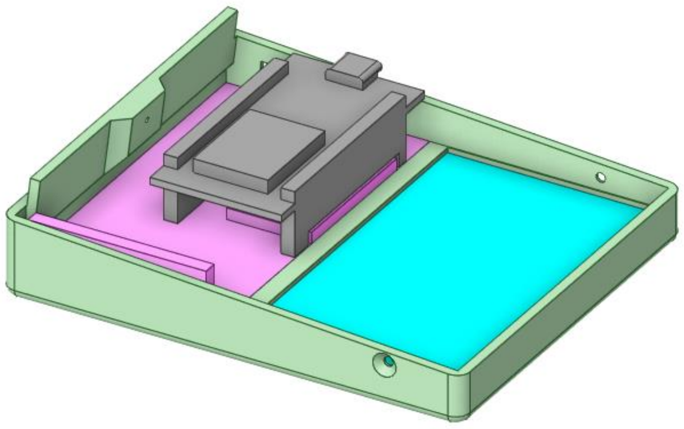
The parse-tree is kept on a home-made "heap" in memory.cpp. I don't use the actual heap provided by gcc because, of course, C doesn't have a garbage collector.

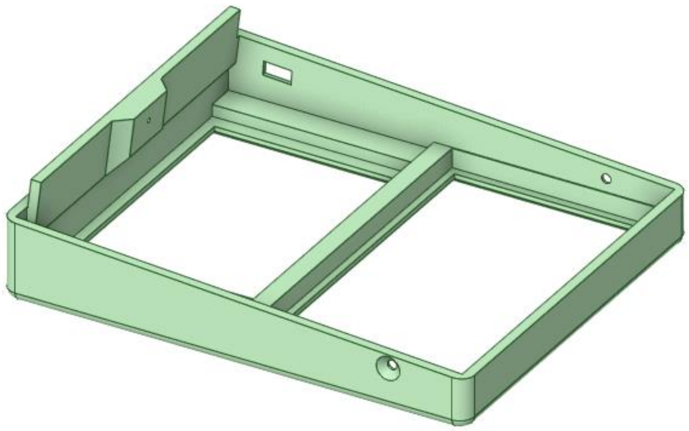
Each time you press "Solve", Storage.cpp saves previous page of text in EEPROM (if it has changed). Holding down the Left key loads the previous saved file so it acts like an "undo" key.

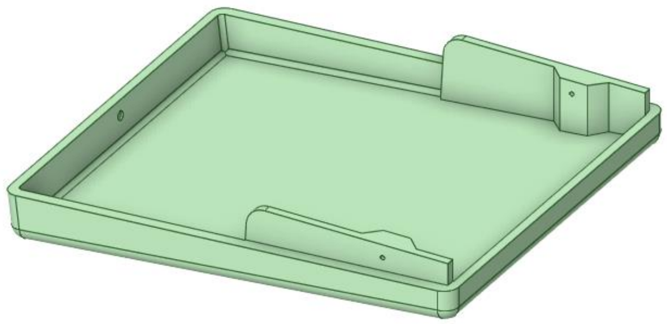
The Case











The case is 3D-printed. The STL can be downloaded (below). The source files for it are available here; they can be edited using the DesignSpark software available free from RS.

https://www.rs-online.com/designspark/mechanical-download-and-installation

If you don't have a 3D-printer, you could make a case out of polstyrene sheet - it would probably give a nicer result.

The boards are held in place with UHU glue and scrap plastic or PLA. You could use a hot-glue-gun but I find that UHU is strong enough to do the job but can be un-peeled if you need to change things.

The two halves of the case are held together with three self-tapping screws.

How to solve equations

You can ignore this section if you like. But if you're interested in the mathmatics of numerical equation solving then read on.

There are two ways to solve an equation like:

    x*x = 2*x + 15

The first is to solve it 'symbolically'. At school you were taught how to rearrange the equation to the form:

    (x - 5) * (x + 3) = 0

and deduce that x must be either +5 or -3.

The second way is to solve it 'numerically'. You can try various values of x in the equation until you hit upon one that makes both sides equal.

Symbolic solutions are exact and give you the general answer. For instance, you can rearrange:

    x*(x + y - 7) = 7*y

to show that x is either +7 or -y. And that will remain true for every value of y.

A numerical solution is usually only approximate, though you can choose the level of accuracy you want. In the equation above, it would give you only one answer for just the value of y in the example you solved.

Also you cannot be sure you have found all the possible solutions.

But numerical methods have two important advantages: they are easy for a computer and they can find answers where symbolic manipulation cannot.

For instance, the equation:

    exp(x) = 10*x

cannot be solved symbolically but you can solve it numercally. It has a solution with x approximately equal to 3.577.

Solve uses the following methods to solve equations containing one unknown:

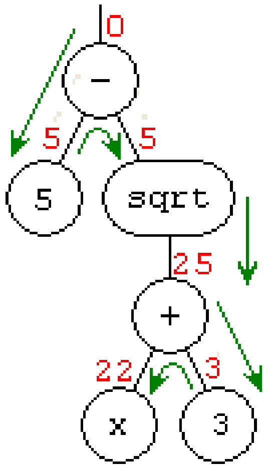
- Direct method
- Newton-Raphson method
- Regula-Falsi method
- Secant method
- Bisection method

To solve simultaneous equations, it uses:

- Newton-Raphson method

There are other methods which require the computer to rearrange the equations symbolically. Solve does not use these methods.

DirectMethod



Direct method

The Direct method solves equations with only one unknown, for instance

5 = sqrt(x+3)

Consider the equation as a tree (see above). The solver knows that the overall value must be zero so it tries to evaluate each side of each binary-node in turn.

If it succeeds with one side then it uses that answer in calculating the other side.

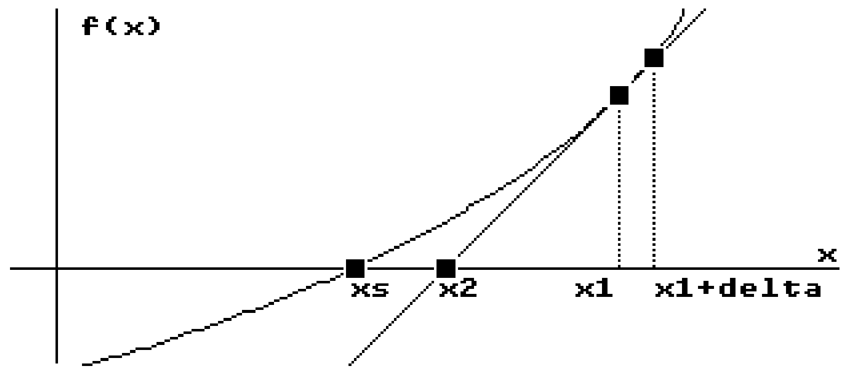
For instance, it knows that the sum node must evaluate to 25 and it finds that the right sub-branch evaluates to 3 so it knows that left sub-branch must evaluate to 22.

Eventually, it will find that one of the sub-branches is the unknown variable.

The Direct method is always tried first with every equation because it is very fast.

The Direct method ignores Initialisations. For instance, in cos(y)=0.5, it will always yield the value of y as +1.047 even though -1.047 is equally valid.

Newton-Raphson Method



Newton-Raphson method

The Newton-Raphson method finds solutions to equations of the form

f(x) = 0

We have some function 'f' of 'x' and want to find a value of 'x' for which f(x) is zero. If your equation has the form:

sin(x) = 1 - 0.5*x

then treat the '=' as a low priority minus sign and set f(x) equal to:

(sin(x)) - (1 - 0.5*x)

Imagine we have drawn a graph of f(x) against x and we are trying to find a value of x, call it xs, for which f(xs)=0. We choose an intial value for x, let's call it x1, and draw the tangent to the graph at the value of f(x1).

We mark a point where this tangent crosses the x-axis and call it x2. The Newton-Raphson method assumes x2 is closer to the point xs than x1 was. We can draw another tangent at f(x2) to give us x3 and so on (see above).

With luck, the successive estimates of x will get closer and closer to xs and f(x) will get closer to zero. Eventually, f(x) and the changes in x will be so small that that we can be satisfied that we have found a solution.

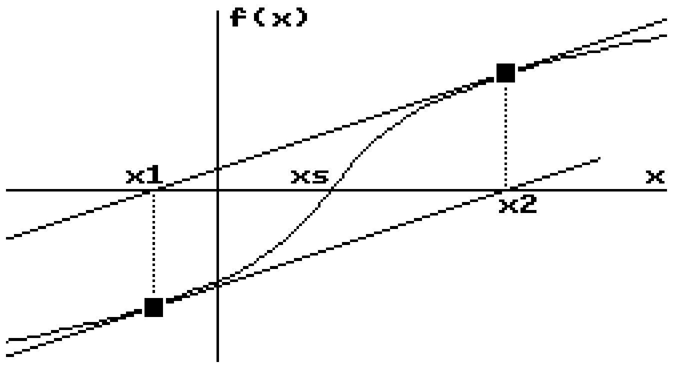
To draw the tangent we must differentiate f(x): we do so by calculating the value of f(x) at two values of x:

x = x1
x = x1+delta

where delta is a small value.

This process is repeated until the change in the estimates of x is less than Accuracy and f(x) is less than Accuracy or no solution can be found.

When an equation has more than one solution, the Newton-Raphson method usually finds a solution close to the initial guess for x.



Newton-Raphson method fails

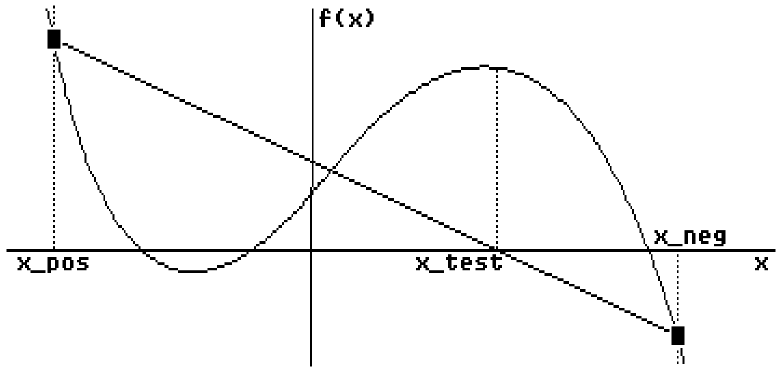
The Newton-Raphson method cannot solve all equations. In some cases, successive estimates of x get further from the solution; in the following case, the estimates cycle through the same values (see above).

It may be that by choosing a new initial guess for x, the Newton-Raphson method will get back on the right track or it may be that you should use an entirely different method.

If there is an arithmetic error while calculating f(x) or if the tangent is parallel to the x-axis then a new initial guess for x is generated.

The Newton-Raphson method is also used to solve simultaneous equations

Regula-Falsi Method



Regula-Falsi method

The Regula-Falsi method of solving an equation works with continuous functions.

A continuous function is one where you can draw the graph of f(x) versus x without lifting your pen from the paper. Sin(x) and sqrt(x) are continuous while tan(x) and 1/x are not.

We start with two guesses at the solution, x_pos and x_neg and we choose these so that f(x_pos) is greater than zero and f(x_neg) is less than zero.

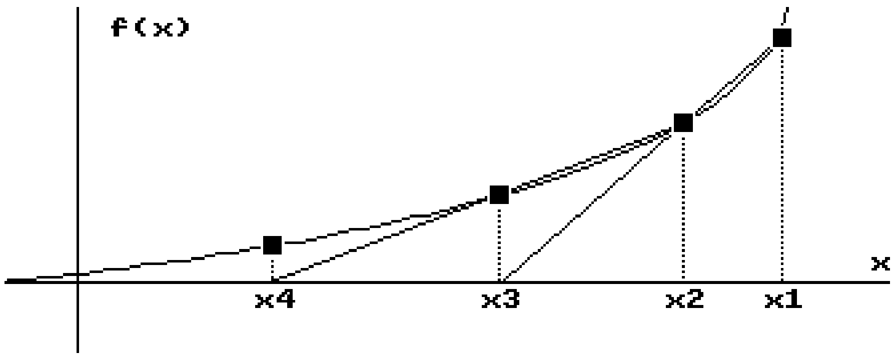
If f(x) is a continuous function then a solution f(xs)=0 must exist with xs somewhere between x_pos and x_neg (see above).

We can find a new test point by choosing x_test somewhere between x_pos and x_neg. In the Regula-Falsi method, x_test is where the straight line joining (x_pos,f(x_pos)) to (x_neg,f(x_neg)) crosses f(x) = 0.

If f(x_test) is greater than or equal to zero we set x_pos equal to x_test. If f(x_test) is less than or equal to zero we set x_neg equal to x_test. The difference between x_pos and x_neg must now be smaller than it was.

If we repeat these steps often enough, we will find a solution. We stop the loop when abs(x_pos-x_neg) is less than Accuracy and abs(f(x_test)) is less than Accuracy.

Secant Method



Secant method

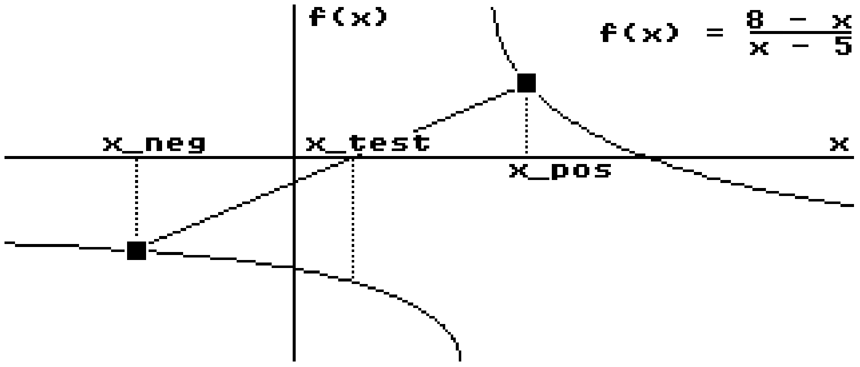
The Secant method is a slight variant on the Newton-Raphson method. We start with two guesses for x: call them x1 and x2. We draw a line through (x1,f(x1)) and (x2,f(x2)) and set x3 to where the line crosses the x-axis.

In the next iteration, we draw a line through (x2,f(x2)) and (x3,f(x3)) to give x4 and so on (see above).

The Secant method has the advantage that it calculates f(x) only once for each step.

Solve uses the Newton-Raphson method to find the first two estimates of x then uses the Secant method. It stops trying the Secant method after a iterations/2 attempts and switches to the Regula-Falsi and Bisection methods.

Bisection Method



Bisection method

The Bisection method is a slight variant of the Regula-Falsi method. In the Bisection method, x_test is half way between x_pos and x_neg.

For some equations, x_test converges on the solution faster with the Bisection method. For other equations, it converges faster with the Regula-Falsi method.

The Regula-Falsi method converges in a single step for linear equations but the Bisection method converges more quickly over a wider range of equations.

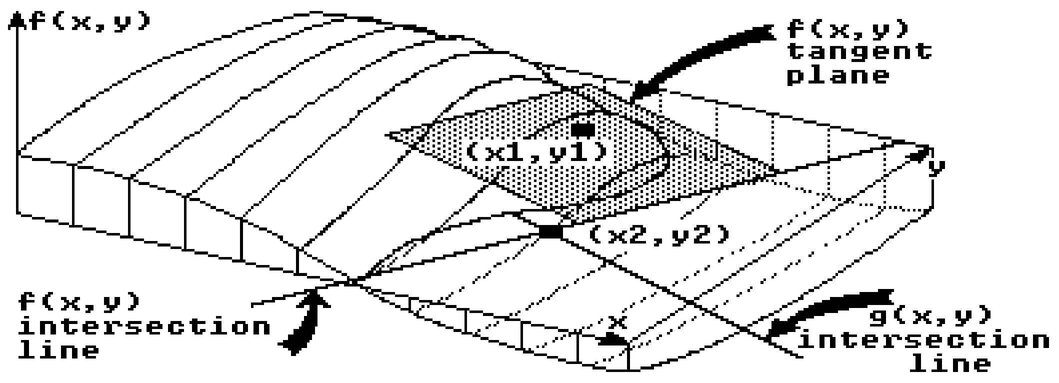
Solve uses the two methods alternately when calculating new values for x_test; that way, it gets the best of both.

Both the Bisection and Regula-Falsi methods need initial values of x_pos and x_neg such that f(x_pos) is positive and f(x_neg) is negative.

Solve uses the Newton-Raphson or Secant functions first. While these are running, they may find suitable values for x_pos and x_neg. If not, the algorithm described in the Initialisation section is used.

Neither method will work if x_pos and x_neg are on either side of a discontinuity (for instance see above).

Simultaneous Newton-Raphson Method



Simultaneous Newton-Raphson

So far we have considered only a single equation involving one unknown. In practice, we often need to solve systems of simultaneous equations. Of the methods used by Solve, only the Newton-Raphson method extends to simultaneous equations.

It is easiest to imagine the simultaneous version of the Newton-Raphson method with just two equations in two unknowns. The unknowns are x and y and the equations we want to solve are f(x,y)=0 and g(x,y)=0.

Imagine the graph of z=f(x,y): it has the form of a curved surface. We choose a test point (x_test, y_test) and find the tangent to the graph at that point. Once again, we can use numeric differentiation.

The tangent is a plane which crosses the x-y-plane along a straight line (see above).

Now we do the same for z=g(x,y). The tangent to the graph at (x_test,y_test) also crosses the x-y-plane along a straight line. Each tangent plane crosses the x-y-plane along a straight line: the "intersection lines" in the drawing above.

Where the two lines cross is where both planes are zero. That's where we choose our new estimate of (x,y).

As in the original version of the Newton-Raphson method, with luck, each new estimate of (x,y) is closer to the solution. We repeat the process until we are satisfied that (x,y) is sufficiently close to the solution.

In N equations with N unknowns, we will end up with N tangent planes. Each Tangent plane is defined by an equation of the form:

ax + by + cz + d = 0

We solve these N tangent plane equations using Gaussian elimination.

Initialisation

The following equations have two sets of solutions:

x*x = 2*x + 15
x= ?

They are

x=<-3.24264068>

and

x=<5.242640687>

The solution which the solver finds depends on its initial guess of the value of x.

The solver generates its first initial guess for x which is usually 1.0. If this guess does not lead to a solution then the solver generates a succession of initial guesses until either one leads to a solution or the solver gives up completely.

The initial guesses form the sequence:

• -1	negate
• -2	reciprocal and double
• 2	negate
• 0.5	reciprocal
• -0.5	negate
• -4	reciprocal and double
• 4	negate
• 0.25	reciprocal
• -0.25	negate
• -8	reciprocal and double
• 8	negate
• and so on	

These are alternately positive and negative and alternately get larger and smaller.

In the example above, the initial guess of 1.0 leads to the solution:

x=<-3.24264068>

and the solver stops.

To find the other solution, we must force the solver to use an initial guess somewhere near 5. We do so by adding the line:

init(x) = 5

The solver now finds the answer:

```
x=<5.242640687>
```

The solver makes two passes through the equations. On the first pass, it solves all the equations with an INIT in them. On the second pass, it solves the remaining equations and uses the initialisation values as the initial guesses for the variables .

If the solver cannot find a solution using the initialisation value for a variable then it reverts to the sequence of guesses described above.

When an equation has several solutions, a generator can be used with an INIT to generate a set of initialisation values.

Non-linear simultaneous equations can have several solutions, for instance, solving:

```
x*x+2*x*y=32
y*y+x=y+15
x=?
y=?
```

gives just one of the solutions

```
x=<2.923603392>
y=<4.010896838>
```

to find the other three, you must set various INITs, setting

```
init(x)=-14
init(y)=6
```

gives

```
x=<-14.1047439>
y=<5.918001837>
```

or setting

```
init(x)=-3
init(y)=-4
```

gives

```
x=<-3.02577347>
y=<-3.77501736>
```

or setting

```
init(x)=8
init(y)=-2
```

gives

```
x=<8.206913995>
y=<-2.15388130>
```

When there are two unknowns, you can use the plotter to find the approximate locations of solutions and use these for initialisation. But with three or more unknowns, it's more difficult.

Initialisaton has no effect when an equation is solved by the Direct method.

Other methods

Newton-Raphson, Secant and Regula-Falsi methods can be extended to work with higher order polynomials.

For instance, rather than use a straight line (a+bx=0) to determine the next estimate of x we could use a quadratic (a+bx+cx2=0) and have three test points.

Whether the extra calculations are compensated for by a faster convergence on the solution will depend on the equation being solved.

There are several numerical methods based on rearranging the equation. In general thsese are able to solve a wider range of equations than the purely arithmetic methods.

One of the simplest methods involves rearranging the equation f(x)=0 so that it has the form x=g(x). For instance, you could rearrange x*x-2=0 into x=(x+2)/(x+1). Choose an initial value for x1 and calculate x2=g(x1). Iterate until x is unchanged.

If you choose a rearrangement so that the absolute value of the differential of g(x) is less than 1 then successive values of x will converge on a solution.

The fixed point method involves rearranging the equation to the form x2=f(x1)+x1. Once again, you should iterate until x is unchanged. Whether this converges on a solution depends on the signs of the slopes of f(x) and f(x)+x.

It fails to converge for roughly half the equations I've tried. I've not yet found a case where the fixed point method finds a solution but the Newton-Raphson, Secant, Bisection or Regula-Falsi do not.

One of the dificulties of using these methods on a computer is that you must inspect the function to see which method to use and to know whether it will work.

Things you could change

You could use an ESP32-C3. It looks like a nicer chip but I couldn't get it to work.

The "VDU" and "keyboard" code should work on many other processors. For instance, they should work on a Nano. However, a Nano has only 2k of RAM which will limit the size of the "page" of text.

You can add more keys to the keyboard. 10 pins would give 55 keys which is enough for a QUERTY keyboard.

Maybe you can work out how to use scrolling in the display (the ILI9341 can do it) so that we don't need to keep redrawing the screen.

You could replace the whole of the calculator software with other maths functions. Perhaps you like RPN calculators or you want a BASIC computer like the Sharp Pocket Computer. Maybe you love Lisp or APL or Forth.