ESP32Forth Programming Guide

by John Talbert 2026

Installation

The Forth install file, ESP32forth.ino, loads from the Arduino IDE just like any other Program Sketch, which is amazing considering that it results in a complete Forth programming development environment for your ESP32. The suggested ESP32 versions to use are the classic WROOM or DO-IT-KIT. The installation steps are simple if you are familiar with the Arduino IDE application.

1. The Arduino IDE application can be downloaded from https://www.arduino.cc/en/software/

2. There is a necessary add-on Library for the Arduino IDE that allows you to create and run programs on the ESP32. Thanks to Random Nerd Tutorials for providing a tutorial on how to install and use this Library. https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/

3. Once your ESP32 is connected to your computer through a USB cable, it should show up under the Arduino TOOLS Menu where you can select exactly what kind of ESP32 you have. If it does not show up, check your USB cable. There are many USB cables out there that are charge-only and are not well labeled as such.

4. Under the Tools Menu are some settings for your ESP32. Make sure they are set as shown here:

```
+-- CPU Frequency ---------+-- 240 Mhz
+-- Core Debug Level ------+-- None
+-- Erase All Flash...-----+-- Disabled
+-- Events Run On ---------+-- Core 1
+-- Flash Frequency -------+-- 80 Mhz
+-- Flash Mode ------------+-- QIO
+-- Flash Size ------------+-- 4MB
+-- JTAG Adapter ----------+-- FTDI Adapter
+-- Arduino Runs on -------+-- Core 1
+-- PSRAM -----------------+-- Disabled
+-- Partition Scheme ------+-- Default 4MB with SPIFFS
+-- Upload Speed ----------+-- 921600
```

5. Download the latest version of ESP32forth from https://esp32forth.appspot.com/ESP32forth.html and unzip it. You should find a folder named ESP32forth with the install file ESP32forth.ino inside. Do not change the folder or file name. There are several header files (.h) available in an "optional" folder that add specific capabilities to the installation. Choose which ones you are interested in and move them into the ESP32forth folder that holds the ESP32forth.ino file.

6. Under the Arduino FILE Menu choose OPEN, and then find your unzipped ESP32forth.ino installation file. Finally, select the UPLOAD button for it to compile and load. It is a large file and will take a full minute to finish.

7. If all goes well you should be ready to program your ESP32 in Forth Language from a Serial Terminal Application or the Monitor available as an Arduino IDE Tool. On the Mac I use "Serial" by decisivetactics.com.

8. The ultimate go-to guide is **"The Great Book for ESP32forth"** by Marc Petremann. It covers everything from installation to hardware control, making it a perfect aid to driving your ESP32 board using Forth programming language. Download it from Marc's Github page at https://github.com/MPETREMANN11/ESP32forth. If you are looking for more foundational reading on the environment by the creators Dr. Ting and Brad Nelson, you can also check out the ESP32forth Documents Page.

Program Development

The ESP32forth microprocessor board is connected to your main computer through a USB Serial cable. The main computer can then talk to the ESP32Forth Microprocessor using any Serial Terminal Application, or even the Monitor Tool provided within the Arduino IDE. Be sure to config the serial port to: baud rate = 115200, data bits = 8, stop bits = 1, and parity = N.

Compiling and running ESP32Forth "word" commands for testing can be accomplished by directly typing them onto one line in the Serial App. They are executed upon pressing the ENTER key. The ESP32Forth system has a large "dictionary" of word commands stored in flash memory. Type in the command "words" to see the current list of words in its dictionary. Most of these are part of a standard Forth library. Explore the website https://books.forth2020.org/ for a list of available books on the Forth Language and its standard dictionary of words.

Extending the Forth Dictionary with your own word commands is most commonly accomplished with the following program construct:

```
:  word-name   your code made up of previously defined words   ;  (  n1 n2  -- n3 )
```

The parenthesized item is a non-executing comment noting what values are expected on the stack before your word command is executed and what is left on the stack after finished executing. Stack operations are probably the most important thing to figure out to get comfortable with the language. Note that your created words will extend the Forth Dictionary into RAM memory space which will disappear after any reset or power cycling. Read on to find out how to get them into the non-volatile flash memory space.

Forth Development is usually done from your main computer using a simple text editor like TextEdit on the Apple Mac or an IDE programming environment like Visual Studio Code or BBedit, both of which have GIT capabilities. Then use a simple Copy and Paste to get your work into the ESP32Forth -- copy the code from any text editor on your main computer and then paste it onto the Serial Application's Forth Terminal Screen at the cursor.

FLASH MEMORY STORAGE

For storing larger amounts of development code, the ESP32 has 4MB, 8MB or 16MB of flash memory, usually on a separate chip. This flash memory is non-volatile, meaning that the data stored there will remain even after powering down or resetting

the ESP32. It has limited write cycles, as low as 10,000, so treat it as semi permanent memory space, not as a working memory space such as RAM. Since it exists on a separate chip, the main ESP32 processor communicates with it over a serial line, specifically SPI, making access to it much slower than the processor's on board RAM memory. Because of this the Forth flash access words will create a buffer in RAM memory from which to edit and load your code. When done, the buffers are then transferred to Flash memory.

There are several methods for loading and editing Forth Programs, both new words and immediate executable code lines, into the Flash Memory Space. All the methods involve **copying** the text code from some program development app on your main computer, and then **pasting** it into the Terminal Application that is talking with the ESP32Forth micro over USB serial lines. ESP32Forth cannot directly read a program file from your main computer, which is why you must manually transfer the program data to the ESP32's own Flash Memory using **copy/paste** from your main working computer.

Once loaded onto the Flash Memory space, the code can easily be accessed by the ESP32forth system. Flash Memory files can be read, listed, copied, removed, edited, and most important - compiled into the Forth dictionary and/or executed. What follows are four methods for loading and editing Forth Programs.

SIMPLE DIRECT METHOD - 1

The ESP32forth.ino system was actually loaded and stored directly onto the ESP32's flash memory by the Arduino IDE when you pressed UPLOAD, as it would with any program sketch. It is programmed for two-way serial communication with a host computer using any Terminal Application. You can create new words or execute existing words by simply typing the code onto a line on the Terminal App. The line can be edited by backspace erasing and will not be compiled or executed until the "Return" key is pressed to complete the code entry.

"Serial", a Terminal program for the Mac by decisivetactics.com has a space at the bottom of the terminal screen where you can paste or type in one line of code and edit it. It will load the line when you hit Return. Pasting more than one line of code there doesn't quite seem to work. Under its File Menu you can also "Send File". The Terminal will process the file one line at a time showing you its progress in the Terminal Window. I also found that you can simply paste any number of lines of code at the Terminal cursor and it will process them one line at a time. Other Terminal Applications

may have different code entry features.

In this way you can directly and immediately create new words for the Forth Dictionary or execute existing words; however, with this method the forth dictionary will be extended with your new words into RAM memory space, not the Flash Memory space which holds all the core ESP32forth system words. Those new words will disappear when the ESP32 is rebooted or powered off. This is fine for short, quick, and simple code explorations, but may prove cumbersome for larger projects. You may also find that the rebooting is useful in the beginning stages of your work for erasing it all and starting over. There is a tiny pushbutton on the ESP32 board used for rebooting the ESP32forth system back to its starting vocabulary. Note also that a word can be redefined in the same session and any future use of it will use the latest defined version.

BLOCK EDITOR METHOD - 2

Block-based Edit and Storage has been a feature of Forth Programming Systems since its beginning. It was originally used to access storage space on hard and floppy disk drives. Our "floppy drive" here is the Flash Memory Chip incorporated on the ESP32 microprocessor board.

A block is a storage space of 1024 bytes, which consists of 16 lines of 64 ASCII characters per line. With an available space of about 1.8MB you can easily have hundreds of these blocks stored in the Flash Memory.

A Block to be edited is first moved from Flash Memory to a RAM Buffer. Editing a block is accomplished within the RAM buffers, not the Flash Memory space, one 64 character line at a time. Here are the word commands available with the Block Editor:

editor	opens the vocabulary set of words for the block editor
n list	lists the contents of block n, and makes n the current block
l	lists the contents of the current block, along with the Block number
n	selects the next block
p	selects the previous block
wipe	empties the contents of the current block
from to copy	copies contents of block "from" to block "to"
n d	deletes line n (0 to 15 range), and moves up all lines below n
n e	erases the contents of line n, lines around n stay in place

n a	inserts the text that follows at line n, moves down all lines below n
n r	replaces contents of line n with text that follows, terminated with <cr>
flush	Save all buffer blocks back to Flash Memory and then empty the buffers
save-buffers	Save all buffer blocks back to Flash Memory
n load	Compile and/or Execute code in block n
n m thru	Compile and/or Execute code in blocks n through m

As you can see here, the block contents are filled one line at a time. Your work might entail creating a forth code work file on a text editor from your main computer and then copying the code three lines at a time from your text editor work file. Here is an example:

```
0 r    / my silly code test
1 r   : print44 44 . ;
2 r   print44
```

Copy these three lines from your work file, and then paste them to the block editor running from the Terminal Application. Note that the "n r" editor replace command is included at the start of each line. The block "editor" will re-list the block contents after each line entry. Continue this for all 16 lines in the block. When finished editing a block be sure to use the "flush" command to save your work, moving it from RAM Buffers back to Flash Storage.

The "n load" command will compile and/or execute all the code from Block number "n". One special Block could be filled with "n m thru" and "n load" commands along with progress report print commands to serve as the main source code loading block. You will find that the speed of execution is almost instantaneous.

The source code stored in these Blocks will be available for all future ESP32Forth sessions even after power down and reboot cycles.

FILE BASED DISK ACCESS METHOD - 3

It was inevitable that the ancient Block Disk Access, described above, would be replaced by something less constricting. When modern Forth implementations showed up running under host systems such as Windows, Linux, or MacOS, a Forth Vocabulary was developed to enable access to files living on those host systems.

Forth Data files are created and managed on the host computer. They are simple text files of ASCII characters and, unlike the Block Disks, can be of any size. The filenames use character strings and can include system path information using the forward slash character (/) to designate a folder structure.

However, there is a problem. The ESP32Forth system lives on and operates from the ESP32 micro board, not a host computer system. It can only communicate with a host computer through a Serial USB cable and a Terminal Application program on the host computer. This means that the ESP32Forth system does not have direct access to files on a host operating system using the File Based Disk Access word vocabulary.

There is a way around this problem. Check out this short word definition developed by Bob Edwards that draws from the File Access word vocabulary:

```
\ These chars terminate all text lines in a file
create crlf 13 C, 10 C,
\ Records the input stream to a spiffs file until
\ an <EOF> marker is encountered, then close file

: RECORDFILE ( "filename" "filecontents" "<EOF>" -- )
     bl parse \ read the filename ( a n )
     W/O CREATE-FILE throw >R \ create the file to record to -
          \ put file id on R stack
     BEGIN
          \ read a line of the file from the input stream
          tib #tib accept
          tib over
          S" <EOF>" startswith? \ does the line start with <EOF> ?
          DUP IF
               \ Yes, so drop the end line of the file containing <EOF>
               swap drop
          ELSE
               swap
               tib swap
               \ No, so write the line to the open file
               R@ WRITE-FILE throw
               \ and terminate line with cr-lf
               crlf 2 R@ WRITE-FILE throw
          THEN
     UNTIL                    \ repeat until <EOF> found
     R> CLOSE-FILE throw      \ Close the file
;
```

CREATE-FILE Creates a file in ESP32Forth with the pathname designated after the **RECORDFILE** word command and gives it a file ID.

WRITE-FILE Will write a designated number of characters into the file referenced by file ID. It will update FILE-POSITION and FILE-SIZE, two variables associated with the file ID. The characters are picked up from a given address in ESP32 RAM memory space.

CLOSE-FILE Will close the file identified by file ID.

These three word definitions were created to handle Forth's File based disk access. One more word definition will help explain what is happening here in **RECORDFILE**.

tib Will return the address of the terminal input buffer where the serial input text stream is being held. This sets up the location where **WRITE-FILE** will pick up its ASCII characters.

Here is a demonstration of how RECORDFILE is used:

```
RECORDFILE  /spiffs/mysilly.fs
/ my silly code test
: print44 44 . ;
print44
<EOF>
```

When this code is entered onto the Terminal Application window a new file called mysilly.fs (fs stands for forth source) is created on the ESP32Forth system with the content shown. The three lines of code here can be pasted from a text work file on the host computer. The pasted code can be of any length. Note the <EOF> end of file marker at the end of the code. This must be included and must be capitalized.

We could also create a master loading file that compiles and/or executes the code from several forth source files stored in flash memory as shown here.

```
RECORDFILE  /spiffs/mymain.fs

s" /spiffs/mysilly.fs"   included
CR ." print44 word compiled" SPACE

s" /spiffs/mysilly2.fs"  included
CR ." mysilly2 loaded"

s" /spiffs/mysilly3.fs"  included
CR ." mysilly all loaded"
```

<EOF>

include /spiffs/mymain.fs

As you can see, with this file access method we are still stuck with the same copy/paste to the Terminal App window, but we are left with a much more convenient file based storage. **RECORDFILE** is a good method for creating and loading Forth data files onto the ESP32Forth Flash Memory. What follows are words defined to manage our newly created files in the ESP32 Flash memory.

FLASH MEMORY SPIFFS FILE MANAGEMENT

Flash memory is on a separate chip on the ESP32 board. The ESP32Forth system uses a special file management system to read, write, and delete files on the flash memory chip over the serial protocol SPI. It is called SPIFFS (Serial Peripheral Interface Flash File System).

There are several ESP32forth words designed to handle Flash memory files over SPIFFS:

To see a list of files, including block files, stored in SPIFFS use the word **ls** :

```
ls /spiffs/     \  to list all the files
ls /spiffs/dir1  \  to list only those files in the subdirectory dir1
```

To compile the contents of a source file use the word **include**:

```
include /spiffs/MyFile.fs
```

To delete a file from SPIFFS use the word **rm** :

```
rm /spiffs/MyFile.fs
rm /spiffs/dir1/MyFile.fs
```

To copy a file use the word **cp** :

```
cp /spiffs/MyFile.fs  /spiffs/dir1/MyFile2.fs
```

To see the contents of a file, use the word **cat** :

```
cat /spiffs/MyFile.fs
```

To create a new empty file in SPIFFS, use the word **touch** :

```
touch /spiffs/MyNewFile.fs
```

To insert string contents into a file, use the word **dump-file** :

```
r|    : mysillydef  2 dup dup ;     |
s"    /spiffs/MyNewFile.fs"  dump-file
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

r| ***** | *(string| -- a n)*
Creates a temporary counted string ending with |
s" **** " *(-- addr cnt)*
creates a string until a terminating ". Leaves the string address addr and the character count cnt on the stack
dump-file *(a n a n --)*

VISUAL EDITOR METHOD - 4

Use **visual edit** to create and edit source files in the SPIFFS Flash file system.

visual edit /spiffs/myfile.fs

If myfile.fs exists it will come up in the editor. If it doesn't exist, it is created. The entire text file will be listed in the Terminal screen. A blinking "|" text insertion cursor will be available within the body of the text. This text cursor can be moved up, down, left, and right using the arrow keys on your keyboard. Delete/Backspace can be used to erase characters. Code is entered and edited by typing it in. Use Return at the end of lines. Text Copy/Paste is also supported.

A block cursor will also appear below the text file. Make sure that it is not sitting on top of any of the text. If it is, type a Return at the end of the last text line. This block cursor is where the "control" exit commands are displayed. To save the file being edited type CTRL-S. To exit the editor type CTRL-X followed by a Y to save all changes or an N to exit without file changes.

Here is a practical initial use for visual editor:

visual edit /spiffs/autoexec.fs

autoexec.fs is a special system forth source file whose contents are automatically loaded at startup. After the visual edit line shown above, type in or paste the code for the **RECORDFILE** definition displayed previously, including the crlf word. Then press CTRL-S to save it and CTRL-X and Y to exit visual. Use ls, cat and include to test that
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

autoexec.fs is correct and working.

Relaunch ESP32Forth. If all went well, **RECORDFILE** is now available at the top of the WORD stack whenever ESP32Forth restarts. You are welcome to add other word commands into the autoexec.fs file.

GPIO PINS

Access to the ESP32 GPIO Pins using ESP32Forth programming should look familiar to all who have experience programming the ESP32 from the Arduino IDE. The commands are very similar. Check the website https://www.xecor.com/blog/esp32-pinout-diagram for your ESP32 version and a pinout showing the functions each pin is capable of serving, and the few pins that should be left alone.

With Forth there is also the possibility of directly accessing the ESP32 registers that control the GPIO pins by using bit masks and binary bit manipulations with the words m@ and m!. This is useful when the programming calls for fast and complex Input/ Output manipulations. See Marc PETREMANN's (petremann@arduino-forth.com) pdf book for a good description and examples of this.

Here, though, I would like to make GPIO access as simple and clear as possible. What follows is a simple LED blink program example in Forth Programming Language. Four word definitions are created and the forth one, the "blink" definition, which incorporates the previous three, is executed.

```
: led-setup 2 OUTPUT pinMode ;
: led-on HIGH 2 pin ;
: led-off LOW 2 pin ;
: blink led-setup begin led-on 500 ms led-off 500 ms key? until ;
  blink
```

How it Works:

***2 OUTPUT pinMode**: Configures GPIO pin 2 as the output to the LED.*

If there is an LED onboard the ESP32 it is usually connected to pin 2. If you connect your own LED, the anode side (the longer wire lead) is connected to a GPIO pin and the cathode side (the lead nearest a straight edge on the casing) is connected to a resistor of about 200 ohms to Ground, which determines how bright it lights up. No harm if you connect it backwards, it just won't light. Turn it around.

***pin** (value pin# --) is the same as **digitalWrite** (pin value --)*

***HIGH 2 pin**: Sets pin 2 to HIGH (3.3V) to turn the LED on.*

***LOW 2 pin**: Sets pin 2 to LOW (0V, Ground) to turn the LED off.*

***500 ms**: Pauses execution for 500 milliseconds (half a second).*

***begin ... key? until**: Creates an infinite loop that blinks the LED repeatedly until you press any key on your keyboard to jump out of the loop, which stops it.*

GPIO INPUT/OUTPUT Words

adc *(pin# -- n) short alias for analogRead*
analogRead *(pin -- n) Analog read, n result from 0-4095*
dacWrite *(pin 0-255 --) Write to DAC (pin 25, 26)*
digitalRead *(pin -- value) Read GPIO state*
digitalWrite *(pin value --) Set GPIO pin state*
pin *(value pin# --) short alias for digitalWrite*
pinMode *(pin mode --) Set GPIO pin mode*
pulseIn *(pin value usec -- usec/0) Wait for a pulse*
tone *(channel freq) Write tone frequency*

HIGH LOW INPUT OUTPUT

Low Level words In the Interrupts Vocabulary

ESP_INTR_FLAG_DEFAULT -- Default handler allows per pin routing

Various triggers:
#GPIO_INTR_DISABLE
#GPIO_INTR_POSEDGE
#GPIO_INTR_NEGEDGE
#GPIO_INTR_ANYEDGE
#GPIO_INTR_LOW_LEVEL
#GPIO_INTR_HIGH_LEVEL

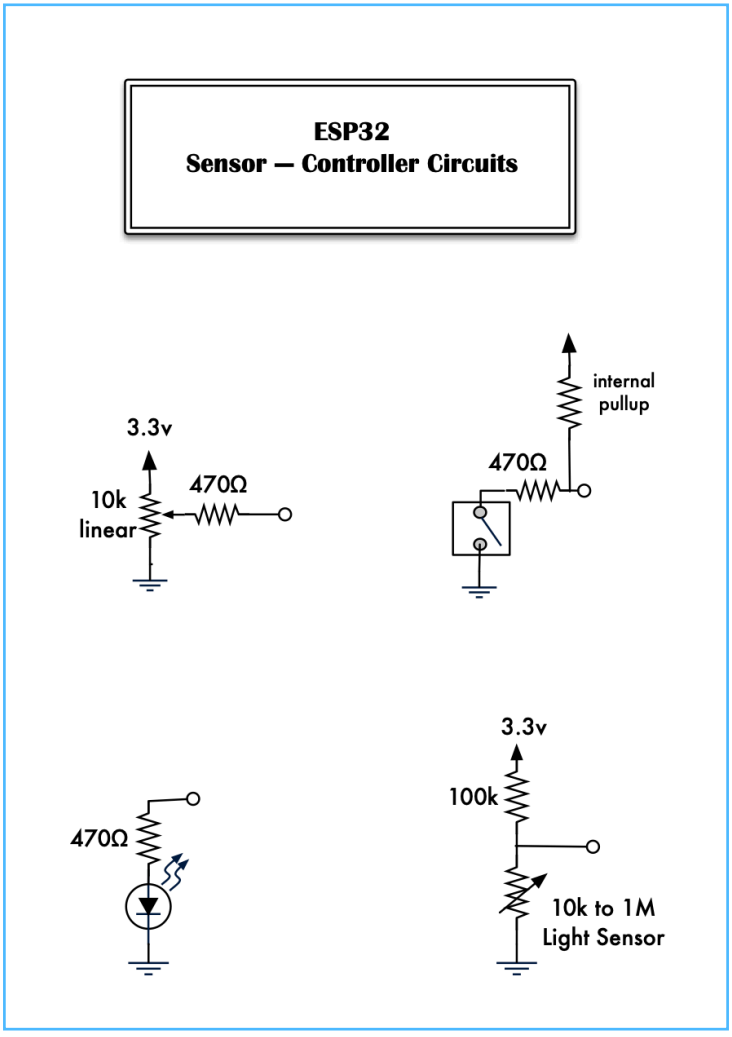
gpio_config (gpio_config_t* -- 0/err)
gpio_reset_pin (pin -- 0/err)
gpio_set_intr_type (pin type -- 0/err)
gpio_intr_enable (pin -- 0/err)
gpio_intr_disable (pin -- 0/err)
gpio_set_level (pin level -- 0/err)
gpio_get_level (pin -- level)
gpio_set_direction (pin mode -- 0/err)
gpio_set_pull_mode (pin mode -- 0/err)
gpio_wakeup_enable (pin type -- 0/err)

gpio_wakeup_disable (pin -- 0/err)
gpio_pullup_en (pin -- 0/err)
gpio_pullup_dis (pin -- 0/err)
gpio_pulldown_en (pin -- 0/err)
gpio_pulldown_dis (pin -- 0/err)
gpio_hold_en (pin -- 0/err)
gpio_hold_dis (pin -- 0/err)
gpio_deep_sleep_hold_en (--)
gpio_deep_sleep_hold_dis (--)
gpio_install_isr_service (a --) Typically ESP_INTR_FLAG_DEFAULT
gpio_uninstall_isr_service
gpio_isr_handler_add (pin xt arg -- 0/err)
gpio_isr_handler_remove (pin -- 0/err)
gpio_set_drive_capability (pin cap -- 0/err)
gpio_get_drive_capability (pin cap* -- 0/err)
esp_intr_alloc (source flags xt args handle* -- 0/err)
esp_intr_free (handle -- 0/err)

Sample GPIO Programs

What follows are heavily commented short program examples for simple GPIO sensor circuits. (The pin connections are based on my ESP32 4-voice sound synthesizer project described at https://jtalbert.xyz/esp32/.)

The figure below illustrates four sensor circuits. The small circle in each circuit is the connection point to an ESP32 GPIO pin. Most GPIO pins can be connected to any of these circuits, especially those pins labelled as ADC capable. Note the 470 ohm resistor in the potentiometer and switch circuits. This small resistance acts as short circuit (to ground or 3.3v) protection if the pin is mistakenly setup as an output instead of input.

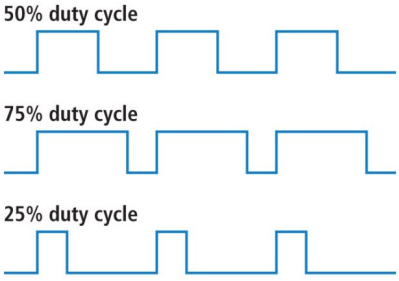


LED LIGHTS

The 470 ohm resistor in series with the LED controls how much current goes through the LED and therefore affects how bright the LED shines. It can be set as low as about 100 ohms for a very bright light. The color of the LED will also affect the brightness. A blue LED, for example, will require a lower resistor. Try out different resistances to get the result you want. Be careful not to connect a live GPIO pin directly to a voltage or Ground which can blow the ESP32 chip!

When a GPIO pin, set up as a digital output, is high at 3.3 volts, the LED will light. When it is a low of ground (0 volts) there is no voltage difference, therefore no current and the LED will be dark.

If the GPIO pin is programmed to oscillate high and low at a fast frequency the LED will seem dimmer because it is only "on" half of the time. With PWM, pulse width modulation, you can control the LED brightness with the oscillation's duty cycle, the percentage of each cycle that is high versus low. The **ledc** vocabulary in ESP32Forth has words designed to work with PWM waveforms controlling LEDs.



The LED circuit can be connected in several ways. The resistor could be inserted into the bottom ground connection instead of the top pin connection as shown. The circuit ground connection could instead be a 3.3v connection, in which case a Low voltage on the GPIO pin will turn on the LED. In order to be lit the LED circuit arrow must point in the direction of high to low voltage. If it is connected backwards, no harm is done, it just won't light up.

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ AY Synth Board, routines for using the LED pins
\ John Talbert  July 2026
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 \ pins used for LEDs

5   CONSTANT LED1
15  CONSTANT LED2
2   CONSTANT LED3    \ the Blue LED on ESP32S boards

 \  setup for LED pins as outputs

LED1 OUTPUT pinMode  LED2 OUTPUT pinMode   LED3 OUTPUT pinMode

 \ words for turning an LED light off or on

: led-on HIGH digitalWrite ;  ( pin# -- )
: led-off LOW  digitalWrite ;  ( pin# -- )

VARIABLE timeOn   \ to load this variable:  200 timeOn !
VARIABLE timeOff

: led-blink     BEGIN
                     LED1 led-on LED2 led-off LED3 led-on   timeOn  @ MS
                     LED1 led-off LED2 led-on LED3 led-off  timeOff @ MS
                KEY? UNTIL
;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



\ LED fade using Pulse Width Modulation provided in the LEDC vocabulary.

ledc  \ open ledc vocabulary. Using version 3 -- Pin based instead of channels.

: led-fade  LED2 100 8 ledcAttach DROP
                BEGIN
                     255 0 DO LED2 I ledcWrite 20 ms LOOP
                KEY? UNTIL
                LED2 ledcDetach DROP
;
```

The above code sets LED2 to oscillate at 100Hz. The PWM duty cycle has an 8-bit resolution meaning that it can be set with values between 0 and 255. The duty cycle is then looped between 0 and 255, fading the LED in slowly over and over.

POTENTIOMETER CONTROLS

The top left circuit connection diagram in the circuit figure is for a Rotary or Slide Pot. The moving wiper of the pot is designated by an arrow and it can move between 0 volts of Ground and 3.3 volts at the other end. The ESP32 ADC (Analog to Digital Converter) converters have a 12-bit range of 0 to 4095 corresponding with voltages from 0 volts (Ground) to 3.3 volts (the ESP32 power supply). The pot wiper is connected to an ADC capable GPIO pin after a small protection resistor. The word analogRead (pin# -- n result) is used to read the pot's wiper voltage.

The pot resistance (shown as 10k) can be anything above around 5k, but it must have a Linear Taper, not Log, for a better response.

The lower right circuit connection diagram in the circuit figure is used for any variable resistance sensor. The arrow over the resistor indicates a resistance that varies between some low value and high value. This fits the description of a number of sensors - flex sensor, pressure sensor, light sensor, moisture sensor. It can even be as simple as two #2 pencil smudges on paper pressed together.

The 100k resistor in series with the variable resistor can be tweaked in value to obtain the greatest range of voltage at the center connection point. An analogRead will result in some minimum value above 0 and some maximum value below 4095, all depending on the minimum and maximum resistance values of the sensor and that fixed resistor.

You can mathematically find the optimum value for the fixed resistor. Measure the sensor's MAX and MIN resistance values. The highest voltage you can get from the circuit is

3.3 x (MAX/(MAX + R)).

The lowest voltage you can get is

3.3 x (MIN/(MIN + R)).

Subtrack the above highest equation from the lowest, graph that equation on a scientific calculator and read the value of R at its peak. Most likely it will be near 100k.

The following Forth code can be used for any variable resistance sensor.

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ AY Synth Board, routines for using Pot Controllers
\ or any analog sensor such as light sensors, flex sensors, pressure sensors
\ John Talbert  July 2026
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ ADC pins connected to potentiometers

36 CONSTANT POT1
39 CONSTANT POT2
34 CONSTANT POT3
35 CONSTANT POT4

POT1 INPUT pinMode  POT2 INPUT pinMode  POT3 INPUT pinMode  POT4 INPUT pinMode

VARIABLE pot_value1
VARIABLE pot_value2
VARIABLE pot_value3
VARIABLE pot_value4

: potRead!   ( pin#, variable address -- )  \ read one pot value
          SWAP analogRead 2 RSHIFT SWAP ! ;

: allPotRead! ( -- )  \ read 4 pot values and store them in variables
          POT1 pot_value1 potread!
          POT2 pot_value2 potread!
          POT3 pot_value3 potread!
          POT4 pot_value4 potread!
;

: potPrint ( -- )  \ print out all 4 pot values until any key is pressed
          BEGIN
               allPotRead!
               CR   pot_value1 @ . 3 SPACES
                     pot_value2 @ . 3 SPACES
                      pot_value3 @ . 3 SPACES
                       pot_value4 @ . 3 SPACES
          KEY? UNTIL
;

\ ADC pin resolutions are actually 12-bit, 0-4095
\ to take out some of the result jitter, a right bit shift is included in potRead
\  1 RSHIFT   (for values 0 to 2047)
\  2 RSHIFT   (for values 0 to 1023)

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

SWITCH CONTROLS

The top right circuit in the circuit figure shows the circuit hookup for a simple SPST switch, either toggle or pushbutton. It shows the bottom of the switch connected to Ground and the top pin connection with a "pullup" resistor to 3.3 volts. There is also a small 470 ohm resistor that protects the pin from being directly shorted to Ground in case the pin is set up as an output by mistake.

There are two possibilities for the pullup resistor. It can be an actual physical resistor of around 100k connected between the pin and 3.3 volt power, or it can be provided by the ESP32 as an "internal pullup". Enabling or disabling an internal pullup for a GPIO pin involves a couple words found in the "interrupts" vocabulary. See the forth code below for how to implement an internal pullup in code.

Other types of digital switches can use the same digitalRead word, sensors like the PIR Infrared Motion detector, a magnetic contact Reed Switch, photoelectric light beam sensor, proximity switch, Tilt switch, Foot Pedals. Some of these require you to hook up a circuit as shown in the figure with a pullup resistor. Others will include the circuitry for a digital output and may just need to be powered with Ground and 3.3v connections.

As an interesting side note, the ESP32/Arduino platform has the capability of setting up some ESP32 pins as Capacitive touch switches using the C function touchRead( ). However, at the present time, the touchRead command is not available on the ESP32forth platform. A way around this is to use the TTP223 sensor, an inexpensive sensor device that converts capacitive touch to a simple digital output, requiring only power connections.

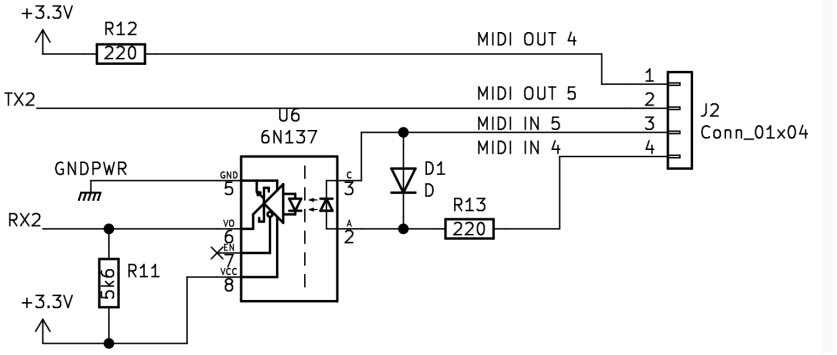
```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ AY Synth Board, routines for using Switches
\ John Talbert  July 2026
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ pins connected to Switches, 0 is pressed, 1 is unpressed

33 CONSTANT SWITCH1
32 CONSTANT SWITCH2

VARIABLE switch_value1
VARIABLE switch_value2
```

```
SWITCH1 INPUT pinMode   SWITCH2 INPUT pinMode

\ gpio words in the interrupts vocabulary can enable or disable pin pullups
\ gpio_pullup_en ( pin -- 0/err ), gpio_pullup_dis ( pin -- 0/err )

interrupts                      \ open interrupts vocabulary
SWITCH1 gpio_pullup_en drop     \ enable pullups for each switch
SWITCH2 gpio_pullup_en drop
forth


: switchPrint ( == )   \ print the two switch values until any key is pressed
          BEGIN
               SWITCH1 digitalRead switch_value1 !
               SWITCH2 digitalRead switch_value2 !
               CR
               switch_value1 @ .   3 SPACES   switch_value2 @ .
          KEY? UNTIL
;\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

MIDI Input/Output

The ESP32 WROOM has three Hardware UARTs for serial communications. One of them is used for the ESP32's USB connection (simply called UART with word commands labeled as Serial.xxx). That leaves UART1 and UART2. MIDI I/O on the ESP32 normally uses UART2 with Forth word commands labeled as Serial2.xxxx in the "serial" vocabulary. GPIO pin 17 is usually the MIDI TX or transmit output and pin 16 is the MIDI RX or Receive Input. (Note that this won't work on the ESP32 WROVER.)

A small amount of circuitry is required to connect the TX and RX pins to two 5-pin DIN sockets, one for MIDI IN and one for MIDI OUT. As you can see, TX is very easy, needing just one 220 ohm resistor from the 3.3 volt board power and a direct TX pin connection to the DIN MIDI OUT plug (you may want to include a small 47 ohm protection resistor in series). The RX MIDI INPUT side is a bit more involved required an opto-isolator chip.



Modern MIDI I/O has moved on to a USB type interface. The old 5-pin DIN plugs are rarely used now but are useful in cases like this where the ESP32 board has only one USB connection available. Inexpensive cable converters are readily available from Amazon for $20 that convert the two MIDI DIN I/O to USB.

ESP32forth includes the following set of word commands in its "serial" vocabulary for the Serial UART interface.

Serial communication			
Serial.available	(-- f)	Get the number of bytes (characters) available for reading from the serial port. This is data that's already arrived and stored in the serial receive buffer (which holds 64 bytes)	serial
Serial.begin	(baud --)	Start serial port. Sets the data rate in bits per second (baud) for serial data transmission	serial
Serial.end	(--)	Disables serial communication, allowing the RX and TX pins to be used for general input and output. To re-enable serial communication, call Serial.begin	serial
Serial.flush	(--)	Waits for the transmission of outgoing serial data to complete	serial
Serial.readBytes	(a length -- n)	Serial.readBytes reads characters from the serial port into a buffer, address a. The function terminates if the determined length has been read. The number of bytes, n is returned	serial
Serial.write	(a n -- n)	Writes n bytes of data to the serial port from buffer at address a	serial
the same SERIAL words are available for SERIAL PORT2			serial
for example Serial2.write, Serial2.Begin, Serial2.readbytes, Serial2.flush etc etc...			

These words are for the USB connection on the ESP32. The same set is available for the Serial UART2, labeled as Serial2. Basically, the serial interface is initialized with Serial2.begin, MIDI bytes are transmitted using Serial2.write, and MIDI bytes are received using Serial2.available and Serial2.readBytes.

The initializing command, Serial2.begin (baudrate --), provided in the vocabulary only requires you to specify the serial baudrate, which for MIDI is 31250. It was assumed that the default TX pin, RX pin, and Byte format values are already set up. However, if you find your serial code is not doing anything (as I did), it may be that these default values are not engaged and you will need to set them. The problem is that there is no way to do this from the current ESP32forth version 7.0.8.0

Thank you Claude AI from Anthropic for suggesting a solution to this problem. It consists of changing one line in the ESP32forth.ino install file. This can be done from any text editor or even from the Arduino IDE before you reload it.

Find the following line at around 705:

XV(serial, "Serial2.begin", SERIAL2_BEGIN, Serial2.begin(tos); DROP) \

replace it with:
XV(serial, "Serial2.begin", SERIAL2_BEGIN, Serial2.begin(n3, n2, n1, n0); DROPn(4)) \

Reload the edited installation file from Arduino IDE. Your code can then use the full Serial2.begin command to initialize all the serial parameters:

```
MIDI_BAUD_RATE   SERIAL_8N1   MIDI_RX_PIN   MIDI_TX_PIN   Serial2.begin
                          or
     31250   $C00001C   16   17   Serial2.begin
```

MIDI OUTPUT

MIDI code is transmitted one byte (8-bits) at a time along with one start bit, one stop bit, and no parity (Serial_8N1). There are two main types of MIDI byte codes, Status and Data bytes. The STATUS byte has a 1 in the most significant bit and designates the type of command such as NoteOn, NoteOff, ControlChange, ProgramChange in the higher 4 bits of the byte. The Status byte also has the MIDI Channel number (0 to 15) in the lower 4 bits. The DATA bytes all have a 0 in the most significant bit, constricting the data to 7 bits, with a range of 0 to 127. Each specific type of Status byte is followed by one or two data bytes. For example, a NoteOn Status byte is always followed by a velocity (loudness) data byte and a Note Number data byte.

With this information you can more easily follow along with the code below designed for MIDI Output:

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\  MIDI INPUT/OUTPUT Interface for ESP32Forth
\  by John Talbert July 2026
\
\ On an ESP32 WROOM MIDI can be set up on Serial2
\ RX2 and TX2 (pins 16 and 17). Does not work on ESP32 WROVER
```

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ UART2 May already to have the following settings in ESP32Forth:
 17  constant MIDI_TX_PIN
 16  constant MIDI_RX_PIN
 17 OUTPUT pinMode
 16 INPUT  pinMode

$C00001C constant SERIAL_8N1  \ 8 bits, no parity, 1 stop bit

31250 constant MIDI_BAUD_RATE

\ Some Standard MIDI Control Change Codes
 1 constant CC_MODWHEEL
 7 constant CC_VOLUME
10 constant CC_PAN
11 constant CC_EXPRESSION
64 constant CC_SUSTAIN    \ values 0-63 are Off, 64-127 are On
71 constant CC_RESONANCE
74 constant CC_FREQCUTOFF

VARIABLE chan   \ MIDI Channel, 0 to 15 only.
0 chan !       \ Load 0 into VARIABLE chan address

Serial    \ open Serial Vocabulary

\ Start up Serial2 for MIDI output with a 31250 baud rate
\ The old version:  ->  midiSetup  MIDI_BAUD_RATE Serial2.begin ;

: midiSetup MIDI_BAUD_RATE SERIAL_8N1 MIDI_RX_PIN MIDI_TX_PIN Serial2.begin ;

midiSetup


\ All MIDI values except status operation codes, must be in the range of 0 to 127
\ If needed to insure this do a "$7F AND" on MIDI values before MIDI.write

\ Serial2.write ( data address, number of bytes at address -- number of bytes sent)
\ for MIDI here we will always send just one byte of data at a time


VARIABLE byte_to_send   \ creates the address "byte_to_send" to hold a MIDI byte

: MIDI.write ( data byte to send -- )

               byte_to_send !  \ load MIDI data byte into byte_to_send address
               \   5 SPACES ." Midi Send " byte_to_send @ .
               byte_to_send 1 Serial2.write drop    \ MIDI data byte sent
;


: sendNoteOn ( velocity note -- )
               $90 chan @ OR MIDI.write \ NoteOn + channel
```

```
                MIDI.write   \ note
                MIDI.write   \ velocity
;


: sendNoteOff ( note -- )
                $90 chan @ OR MIDI.write  \ NoteOn + channel
                MIDI.write      \  a saved note that is on
                0 MIDI.write    \  velocity of zero to turn off note
;


: sendControlChange ( ccVal ccCode -- )
                $B0 chan @ OR MIDI.write  \ Control Change + channel
                MIDI.write    \ cc code
                MIDI.write    \ cc value
;


: sendProgramChange ( Program -- )
                $C0 chan @ OR MIDI.write    \ Program Change + channel
                MIDI.write                     \ Program change number
;
```

MIDI INPUT

Serial2 has an input buffer area of 64 bytes to receive incoming MIDI data bytes. Generally, the first step in your MIDI Input code is to check this buffer to see if any data has been received using the command Serial2.available (-- f). The resulting "f" value from this Serial2 command reveals how many bytes have been received in the serial input buffer. In our code we are only interested in whether or not it is non zero since we will be processing only one byte at a time.

If the buffer is non-empty, the Serial2.readBytes (address, length -- n) command is processed. This command requires the address of some pre-defined buffer and the number of bytes to dump into that buffer. It then returns the number of bytes read. For our purposes, only one byte will be read and it can be placed in a forth VARIABLE since this holds one data value and its name is actually the RAM memory address of the data, not the data itself. For example, "VARIABLE xxx" creates RAM memory space for one data cell. "xxx . " will print the variable address called xxx. "22 xxx !" will load the value 22 into address xxx. "xxx @ ." will print out 22, the value stored at address xxx. Thus " xxx 1 Serial2.readBytes" will remove one byte from the serial input buffer and load it into the address xxx.

The next step is to parse the byte just read and stored by Serial2.readBytes.

Command words can be built that are triggered by a specific MIDI status byte and then processed using its MIDI data bytes. In the example below we look only for MIDI NoteOn status bytes in order to print out the note number and velocity whenever one is received - a sort of MIDI Note Monitor. Anything not associated with MIDI NoteOn is ignored. Another example might be to build a chord out of the incoming NoteOn's by transmitting two more NoteOn's for each incoming NoteOn. Parsing for any number of the other incoming Status bytes could easily be done using a forth CASE structure, each specific status byte having its own special triggered word command.

Here is some sample code for performing MIDI input on Serial2.

```
\ Simple MIDI INPUT
\ by John Talbert July 2026

\ Use Serial2.available ( -- n) and Serial2.readBytes ( address, length -- n)
\ to read incoming MIDI one byte at a time.
\
\ If a NoteOn command is received, get the next two bytes Note# and Velocity
\ and do something with that info, else ignore the data.

\ Be sure to first run Serial2.begin (midisetup) and open the serial vocabulary
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

     VARIABLE byteIN
     VARIABLE note#
     VARIABLE velocity

     \ check the serial input buffer, read one byte if something is in it
     \ and send it to MIDI.parse

     : MIDI.read  ( -- byteIN@)
                  \ check for one byte of input and send it to MIDI.parse

                    Serial2.available
                    IF        byteIN 1 Serial2.readBytes DROP
                              byteIN C@ MIDI.parse
                    THEN
     ;

     : HandleNoteOn ( -- )  \ Do something when NoteOn is received
                CR ."   Channel  " byteIN @ $F AND .
                   ."   Note  "          note# @ .
                   ."   Velocity  "     velocity @ .

                    \  note# @ 4 + 55 sendNoteOn  note# @ 7 + 55 sendNoteOn 500 ms
                    \  note# @ 4 +0   sendNoteOn  note# @ 7 + 0  sendNoteOn
```

```
;

\ If MIDI NoteOn is received then get note# and velocity to run HandleNoteOn
\ Serial2.readBytes will wait and time out at 1 sec if no data comes in.

: MIDI.parse ( byteIN@ -- )
               $F0 AND $90 =
               IF        \ Is it a NoteOn MIDI Command?

               note#    1 Serial2.readBytes DROP   \ load note#
               velocity 1 Serial2.readBytes DROP   \ load velocity

               velocity @ 0<> IF HandleNoteOn THEN \ NoteOff check

               THEN  \ Do nothing with input byte if it's not NoteOn
;


\ Continuously watch for MIDI Input. Run MIDI.parse when something shows up
: MIDI.watch ( -- )
               ." Begin MIDI Input Watch"
               BEGIN
                    MIDI.read
               KEY?   \ stop on key press, not MIDI In activity
               UNTIL
                    CR ." End MIDI Input Watch"
;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Some code observations:

In MIDI.parse it might be more correct coding to continuously check Serial2.available for a return of 2 or more, indicating that at least two bytes are in the input buffer, before running Serial2.readBytes for the note# and velocity.

If you are wanting to run other code, besides just this MIDI Input polling, Forth does have multitasking capabilities in which case you may want a YIELD inside the MIDI.watch command to allow other tasks a chance to run.

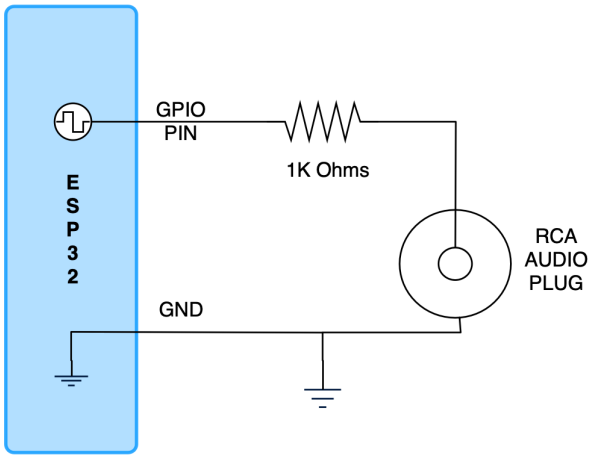
To emphasize the wide possibilities of this MIDI Input code, any number of parsing triggers can be set up in a CASE structure. Consider coding triggers on different MIDI Notes, setting up your MIDI keyboard as a kind of command center. The resulting "Handle" word commands can perform any tasks you can imagine, though you will want to make them fast and efficient so as not to slow down the polling.

AUDIO

Audio signals, between about 30Hz and 10KHz, can easily be produced on most any GPIO pin. Set the pin as an output with pinMode and then oscillate it between high and low digital output values using time delays to set the frequency.

```
22 CONSTANT AUDIO1  \ here pin 22 is given the name AUDIO1
AUDIO1 OUTPUT pinMode
: run-audio   \ wavecycle of 2ms -> 500 cycles per second
     BEGIN
     AUDIO1 LOW  digitalWrite  1 ms
     AUDIO1 HIGH digitalWrite  1 ms
     KEY? UNTIL
;
```

This GPIO pin signal cannot directly drive a speaker cone. That would require an audio amplifier, which can be as simple as a transistor circuit. However, it can easily be wired to an audio connector along with Ground, and then connected to an audio amplifier input or any audio device with a "line" level input connection and at least a headphones output. It is good practice to add a 1k Ohm resistor in series with the pin output to protect the ESP32.



LEDC VOCABULARY

In spite of its name association with LEDs, LEDC is the ideal vocabulary for producing audio squarewave and pulsewaves on a GPIO pin. Best of all, it has no CPU cost since it uses ESP32 internal hardware timers to create the signal.

```
ledcAttach ( pin#, frequency,  dutycycle bit resolution -- flag )
```

This command is used to initialize a pin for use with the ledc vocabulary. It associates the pin with a separate channel and a hardware timer. It must be followed by **ledcWrite** to set a dutycycle before it will sound.

```
ledcWrite ( pin#, PWM dutycycle -- )
```

This is the main command for changing the duty cycle of the waveform. It will not affect the frequency, but will add more harmonics to the audio signal as the pulse width is narrowed. With a bit resolution of 8, set in ledcAttach, the pulse width can be varied with duty cycle values between 0 and 255. A value of 127 will produce a squarewave with 50% pulse width.

```
ledcWriteTone ( pin# freq -- freq)   \ Will set dutycycle to 50% --> squarewave
```

ledcWriteTone is the main command for changing the frequency of pin's audio signal. However, it will automatically set the duty cycle to 50% creating a squarewave. Use ledcChangeFrequency if you want to change the frequency without affecting the duty cycle. The frequency value is an integer given in Hertz, or cycles per second. Audio signals range from about 30Hz to 10,000Hz. To silence a signal use a frequency of zero.

ledcChangeFrequency (pin# freq res -- flag)

Used to change the frequency without affecting the duty cycle.

ledcWriteNote (pin# note octave -- freq)

For musical note applications this command allows you specify frequency as the notes on an "equal-tempered" musical keyboard, like a piano. "Note" specifies one of twelve notes in an octave -- C, C#, D, D#, E, F, F#, G, G#, A, A#, B -- with numbers 0 to 11. "Octave" specifies the keyboard octave with 0 being the lowest and 10 the highest.

To give you a better idea of the note frequencies on a musical keyboard here is a Forth byte array of MIDI NoteON frequency values from 0 to 127 that you can use instead with ledcWriteTone.

```
: ROM-TABLE  ( "name" -- )      \ defining word: Read Only Table or Array
    CREATE                      \ the data itself (via ,) sets the size
    DOES>  ( i addr -- value )   \ addr is the address of "name"
    SWAP CELLS + @               \ index -> address -> fetch, all in one
;

ROM-TABLE MIDI_Freq    \ create the Table MIDI_Freq, 128 NoteOn key frequencies

\  C     C#     D     D#      E      F     F#     G      G#     A      A#     B
   8 ,    9 ,    9 ,   10 ,   10 ,  11  ,  12  ,  12  ,  13  ,  14  ,  15  ,  15  ,
  16 ,   17 ,   18 ,   19 ,   21 ,  22  ,  23  ,  24  ,  26  ,  28  ,  29  ,  31  , \ C0
  33 ,   35 ,   37 ,   39 ,   41 ,  44  ,  46  ,  49  ,  52  ,  55  ,  58  ,  62  , \ C1
  65 ,   69 ,   73 ,   78 ,   82 ,  87  ,  92  ,  98  , 104  , 110  , 117  , 123  , \ C2
 131 ,  139 ,  147 ,  156 ,  165 , 175  , 185  , 196  , 208  , 220  , 233  , 247  , \ C3
 262 ,  277 ,  294 ,  311 ,  330 , 349  , 370  , 392  , 415  , 440  , 466  , 494  , \ C4
 523 ,  554 ,  587 ,  622 ,  659 , 698  , 740  , 784  , 831  , 880  , 932  , 988  , \ C5
1047 , 1109 , 1175 , 1245 , 1319 , 1397 , 1480 , 1568 , 1661 , 1760 , 1865 , 1976 , \ C6
2093 , 2217 , 2349 , 2489 , 2637 , 2794 , 2960 , 3136 , 3322 , 3520 , 3729 , 3951 , \ C7
4186 , 4435 , 4699 , 4978 , 5274 , 5588 , 5920 , 6272 , 6645 , 7040 , 7459 , 7902 , \ C8
8372 , 8870 , 9397 , 9956 , 10548 , 11175 , 11840 , 12544 ,                        \ C9

 USAGE:    69  MIDI_Freq .    -> 440
           127 MIDI_Freq .  -> 12544
```

~~~~~~
~~~~~~

These are the words found inside the version 3 **ledc vocabulary.** Note that this updated ledc vocabulary is more pin oriented, compared with version 2 which talks more of "channels". Channels are a way of grouping pins together so that a change to one will affect all in the group.

```
ledcAttach ( pin freq resolution -- flag ) \ Follow with ledcWrite
ledcAttachChannel ( pin freq resolution channel -- flag )
ledcDetach ( pin -- flag )
ledcRead ( pin -- n )
ledcReadFreq ( pin -- freq )
ledcWrite ( pin duty -- )  \ loads PWM duty, 0-255 with 8-bit resolution
ledcWriteTone ( pin freq -- freq ) \ Will set duty to squarewave 50%
ledcWriteNote ( pin note octave -- freq ) \ note 0-11, octave 0-9 ?
ledcChangeFrequency ( pin freq resolution -- flag ) \ Does not affect duty
```

Producing Audio signals from the ESP32 is easy if you stick to digital signals which are either high or low. ESP32forth makes it super convenient with the ledc vocabulary. With these commands a squarewave or pulse wave of any pitch can be produced on most of the ESP32 pins. You can even get a pitched noise waveform by sending random duty cycle values to the waveform. The main limitation here is that ledc only produces these digital, low or high value waveforms.

DAC WAVEFORMS

To get beyond the rather harsh sounding squarewaves you can employ DACs, digital to analog converters. The ESP32 provides two pins, 25 and 26, that can act as 8-bit DACs using the command dacWrite (pin, value ---). Here, "value" is loaded into one of the two available DAC pins and has a range of 0 to 255 (8-bits).

A zero written to the DAC pin 25 with "25 0 dacWrite" will produce 0 volts on pin 25. 255 written to DAC pin 25 with "25 255 dacWrite" will produce 3.3v on pin 25. So far, this does nothing more interesting than digitalWrite (pin, value --) with the value set as HIGH or LOW, which can also be used on the DAC pins 25 and 26. However, the command 26 127 dacWrite will produce a voltage of about 1.7 volts on pin 26, which is neither high or low, but right in the middle. Using dacWrite results in 256 possible voltages ranging from zero to 3.3 volts as the input value ranges from 0 to 255.

By loading dacWrite input values over time at a fast audio sample rate you can produce any sound from the DAC output pins, which is explored in the following forth code:

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ ESP32Forth Code to demonstrate dacWrite and Timers
\ John Talbert    July 2026

\ SineWave, SquareWave, and TriangleWave output on a DAC
\ using do-loop timers.  Waveforms selected from Keyboard
\ input along with frequency increment and decrement.
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ Sinewave Table - 256 entries, one wave cycle

CREATE Sinewave
127 C, 130 C, 133 C, 136 C, 139 C, 143 C, 146 C, 149 C, 152 C, 155 C,
158 C, 161 C, 164 C, 167 C, 170 C, 173 C, 176 C, 179 C, 182 C, 184 C,
187 C, 190 C, 193 C, 195 C, 198 C, 200 C, 203 C, 205 C, 208 C, 210 C,
213 C, 215 C, 217 C, 219 C, 221 C, 224 C, 226 C, 228 C, 229 C, 231 C,
233 C, 235 C, 236 C, 238 C, 239 C, 241 C, 242 C, 244 C, 245 C, 246 C,
247 C, 248 C, 249 C, 250 C, 251 C, 251 C, 252 C, 253 C, 253 C, 254 C,
254 C, 254 C, 254 C, 254 C, 255 C, 254 C, 254 C, 254 C, 254 C, 254 C,
253 C, 253 C, 252 C, 251 C, 251 C, 250 C, 249 C, 248 C, 247 C, 246 C,
245 C, 244 C, 242 C, 241 C, 239 C, 238 C, 236 C, 235 C, 233 C, 231 C,
229 C, 228 C, 226 C, 224 C, 221 C, 219 C, 217 C, 215 C, 213 C, 210 C,
208 C, 205 C, 203 C, 200 C, 198 C, 195 C, 193 C, 190 C, 187 C, 184 C,
182 C, 179 C, 176 C, 173 C, 170 C, 167 C, 164 C, 161 C, 158 C, 155 C,
152 C, 149 C, 146 C, 143 C, 139 C, 136 C, 133 C, 130 C, 127 C, 124 C,
121 C, 118 C, 115 C, 111 C, 108 C, 105 C, 102 C, 99 C, 96 C, 93 C,
90 C, 87 C, 84 C, 81 C, 78 C, 75 C, 72 C, 70 C, 67 C, 64 C, 61 C,
59 C, 56 C, 54 C, 51 C, 49 C, 46 C, 44 C, 41 C, 39 C, 37 C, 35 C,
33 C, 30 C, 28 C, 26 C, 25 C, 23 C, 21 C, 19 C, 18 C, 16 C, 15 C,
13 C, 12 C, 10 C, 9 C, 8 C, 7 C, 6 C, 5 C, 4 C, 3 C, 3 C, 2 C, 1 C,
1 C, 0 C, 0 C, 0 C, 0 C, 0 C, 0 C, 0 C, 0 C, 0 C, 0 C, 0 C, 1 C,
1 C, 2 C, 3 C, 3 C, 4 C, 5 C, 6 C, 7 C, 8 C, 9 C, 10 C, 12 C, 13 C,
15 C, 16 C, 18 C, 19 C, 21 C, 23 C, 25 C, 26 C, 28 C, 30 C, 33 C,
35 C, 37 C, 39 C, 41 C, 44 C, 46 C, 49 C, 51 C, 54 C, 56 C, 59 C,
61 C, 64 C, 67 C, 70 C, 72 C, 75 C, 78 C, 81 C, 84 C, 87 C, 90 C,
93 C, 96 C, 99 C, 102 C, 105 C, 108 C, 111 C, 115 C, 118 C, 121 C,
124 C,

VARIABLE WAVEFORM    \ holds waveform word execution token, xt
VARIABLE FREQ        \ holds time delay value between wave samples
VARIABLE INDEX       \ holds waveform index, 0-255 covers one waveform cycle
VARIABLE INDEX_INC   \ Value waveform index is incremented by
25 CONSTANT DAC0     \ GPIO pins for the two 8-bit DACs
26 CONSTANT DAC1
VARIABLE QUIT

: index_increment  ( -- ) \ Increment waveform index, set to 0 if > 255
               INDEX @ INDEX_INC @ + DUP
               254 >
                    IF   DROP 0 INDEX !
                    ELSE INDEX !
                    THEN
;
```

```
\ Three Waveform Generators, sends one indexed wave sample to DAC

: sine    ( -- )
          index_increment
          DAC0 Sinewave INDEX @ + C@ dacWrite
;

: square  ( -- )
          index_increment
          DAC0 INDEX @ 127 < IF 0  ELSE 255  THEN  dacWrite
;

: triangle  ( -- )
          index_increment
          DAC0 INDEX @ DUP 128 < IF 1 LSHIFT ELSE 255 swap - 1 LSHIFT THEN dacWrite
;

\ Cheap software delay using do loops, smaller time intervals than using MS
\ Run sine, square or triangle with cheap delay to control the frequency

: wave    WAVEFORM @ EXECUTE    FREQ @ 1 DO LOOP   ;  ( -- )


\ Operation commands from the keyboard to select waveform, change frequency, and quit

: keycommand
          KEY? IF KEY  \ true if key in input buffer, ready to read with KEY
          CASE
               [CHAR] 1 OF [']  sine       WAVEFORM ! ENDOF   \ 1  sinewave
               [CHAR] 2 OF [']  square     WAVEFORM ! ENDOF   \ 2   squarewave
               [CHAR] 3 OF [']  triangle   WAVEFORM ! ENDOF   \ 3  triangle
               [CHAR] > OF FREQ @ 1 -   2 MAX FREQ ! ENDOF   \ >   frequency up
               [CHAR] < OF FREQ @ 1 +   2 MAX FREQ ! ENDOF   \ <  frequency down
               [CHAR] h OF 1 INDEX_INC !             ENDOF   \ h  Index increment 1
               [CHAR] j OF 2 INDEX_INC !             ENDOF   \ j  Index increment 2
               [CHAR] k OF 4 INDEX_INC !             ENDOF   \ k  Index increment 4
               [CHAR] l OF 8 INDEX_INC !             ENDOF   \ l  Index increment 8
               [CHAR] q OF 1 QUIT !                  ENDOF   \ q   quit
                                                             \    ignore unused keys
          ENDCASE
          THEN
;

: mainwave
          0 QUIT !   ['] sine WAVEFORM !   10 FREQ !   1 INDEX_INC !

          ."     Key Commands -->  1 sine, 2 square, 3 triangle "
          CR ."     > frequency up, < frequency down, q QUIT "
          CR ."     h lowest, j 1-octave up , k 2-octaves up, l 3-octaves up"
          CR

          BEGIN
```

```
            keycommand
            wave
            QUIT @
        UNTIL

        CR ." Stopping Waves "
;
```

The code above starts off with three waveform generators. The sinewave generator simply steps through a 256 sample sinewave table of values between 0 to 255 that build one cycle of a sinewave. The squarewave generator outputs zero for the first 127 samples and 255 for the second half of the 255 samples. The triangle wave generator produces an up ramp for the first 127 samples and then calculates a down ramp for the second half of the samples by subtracting the up ramp values from 255.

Each of the three waveform generators, sine/square/triangle, increments a sample index running from 0 to 255 and then loads one indexed sample into one of the two ESP32 DACs using dacWrite.

The execution of these three waveform words uses an interesting Forth Language construct call the execution token, or xt. The code for each word in Forth is stored in memory. The memory address that holds the starting code for executing any particular Forth word is called its xt, or execution token.

The xt for any word is easily extracted by using the "tick" word.

```
' sine       (  -- xt for sine )
' square     ( -- xt for square )
' triangle   ( -- xt for triangle )
```

Note that, when used inside a colon word definition, the bracketed tick must be used instead [']. To start execution of a word from its xt, use the word EXECUTE (xt --). In the code above, the xt of the current waveform is stored at the address of the VARIABLE "WAVEFORM". To execute the xt of the current word stored in the WAVEFORM address, use this code found in the "wave" word:

```
WAVEFORM @ EXECUTE
```

To change the current waveform just load some other xt into the WAVEFORM address, as seen in the "keycommand" word.

```
'  triangle WAVEFORM !
```

Using this special pointer-type Forth construct, the waveforms produced can easily be changed at will as happens in the "keycommand" word.

Now, to actually produce an audio waveform signal, the words sine, square, or triangle must be repeated at a high speed which brings us to the topic of Timers.

TIMERS

One cycle of a 1000 Hz sinewave (about 2 octave above Middle C on a piano keyboard) will take one millisecond, or one thousand microseconds. Our sinewave table has 256 samples per cycle. This comes out to 3.9 microseconds per sample. To produce this 1kHz sinewave there must be a delay of 3.9 microseconds between each execution of the "sine" word.

As it turns out, however, the code for "sine" itself takes around 13 microseconds to complete making it impossible to produce a 1kHz tone with its paltry 3.9 microsecond delay between samples. If the word "sine" is simply repeated as fast as possible, what is produced at the DAC pin out is a very nice sinewave tone a bit above Middle C on the piano (256Hz).

So high frequencies present a problem given our forth code and the speed of the ESP32, but frequencies below Middle C can readily be produced by adding a delay between the single sample loads performed by sine, square, or triangle.

ESP32forth only has the MS time delay word. It sets up a time delay in millisecond increments, or 1000 microsecond increments, which is way too large for our purposes. Arduino ESP32 code does have a Microsecond( ) command which suggests that it may be possible to port it over in the Forth install file. A simple, quick and dirty, solution is to use a forth word in the form of an empty do-loop. In the above code this is built into the "wave" word and fed by the contents of a VARIABLE called FREQ. The time delay produced is simply set by how long our processor takes to count to FREQ from 1.

```
: delay 1 DO LOOP ;    ( n -- )
```

Another possibility that, at first, seemed to hold a lot of promise were the commands found in ESP32forth's "timers" vocabulary. These timer words deal directly with the ESP32's internal timer registers to set up an interrupt driven time interval. An interrupt system allows other code to run while the time interval is counted down. Since there are four available timers, it would also allow coding for more than one waveform to sound at the same time.

The timers code centers around the word "interval", easily set up with a single line:

```
timers  \  open timers vocabulary
: wave.start    ['] wave    FREQ @    0   interval  ;
```

"interval" requires three stack values -- the timer number (0 to 3), the time interval in microseconds (FREQ @), and the xt address of the word to be executed at the end of the time interval. Note that older descriptions talk about a "rerun" word that will retrigger the timer, usually placed at the end of the word triggered. The "rerun" word was demoted after retriggering was included inside the "interval" word and is no longer included in the vocabulary.

A problem with the "interval" setup immediately became apparent. Small microsecond time interval values would crash the system. Perhaps the "interval" timer was triggering "wave" a second time before the previous one was even finished. A problem that might have been prevented by the "rerun" construct. Larger time interval values worked fine but the timers method was abandoned when it was determined that it would not work with the smaller values needed for audio waveform sample spacings.

Here are some other useful words constructed with the timers vocabulary.

```
: wave.stop ( -- ) 0 0 enable! ;

: wave.resume ( -- ) 1 0 enable! ;

: wave.rate ( FREQ-value -- ) 0 0 alarm!  ;  \ ( new microsecond value, hi=0, timer#=0 )
```

Another useful word missing from the timers vocabulary is an isr-removal command to deconstruct the interrupt set up by "interval", which would have allowed us to re-run the "mainwave" command in our code after a "quit".

The problem still remains that we are only able to produce waveform frequencies at Middle C on the piano keyboard and lower. Is there any way to get all those other higher key frequencies? Turns out there is a way. By loading every other sample instead of all 256 in our waveform table we can get the C an octave above middle C. Similarly, by loading every forth sample in the table we can get two octaves above middle C, every eighth sample to get three octaves above middle C, and so on. What is lost in this process is the waveform's resolution. By sampling every other table value we are effectively only using a 128 sample table instead of the original 256 sample table. Continuing this process you will start seeing steps in the resulting wave which introduces a high frequency "edge" to the sinewave sound.

The code above provides a way to explore this method of skipping samples in the wave table. Checkout the VARIABLE called INDEX_INT (index interval). You can see it used in the "index-increment" word and set up in "keycommand".

FINAL DAC/TIMER CODE WORDS

The "keycommand" word in the code provides a way for users to control the audio waveform coming out the the DAC pin using the computer keyboard. If no key has been pressed recently this command is exited almost immediately when it sees that KEY? results in a zero value. Conversely, if a key press has been detected, KEY will read its value and present it to a CASE statement which decides what to do with it. Ten special keys are set up in the CASE statement to perform short modifications on the audio waveform such as changing the waveform or affecting its frequency in various ways.

Finally, the "mainwave" word pulls together all the previously defined words. It first sets up some initial conditions, prints out some user instructions, and then runs an infinite loop constantly running the "wave" command loading one waveform sample at a time into the DAC at the loop speed, along with checking the "keycommand" for any user input. The "q" key will quit out of the loop and stop the wave, but "mainwave" can be entered from the keyboard to restart it at any time.

Install File Revisions

The ESP32forth.ino install file is built using the C/C++ preprocessor **X-Macros**. It can list and build all the words found in the Forth Vocabulary. It can even port over Arduino and ESP32 library functions and build Forth words from them. In the process, parameters passed to and from C functions are turned into Forth stack operations.

A short description for adding words to the ESP32forth.ino install file can be found at the end of this Wordpress website https://esp32forth.wordpress.com/esp32forth-v-7-5-list-of-words/. Here is a description of the process.

Locate the macro called PLATFORM_OPCODE_LIST in ESP32forth.ino and add your words there, using the macros Y, X, YV, or XV.

1. The base macro: X

X("myword!", MY_WORD_BANG, c_function_to_call())
Three parts:

- "myword!" – the actual Forth name, as a string (can have any characters – !, @, :, etc.)
- MY_WORD_BANG – an internal C identifier, used nowhere else, just needs to be unique and contain only letters, numbers or underscore.
- c_function_to_call() – the C code that runs when the word executes

2. The shortcut: Y

To add a word named with only letters, numbers, or underscore use the shortened macro version - Y. Y(MY_WORD123, c_function_to_call()) \ No internal C identifier needed.

3. The list itself: #define PLATFORM_OPCODE_LIST

Found in the ESP32forth.ino install file is the PLATFORM_OPCODE_LIST, an X-Macro list of hundreds of X(...) and Y(...) calls chained with backslashes.

Here, several REQUIRED or OPTIONAL xxx_SUPPORT Word Groups are listed. Each Group then has its own list of words defined with X, Y, XV, and XY under the "# define"

macro.

You can place your own defined words in any of these word groups, even the ones within a specified Vocabulary using XV(...) and YV(...). However, note the Group "USER_WORDS \" at the top of the PLATFORM_OPCODE_LIST. This group was created for you, the USER. User words can go into its own group initiated with the line "#define USER_WORDS \" or they can go into an external file named userwords.h as specified earlier in ESP32forth.ino (// Hook to pull in words from optional userwords.h).

Don't leave out the backslashes. Each backslash glues the next line onto the same macro definition. The last entry in a group list does not need the backslash.

4. Stack Operations

Inside your code, you can read/write the Forth stack via tos (top of stack) and sp (rest of stack), both of type cell_t. You can also refer to elements on the stack with the following types of stack positional names:

```
n10 n9 n8 n7 n6 n5 n4 n3 n2 n1 n0 - Access stack as cell_t integer values
            c4 c3 c2 c1 c0 - Access stack as char* values
            b4 b3 b2 b1 b0 - Access stack as uint8_t* byte values
            a6 a5 a4 a3 a2 a1 a0 - Access stack as void* values
```

5. XV — the vocabulary-aware base macro

XV(internals, "'SYS", SYS, DUP; tos = (cell_t) &g_sys)
Four parts instead of three:

- internals — which vocabulary this word belongs to
- "'SYS" — the Forth name string
- SYS — the internal C identifier
- the code to run

Compare that to plain X, which had no vocabulary argument — those words land in the default vocabulary (FORTH), visible everywhere. XV lets a word be filed under a specific vocabulary instead, so it's only reachable when that vocabulary is active on the search stack.

6. YV — the shortcut version, same relationship as Y to X

YV(internals, YIELD, PARK; return rp)
YV(internals, EVALUATE1, DUP; float *tfp = fp; ...)

Notice word names YIELD and EVALUATE1 have only letters and numbers.
Not needed -> an internal C identifier seen in XV before the word code.

7. Putting it together – adding your own word

To add a word myword! that calls a C function:

1 Open arduino.template.ino (or ESP32forth.ino depending on version), find PLATFORM_OPCODE_LIST.

2 Add a line before the closing of some word group list, ending in \
X("myword!", MY_WORD_BANG, c_function_to_call()) \

3 Inside your code, you can read/write the Forth stack via tos (top of stack) and sp (rest of stack), both of type cell_t.

4 Recompile and upload. The X-Macro machinery automatically threads your new word into the dictionary table – no other file needs touching.

If your word name is a valid C identifier (letters/digits/underscore only), use Y(MYWORD, code) instead and skip writing some internal C string/id yourself.

Words globally visible → use X / Y.

Words existing inside a specific vocabulary (e.g. an assembler vocabulary, or internals for stuff not meant for normal users) → use XV / YV with that vocabulary name as the first argument, and make sure that vocabulary already exists in the vocabulary list.

8. Examples

To add custom words to trigger an LED, you simply locate PLATFORM_OPCODE_LIST within the install file ESP32forth.ino, create the USER_WORDS group with "define USER_WORDS \" and place your words in that word group. Or you can place the word definitions within a file named userwords.h within the same folder as ESP32forth.ino.

```
#define USER_WORDS \
Y(LED_ON, digitalWrite(2, HIGH)) \
Y(LED_OFF, digitalWrite(2, LOW)) \
```

When you hit compile in the Arduino IDE, the preprocessor duplicates those lines into the dictionary name arrays, the ID enums, and the inner interpreter engine

automatically.

Here is another example word that reads an analog sensor, maps its value, and leaves the result on the Forth data stack.

How it works:

```
       Read pin 34 and scale the output to a percentage (0 to 100)
                    34 100 ANALOG_SCALE .
```

What is entered inside your choice of a PLATFORM_OPCODE_LIST word block:

```
X("ANALOG_SCALE", ANALOG_SCALE_ID, int max_val = tos; tos = map(analogRead(sp[0]), 0,
4095, 0, max_val); sp++) \
```

Example Explanation

int max_val = tos;
The code grabs the input limit from the top of the stack and stores it safely in a local C variable.

analogRead(sp[0])
The code peeks at the next item down on the stack (sp[0]), which contains the GPIO pin number.

map(..., 0, 4095, 0, max_val)
The ESP32's 12-bit ADC value (0–4095) is scaled to a new range between 0 and your specified max_val.

tos = ...
The newly calculated value overwrites the old max_val register, putting the final answer at the very top of the stack.

sp++
Because two inputs (pin and max_val) were consumed but only one output was returned, the stack pointer increments by one to cleanly drop the pin variable.

Some Install File Additions

Here are some revisions to the ESP32forth.ino install file that I found necessary for my own applications. These are good examples of how this is accomplished, done with the help of Anthopic's AI Claude.

Serial2.begin

XV(serial, "Serial2.begin", SERIAL2_BEGIN, Serial2.begin(tos); DROP) \

replaced with (near line 71):
XV(serial, "Serial2.begin", SERIAL2_BEGIN, Serial2.begin(n3, n2, n1, n0); DROPn(4)) \

The Forth code can then use the full Serial2.begin command to initialize all the serial parameters needed for MIDI I/O.

```
MIDI_BAUD_RATE   SERIAL_8N1   MIDI_RX_PIN   MIDI_TX_PIN   Serial2.begin
                                    or
              31250   $C00001C   16   17   Serial2.begin
```

Timer Interrupt Remove

Added isr-remove! to interrupts.h
To completely stop or "deconstruct" timer number "t" interrupt, set up by the timer "interval" word. Allows quitting out of a word using interrupt timers and then restart it.

near line 216, before "onalarm" add:

```
: isr-remove! ( t -- ) t>nx timer_isr_callback_remove drop ;
```

near line 61, before timer_init_null add:

```
  YV(timers, timer_isr_callback_remove, \
     n0 = timer_isr_callback_remove((timer_group_t) n1, (timer_idx_t) n0); NIP) \
```

SPI Support

Added SPI support for the ESP32forth.

Exposes just enough of the Arduino SPI.h API to drive simple write-only SPI peripherals (like the MCP4921 DAC) from Forth. Chip-select is intentionally NOT handled here. It's just a GPIO, and can easily be handled using pinMode/digitalWrite.

Creates the **mcp-dac-spi.h** header file to include in the folder with ESP32forth.ino, plus some changes to ESP32forth.ino to create the SPI vocabulary. A more complete version can be found in **"The Great Book for ESP32forth"** by Marc Petremann.

mcp-dac-spi.h

```
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

/*
 * ESP32forth generic SPI bus support (add-on module)
 *
 * Exposes just enough of the Arduino SPI.h API to drive simple
 * write-only SPI peripherals (e.g. the MCP4921 DAC) from Forth.
 * Chip-select is intentionally NOT handled here -- it's just a GPIO,
 * and this build already exposes pinMode/digitalWrite for that.
 */

#include <SPI.h>

static void SpiBegin() {
  SPI.begin();  // use the board's default SCK/MISO/MOSI pins
}

static void SpiBeginPins(cell_t sck, cell_t miso, cell_t mosi, cell_t ss) {
  SPI.begin((int8_t) sck, (int8_t) miso, (int8_t) mosi, (int8_t) ss);
}

static cell_t SpiXfer16(cell_t hz, cell_t data) {
  SPI.beginTransaction(SPISettings((uint32_t) hz, MSBFIRST, SPI_MODE0));
  uint16_t result = SPI.transfer16((uint16_t) data);
```

```
  SPI.endTransaction();
  return (cell_t) result;
}

static cell_t SpiXfer8(cell_t hz, cell_t data) {
  SPI.beginTransaction(SPISettings((uint32_t) hz, MSBFIRST, SPI_MODE0));
  uint8_t result = SPI.transfer((uint8_t) data);
  SPI.endTransaction();
  return (cell_t) result;
}

#define OPTIONAL_SPI_VOCABULARY V(SPI)
#define OPTIONAL_SPI_SUPPORT \
  YV(SPI, SpiBegin, SpiBegin()) \
  YV(SPI, SpiBeginPins, SpiBeginPins(n3, n2, n1, n0); DROPn(4)) \
  YV(SPI, SpiXfer16, n0 = SpiXfer16(n1, n0); NIP) \
  YV(SPI, SpiXfer8, n0 = SpiXfer8(n1, n0); NIP)
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

changes to ESP32forth.ino

~~~~~~~~~

near line 151, before http_client_vocabulary

```
 OPTIONAL_SPI_VOCABULARY \
```

~~~~~~~~~

near lines 595 to 601, before http-client.h

```
// Hook to pull in optional generic SPI bus support.
# if __has_include("mcp-dac-spi.h")
#  include "mcp-dac-spi.h"
# else
#  define OPTIONAL_SPI_VOCABULARY
#  define OPTIONAL_SPI_SUPPORT
# endif
```

~~~~~~~~~

near line 656, before http_client_support

```
 OPTIONAL_SPI_SUPPORT \
```

~~~~~~~~

near lines 3176 to 3181, after DEFINED? rmt-builtins
~~~~~~~~

```
DEFINED? SPI-builtins [IF]
  vocabulary SPI   SPI definitions
  transfer SPI-builtins
  forth definitions
[THEN]
```

~~~~~~

Using it with the MCP4921

Wiring: SCK→DAC's SCK, MOSI→DAC's SDI, any free GPIO→DAC's CS (pulled high at idle). MISO is unused (the 4921 has no data output), so it's fine to leave unconnected or pass -1 if you use SpiBeginPins.

The MCP4921 wants one continuous 16-bit frame while CS is low: a 4-bit config nibble followed by your 12-bit value. 0x3000 | value gives you unbuffered input, 1× gain, active mode — the common default (use 0x7000 | value instead if you want buffered input).

```
forth
SPI definitions

5 constant CS               \ pick whatever GPIO you wired CS to
CS OUTPUT pinMode
CS HIGH digitalWrite        \ idle high

SpiBegin

: mcp4921! ( n -- )         \ n = 12-bit value, 0-4095
  $0FFF and $3000 or        ( cfg-nibble | value )
  CS LOW digitalWrite
  1000000 swap SpiXfer16 drop
  CS HIGH digitalWrite      ( CS rising edge latches the new value )
;

forth definitions
```

Then `2048 mcp4921!` sets mid-scale output. I started the clock at 1000000 (1 MHz) — the 4921 is rated up to 20 MHz, but I'd get it working reliably at 1 MHz first, then push the speed up once you've confirmed the wiring, especially if you're on a breadboard.
~~~~~~

Microsecond Delay

The Arduino has a millisecond delay, delay( ), which Forth retains. However it left out Arduino's microsecond delay which is easily added. Note that this may conflict with some timer interrupts. For a dirty and simpler microsecond delay just use an empty DO LOOP.

~~~~~~~~~~~~
near line 699, after MS_TICKS

```
  X("US-TICKS", US_TICKS, PUSH micros()) \
```

~~~~~~~~~~~~
near lines 2300 to 2302, after ms defined

```
DEFINED? us-ticks [IF]
  : us ( n -- ) us-ticks >r begin us-ticks r@ - over >= until rdrop drop ;
[THEN]
```

ESP32 Core Tasks

The ESP32 has two core processors that can run separate programs simultaneously. ESP32forth does have the multitasking words task, start-task, pause, but these just share the processor in a round-robin manner all in Core 1. WIFI and BlueTooth do use Core 0 for stack and system housekeeping. Other than this, Core 0 mostly sits idle with no way to pin a word task to it.

However, ESP32forth does have true FreeRTOS dual-core tasks — this is a separate rtos vocabulary, and it's real, direct ESP-IDF/FreeRTOS bindings, same style as the timers module:

```
xPortGetCoreID  ( -- core )   \ which core is executing right now (0 or 1)
xTaskCreatePinnedToCore ( ... -- handle )  \ launch a task pinned to a specific core
vTaskDelete     ( handle -- ) \ kill a task
```

xPortGetCoreID is trivial and useful right now — run it anywhere and you'll see 1. The problem here is that vTaskCreatPinnedToCore does not use a Forth xt execution token which would allow pinning an actual Forth word to Core 0, exactly how timer interrupts are set up with the word "interval". Instead, it uses a C function pointer, only allowing pinning to an already compiled C function.

To fix this, the word on-core is added to the rtos vocabulary inside ESP32forth.ino to be used like so:

```
'  some-word  0 on-core
```

Launches "some-word" in Core 0 and returns the task Handle. Note that if this is contained in a word definition, the tick ['] must be used instead.

~~~~~~~~~~~~

near line 823, after #ifndef ENABLE_FREERTOS_SUPPORT block ... #endif

```
#define CORE_TASK_STACK_CELLS 256

struct core_task_args {
  cell_t xt;
};

static void CoreTaskEntry(void *arg) {
  // NOTE: fstack/rstack/stack live on this task's own FreeRTOS-allocated
  // stack (set via the usStackDepth argument below), not on a shared
  // interrupt stack, so a generous cell count here is safe.
  struct core_task_args *args = (struct core_task_args *) arg;
  cell_t code[2];
  code[0] = args->xt;
  code[1] = g_sys->YIELD_XT;
  cell_t fstack[CORE_TASK_STACK_CELLS];
  cell_t rstack[CORE_TASK_STACK_CELLS];
  cell_t stack[CORE_TASK_STACK_CELLS];
  stack[0] = 0;
  cell_t *rp = rstack;
  *++rp = (cell_t) code;
  *++rp = (cell_t) (fstack + 1);
  *++rp = (cell_t) (stack + 1);
  forth_run(rp);
  free(args);
  vTaskDelete(NULL);  // a FreeRTOS task function must never return
}

static cell_t CoreTaskStart(cell_t xt, cell_t core) {
  // NOTE: Leaks memory if task creation fails to later free args on exit,
  // matching the existing leak-tolerant style of this file's other
  // wrapper helpers (EspIntrAlloc, GpioIsrHandlerAdd, etc).
  struct core_task_args *args =
      (struct core_task_args *) malloc(sizeof(struct core_task_args));
  args->xt = xt;
  TaskHandle_t handle = NULL;
  BaseType_t ok = xTaskCreatePinnedToCore(
```
~~~~~~~~~~~~

```
      CoreTaskEntry, "forth-core-task", 16384, args, 1, &handle,
      (BaseType_t) core);
  if (ok != pdPASS) {
    free(args);
    return 0;
  }
  return (cell_t) handle;
}
```

~~~~~~~~~~~~~~~
near line 827, after   YV(rtos, xPortGetCoreID, PUSH xPortGetCoreID()) \

```
  YV(rtos, CoreTaskStart, n0 = CoreTaskStart(n1, n0); NIP)
```

~~~~~~~~~~~~~~~
near line 1917, after   transfer rtos-builtins

```
: on-core ( xt core -- handle ) CoreTaskStart ;
```

Random

Arduino's random( ) on ESP32 is a thin wrapper *around* esp_random( ), not a separate algorithm. esp.random( ) itself is a genuine **hardware** RNG — it draws from analog noise sources (thermal/RF noise) on the chip, not a software algorithm at all. That means, unlike classic Arduino, you don't need randomSeed( ) for good randomness quality here — there's no seed to set in the first place, and calling random( ) repeatedly won't produce a reproducible sequence the way it would on AVR.

Word	Stack effect	Maps to
random	(n -- r)	random(long) — returns 0 to n-1
random-range	(lo hi -- r)	random(long,long) — returns lo to hi-1
esp_random	(-- r)	esp_random() — raw 32-bit value, unbounded

```
255 random           \ 0-254, handy for an 8-bit LEDC duty value

100 200 random-range \ 100-199

esp_random            \ full 32-bit hardware value, no scaling
```

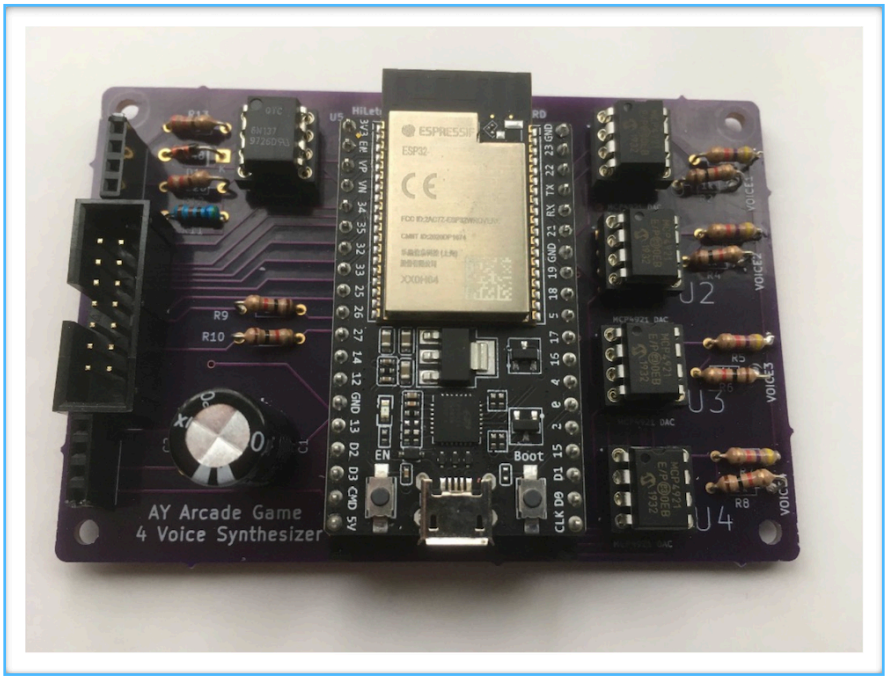
near line 632 among the "REQUIRED" lines, inside `#define PLATFORM_OPCODE_LIST \`

```
  REQUIRED_ARDUINO_RANDOM_SUPPORT \
```

near line 733, under section `#define REQUIRED_ARDUINO_GPIO_SUPPORT \`

```
#define REQUIRED_ARDUINO_RANDOM_SUPPORT \
  X("random", RANDOM, n0 = (cell_t) random((long) n0)) \
  X("random-range", RANDOM_RANGE, n0 = (cell_t) random((long) n1, (long) n0); NIP) \
  Y(esp_random, PUSH esp_random())
```

AY SYNTH BOARD



This section applies to a specialized circuit board created by John Talbert (https://jtalbert.xyz/esp32/). This is a four-voice programmable synthesizer built with the ESP32 DevKit. The four voices are built from signals generated on 4 ESP32 output pins connected to 4 multiplying MCP4921 DAC chips used for volume control. Two of the pins can be programmed as 8-bit DACs to produce any type of waveform. All four can be programmed to produce square waves and PWM pulse waves. If the pulse width is fed random values the pulse wave becomes pitched noise.

Here are a couple Forth Programs created for this board to demonstrate what a more extensive application would look like in Forth Code.

Random Voice Program

The first example Forth Program creates three audio voices with random pitches and note durations. The volume envelopes all have sudden Attacks and timed Releases dropping to zero. Four potentiometers and two pushbutton switches are programmed to control the sounds in real time. An audio file example is included on the website.

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ RANDOM PULSEWAVE VOICES
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ Using the AY Synthesizer Board for the ESP32 designed by John Talbert
\ See website   https://jtalbert.xyz/esp32/
\ Programmed in Forth Programming Language
\ After loading the ESP32forth.ino system onto the ESP32

\ Uses 4 MCP4921 Multiplier DACs to set the 4 voice volumes
\ Loaded using the SPI vocabulary from an mcp-dac-spi.h file
\ 4 GPIO Pin Voices with pitches controlled using the LEDC vocabulary

\ Three voices programmed with semi-random pitch and duration
\ All voice volume envelopes use an immediate attack and timed release

\ POT1 sets the range of random note durations
\ POT2 sets the range of random pitch frequencies
\ POT3 sets the base frequency of random pitches
\ POT4 sets tempo
\ SWITCH1
\ SWITCH2 selects between equal-tempered musical pitches and any frequency


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ ESP32 pin assignments


\ Serial Lines to the MCP4921 Multiplier DACs that use SPI

22 CONSTANT SCK
21 CONSTANT MOSI  \ MISO not used , not on MCP4921

\ Voice Pins Connected to MCP4921 Multiplier DACs

27 CONSTANT V1  \ 25 DAC1 on later boards
14 CONSTANT V2   \ 26 DAC2 on later boards
12 CONSTANT V3
13 CONSTANT V4

V1 OUTPUT pinMode  V2 OUTPUT pinMode  V3 OUTPUT pinMode  V4 OUTPUT pinMode

0 VALUE V1_DUR  \ voice duration
0 VALUE V2_DUR
0 VALUE V3_DUR
0 VALUE V4_DUR
0 VALUE V1_CNT   \ duration countdown counter
```

```
0 VALUE V2_CNT
0 VALUE V3_CNT
0 VALUE V4_CNT
0 VALUE V1_ENV   \ voice amplitude envelope
0 VALUE V2_ENV
0 VALUE V3_ENV
0 VALUE V4_ENV

: Vinit    0 TO V1_DUR  0 TO V2_DUR  0 TO V3_DUR  0 TO V4_DUR
                0 TO V1_CNT  0 TO V2_CNT  0 TO V3_CNT  0 TO V4_CNT
                0 TO V1_ENV  0 TO V2_ENV  0 TO V3_ENV  0 TO V4_ENV
;

\ Chip Select pins to MCP4921 Multiplier DACs

23 CONSTANT CS_V1
19 CONSTANT CS_V2
18 CONSTANT CS_V3
4  CONSTANT CS_V4

CS_V1 OUTPUT pinMode  CS_V2 OUTPUT pinMode  CS_V3 OUTPUT pinMode  CS_V4 OUTPUT pinMode

CS_V1 HIGH digitalWrite
CS_V2 HIGH digitalWrite
CS_V3 HIGH digitalWrite
CS_V4 HIGH digitalWrite

\ ADC pins connected to potentiometers

36 CONSTANT POT1 \ ESP32 ADC resolution is 4095, 12-bit
39 CONSTANT POT2
34 CONSTANT POT3
35 CONSTANT POT4

POT1 INPUT pinMode  POT2 INPUT pinMode  POT3 INPUT pinMode  POT4 INPUT pinMode


33 CONSTANT SWITCH1
32 CONSTANT SWITCH2

SWITCH1 INPUT pinMode   SWITCH2 INPUT pinMode

interrupts                              \ open interrupts vocabulary
SWITCH1 gpio_pullup_en drop \ enable pullups for each switch
SWITCH2 gpio_pullup_en drop
forth


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SPI
\ Used for 4 MCP4921 DAC chips controlling volume on 4 GPIO voices

\ The MCP4921 wants one continuous 16-bit frame while CS is low: a 4-bit config nibble
\ followed by your 12-bit value. 0x3000 | value gives you unbuffered input , 1× gain ,
\ active mode — the common default , use 0x7000 | value instead if you want buffered
\ input
```

```
: mcp! ( cs value -- )
           OVER                              \ cs value cs
           LOW digitalWrite                  \ Chip Select LOW
           $0FFF and $3000 or          \ cfg nibble high bits | 12-bit value 0-4095
           2000000 swap SpiXfer16 drop \ send value at 2MHz clock
           HIGH digitalWrite                 \ CS HIGH , edge latches new value , idle high
;

 SCK -1 MOSI -1 SpiBeginPins  \ Begin SPI with assigned pins, not the default

: mcp-initialize  ( -- )  \ set SPI GPIO pins , turn MCP4921's fully on

           CS_V1 4095 mcp!
           CS_V2 4095 mcp!
           CS_V3 4095 mcp!
           CS_V4 4095 mcp!
;


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

LEDC
\ ESP32 Hardware timers producing Square and Pulse waves on GPIO pins, CPU free.
\ Works well only with classic ESP32 (DEVKIT or WROOM) not S2, S3, C3, C6, H2 versions.


\ Initialize 4 voices to some frequency, 8-bit PWM resolution, and 50% duty cycle
: voices-initialize
                       V1 440 8 ledcAttach drop    V1 128 ledcWrite
                       V2 540 8 ledcAttach drop    V2 128 ledcWrite
                       V3 640 8 ledcAttach drop    V3 128 ledcWrite
                       V4 740 8 ledcAttach drop    V4 128 ledcWrite
;

\ Change Voice pin frequency in Hz, SquareWave Output, 0 freq = silence
: voice-freq! ( pin , freq -- )   ledcWriteTone drop ;

\ ledcWrite ( pin , duty -- )  Sets the pin PWM duty cycle , 0-255 with 8-bit res.

: voice-pwm! ( pin , duty -- ) ledcWrite ;

\ ledcChangeFrequency ( pin freq resolution -- ok )   Does not change PWM duty
: voice-freqx! ( pin , freq -- )   8 ledcChangeFrequency drop ;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 : init    ( -- )     mcp-initialize  voices-initialize ;
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



: ROM-TABLE  ( "name" -- )      \ defining word: Read Only Table or Array
    CREATE                       \ the data itself (via ,) sets the size
    DOES>  ( i addr -- value )     \ addr is the address of "name"
    SWAP CELLS + @               \ index -> address -> fetch, all in one
;

ROM-TABLE MIDI_Freq    \ create the Table MIDI_Freq, 128 NoteOn key frequencies
```

```
\  C      C#     D      D#      E      F      F#     G      G#     A      A#     B
   8 ,    9 ,    9 ,   10 ,   10 ,  11  ,  12  ,  12  ,  13  ,  14  ,  15  ,  15  ,
  16 ,   17 ,   18 ,   19 ,   21 ,  22  ,  23  ,  24  ,  26  ,  28  ,  29  ,  31  , \ C0
  33 ,   35 ,   37 ,   39 ,   41 ,  44  ,  46  ,  49  ,  52  ,  55  ,  58  ,  62  , \ C1
  65 ,   69 ,   73 ,   78 ,   82 ,  87  ,  92  ,  98  , 104  , 110  , 117  , 123  , \ C2
 131 ,  139 ,  147 ,  156 ,  165 , 175  , 185  , 196  , 208  , 220  , 233  , 247  , \ C3
 262 ,  277 ,  294 ,  311 ,  330 , 349  , 370  , 392  , 415  , 440  , 466  , 494  , \ C4
 523 ,  554 ,  587 ,  622 ,  659 , 698  , 740  , 784  , 831  , 880  , 932  , 988  , \ C5
1047 , 1109 , 1175 , 1245 , 1319 , 1397 , 1480 , 1568 , 1661 , 1760 , 1865 , 1976 , \ C6
2093 , 2217 , 2349 , 2489 , 2637 , 2794 , 2960 , 3136 , 3322 , 3520 , 3729 , 3951 , \ C7
4186 , 4435 , 4699 , 4978 , 5274 , 5588 , 5920 , 6272 , 6645 , 7040 , 7459 , 7902 , \ C8
8372 , 8870 , 9397 , 9956 , 10548 , 11175 , 11840 , 12544 ,

\  USAGE  69  MIDI_Freq .    -> 440
\ Grand Piano range is A0-28Hz to C8-4186Hz
\ Frequencies lower than 28Hz will probably not work with ledc

ROM-TABLE Vol32  \ 32-step logarithmic volume, ~1.74 dB/step, 0-4095
           0 ,   10 ,   12 ,   15 ,   18 ,   22 ,   27 ,   33 ,
          41 ,   50 ,   61 ,   74 ,   91 ,  111 ,  136 ,  166 ,
         202 ,  247 ,  302 ,  369 ,  451 ,  551 ,  674 ,  823 ,
        1006 , 1230 , 1503 , 1836 , 2244 , 2742 , 3351 , 4095 ,


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


\ Random Voice Build, 1+ RANDOM -> avoids 0 RANDOM which will crash


: getDUR ( -- n )   \ get a random note duration countdown rate from POT1
           POT1 analogRead 4 RSHIFT 1+ RANDOM 20 + \ 20 to 275
;

: getFreq ( -- freq )       \ get a random pitch value from pots 2 and 3
           POT3 analogRead  20 +                   \ base frequency, 20 to 4116
           POT2 analogRead 3 RSHIFT 1+ RANDOM      \ choose frequency above BASE, 1-512
           +     5588 MIN                          \ base frequency plus random frequency
;

: getNote ( -- freq )  \ get equal-tempered music keyboard note from MIDI_Freq table
           POT3 analogRead 5 RSHIFT 20 +           \ base index to table 20-148
           POT2 analogRead 5 RSHIFT 1+ RANDOM      \ random index above BASE 1-128
           + 115 MIN                               \ base index plus random, 21 to 115
           MIDI_Freq              \ select indexed frequency from MIDI_Freq table
;


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ Controllers


: pitchSelect ( -- freq )   \ use SWITCH2 to select music notes or free frequencies
           SWITCH2 digitalRead IF getFreq ELSE getNote THEN
;


: tempo ( -- )                               \ use set tempo
```

```
           POT4 analogRead 1 RSHIFT US       \ 0 to 2048
;

: pulseWidth ( -- )                          \ use POT4 to set duty cycle
           POT4 analogRead 5 RSHIFT 20 +     \ 20 to 147
           DUP V1 SWAP voice-pwm!
           DUP V2 SWAP voice-pwm!
           DUP V3 SWAP voice-pwm!
                V4 SWAP voice-pwm!
;


 \ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 \ Playback


 : V1_countdown ( -- )
                V1_CNT  IF
                                V1_CNT 1- TO V1_CNT    \ countdown duration counter
                           ELSE                        \ countdown reached zero
                                V1_DUR TO V1_CNT       \ reset duration counter
                                V1_ENV 1- TO V1_ENV    \ voice release envelope lowered
                                CS_V1 V1_ENV Vol32 mcp!      \ load new envelope release
volume
                           THEN
 ;

  : V2_countdown ( -- )
                V2_CNT  IF
                                V2_CNT 1- TO V2_CNT    \ countdown duration counter
                           ELSE                        \ countdown reached zero
                                V2_DUR TO V2_CNT       \ reset duration counter
                                V2_ENV 1- TO V2_ENV    \ voice release envelope lowered
                                CS_V2 V2_ENV Vol32 mcp!       \ load new envelope release
volume
                           THEN
 ;

 : V3_countdown ( -- )
                V3_CNT  IF
                                V3_CNT 1- TO V3_CNT    \ countdown duration counter
                           ELSE                        \ countdown reached zero
                                V3_DUR TO V3_CNT       \ reset duration counter
                                V3_ENV 1- TO V3_ENV    \ voice release envelope lowered
                                CS_V3 V3_ENV Vol32 mcp!       \ load new envelope release
volume
                           THEN
 ;

  : V4_countdown ( -- )
                V4_CNT  IF
                                V4_CNT 1- TO V4_CNT    \ countdown duration counter
                           ELSE                        \ countdown reached zero
                                V4_DUR TO V4_CNT       \ reset duration counter
                                V4_ENV 1- TO V4_ENV    \ voice release envelope lowered
                                CS_V4 V4_ENV Vol32 mcp! \ load new envelope release volume
                           THEN
 ;
```

```
 : V1_reset ( -- )
                V1 pitchSelect voice-freqx!       \ get new random pitch
                getDUR DUP                        \ get new random duration
                TO V1_DUR TO V1_CNT               \ set new duration rate and duration
counter
                31 TO V1_ENV                      \ set volume envelope to max
                CS_V1 V1_ENV Vol32 mcp!           \ fast envelope attack to maximum volume
 ;

 : V2_reset ( -- )
                V2 pitchSelect voice-freqx!       \ get new random pitch
                getDUR DUP                        \ get new random duration
                TO V2_DUR TO V2_CNT               \ set new duration rate and duration
counter
                31 TO V2_ENV                      \ set volume envelope to max
                CS_V2 V2_ENV Vol32 mcp!           \ fast envelope attack to maximum volume
 ;

 : V3_reset ( -- )
                V3 pitchSelect voice-freqx!       \ get new random pitch
                getDUR DUP                        \ get new random duration
                TO V3_DUR TO V3_CNT               \ set new duration rate and duration
counter
                31 TO V3_ENV                      \ set volume envelope to max
                CS_V3 V3_ENV Vol32 mcp!           \ fast envelope attack to maximum volume
 ;

 : V4_reset ( -- )
                V4 pitchSelect voice-freqx!       \ get new random pitch
                getDUR DUP                        \ get new random duration
                TO V4_DUR TO V4_CNT               \ set new duration rate and duration
counter
                31 TO V4_ENV                      \ set volume envelope to max
                CS_V4 V4_ENV Vol32 mcp!           \ fast envelope attack to maximum volume
 ;

 \ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 \ V3 volume envelope is having problems, may be a pin18 conflict. We are using it
 \ as a V3 chip select, but it is also the default SPI clock though not set for that


 : playRandomVoices ( -- )
           init  Vinit
           ." START PLAY    "
           BEGIN
                 tempo
                   V1_ENV IF V1_countdown ELSE V1_reset THEN
                   V2_ENV IF V2_countdown ELSE V2_reset THEN
                 \ V3_ENV IF V3_countdown ELSE V3_reset THEN
                   V4_ENV IF V4_countdown ELSE V4_reset THEN

           KEY? UNTIL
           KEY DROP
           ." END PLAY  " CR
 ;
 \ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

AY Arcade Playback

Among its many applications, this Synth Board can be programmed to simulate an AY Arcade Game Sound chip from the 70's and play back old AY sound files still available on the web.

The Forth Program example shown here recreates this Arcade Game Sound chip. A short Python program was written to translate an YM sound file to a large Forth table of AY frequency and amplitude register data for three voices. Once this Forth table of AY data is loaded into the ESP32Forth RAM memory space, the Forth program shown below can play the Song File. Playback involves simply loading consecutive Frames of these frequency and amplitude registers every 50 milliseconds. Four potentiometers and two pushbutton switches are set up to control the playback in real time.

```
\ AY File Playback
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ This is a simulation of the AY-3-8910 Arcade Game Sound Chip
\ that plays YM files.  A frame of 14 8-bit registers from the YM song file
\ are loaded into the chip simulator every 50 ms to playback the song file.
\
\ Actually, here, only 10 of the 14 chip registers are used.
\ The on-chip noise generators and envelopes are not used in common practice.
\
\ Uses an AY Synthesizer Board for the ESP32 designed by John Talbert
\ See website   https://jtalbert.xyz/esp32/
\ To be programmed in Forth Programming Language
\ After loading the ESP32forth.ino system onto the ESP32
\
\ Four onboard MCP4921 Multiplier DACs control the 4 voice volumes.
\ Loaded using the SPI vocabulary from an mcp-dac-spi.h file,
\ Four GPIO Pin Voices with pitches controlled using the LEDC vocabulary
\
\ Three voices are used here, loaded from an AY SONG file with pitch and volume data.
\
\ POT1           sets position in Song
\ POT2           Tune song frequencies
\ POT3
\ POT4           sets Song tempo
\ SWITCH1        pauses playback
\ SWITCH2        loads pot1 song position ( rewind )
\ any KEY        stops playback

\ ESP32 pin assignments and some data variables

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

: NOT ( flag -- !flag )  0= ;
```

```
62500 VALUE VTUNE  \ Master Clock for voice frequencies

0 VALUE F1  \ previous frame frequency
0 VALUE F2
0 VALUE F3
0 VALUE A1  \ previous frame amp
0 VALUE A2
0 VALUE A3

\ Serial Lines to MCP4921 Multiplier DACs that use SPI Serial

22 CONSTANT SCK
21 CONSTANT MOSI  \ MISO not used , not on MCP4921

\ Voice Pins Connected to MCP4921 Multiplier DACs

27 CONSTANT V1   \ 25 DAC1 on later revised boards
14 CONSTANT V2   \ 26 DAC2 on later revised boards
12 CONSTANT V3
13 CONSTANT V4

V1 OUTPUT pinMode  V2 OUTPUT pinMode  V3 OUTPUT pinMode  V4 OUTPUT pinMode


\ Chip Select pins to MCP4921 Multiplier DACs

23 CONSTANT CS_V1
19 CONSTANT CS_V2
18 CONSTANT CS_V3
4  CONSTANT CS_V4

CS_V1 OUTPUT pinMode  CS_V2 OUTPUT pinMode  CS_V3 OUTPUT pinMode  CS_V4 OUTPUT pinMode

CS_V1 HIGH digitalWrite
CS_V2 HIGH digitalWrite
CS_V3 HIGH digitalWrite
CS_V4 HIGH digitalWrite

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

36 CONSTANT POT1        \ ESP32 ADC resolution is 4095, 12-bit
39 CONSTANT POT2
34 CONSTANT POT3
35 CONSTANT POT4

POT1 INPUT pinMode  POT2 INPUT pinMode  POT3 INPUT pinMode  POT4 INPUT pinMode

33 CONSTANT SWITCH1
32 CONSTANT SWITCH2

SWITCH1 INPUT pinMode   SWITCH2 INPUT pinMode

interrupts                  \ open interrupts vocabulary
SWITCH1 gpio_pullup_en drop \ enable pullups for each switch
SWITCH2 gpio_pullup_en drop
forth
```

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SPI         \ open SPI vocabulary

\ Used for 4 MCP4921 DAC chips controlling volume on 4 GPIO voices

\ The MCP4921 wants one continuous 16-bit frame while CS is low: a 4-bit config nibble
\ followed by your 12-bit value. 0x3000 | value gives you unbuffered input , 1× gain ,
\ active mode — the common default , use 0x7000 | value instead if you want buffered
\ input

: mcp! ( cs value -- )
           OVER                        \ cs value cs
           LOW digitalWrite            \ Chip Select LOW
           $0FFF and $3000 or          \ cfg nibble high bits | 12-bit value 0-4095
           2000000 swap SpiXfer16 drop \ send value at 2MHz clock
           HIGH digitalWrite           \ CS HIGH , edge latches new value , idle high
;

: mcp-initialize  ( -- )  \ set SPI GPIO pins , turn MCP4921's fully on
           SCK -1 MOSI -1 SpiBeginPins
           CS_V1 4095 mcp!
           CS_V2 4095 mcp!
           CS_V3 4095 mcp!
           CS_V4 4095 mcp!
;


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

LEDC
\ ESP32 Hardware timers producing Square and Pulse waves on GPIO pins, CPU free.
\ Works well only with classic ESP32 (DEVKIT or WROOM) not S2, S3, C3, C6, H2 versions.

\ Initialize 4 voices to some frequency, 8-bit PWM resolution, and 50% duty cycle
: voices-initialize
                      V1 440 8 ledcAttach drop    V1 128 ledcWrite
                      V2 540 8 ledcAttach drop    V2 128 ledcWrite
                      V3 640 8 ledcAttach drop    V3 128 ledcWrite
                      V4 740 8 ledcAttach drop    V4 128 ledcWrite
;

\ Change Voice pin frequency in Hz , SquareWave Output , 0 freq = silence
: voice-freq! ( pin , freq -- )   ledcWriteTone drop ;

\  ledcWrite ( pin , duty -- )  Sets the pin PWM duty cycle , 0-255 with 8-bit res.

: voice-pwm! ( pin , duty -- ) ledcWrite ;

\ ledcChangeFrequency ( pin freq resolution -- ok )   Does not change PWM duty
: voice-freqx! ( pin , freq -- )   8 ledcChangeFrequency drop ;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 : init    ( -- )     mcp-initialize  voices-initialize ;
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
: ROM-TABLE  ( "name" -- )      \ defining word: Read Only Table or Array
    CREATE                        \ the data itself (via ,) sets the size
    DOES>  ( i addr -- value )    \ addr is the address of "name"
    SWAP CELLS + @                \ index -> address -> fetch, all in one
;


ROM-TABLE AY_Vol  \ 4-bit AY synth volume to 12-bit logarithmic volume
           0 , 10 , 15 , 24 , 36 , 56 , 86 , 132 ,
           202 , 311 , 478 , 734 , 1128 , 1734 , 2665 , 4095 ,


\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
\ YM File Playback
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ Many old arcade YM song files are still available on the web.
\ A Forth Code Table is created from one of these YM song files
\ using the python program YMReg_ForthBuild.py

\ For example:  the file 3dfight.YM5 from a website archive was
\ run through the python program YMReg_ForthBuild.py to create
\ the Forth byte data file 3dfight.fs   Starts and ends as follows:

\ CREATE 3dfight-song
\ $7D C, $02 C, $FD C, $00 C, $3E C, $01 C, $38 C, $0A C, $0A C, $08 C,
\ ...
\ $3E C, $01 C, $9F C, $00 C, $7D C, $02 C, $3F C, $00 C, $00 C, $00 C,
\ : 3dfight-frames 984 ;

\ Each frame consists of the following 10 AY Synth register values. Use 0-9.
\ r0   0    Voice1 8-bit fine tune frequency
\ r1   1    Voice1 4-bit course tune frequency
\ r2   2    Voice2 8-bit fine tune frequency
\ r3   3    Voice2 4-bit course tune frequency
\ r4   4    Voice3 8-bit fine tune frequency
\ r5   5    Voice3 4-bit course tune frequency
\ r7   6    8-bit Voice/Noise Enable
\ r8   7    Voice1 4-bit Volume
\ r9   8    Voice2 4-bit Volume
\ r10  9    Voice3 4-bit Volume
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ Initial Song File Setup
\ Here I use a Forth Code Song File loaded from the Terminal App,
\ May eventually try loading it from an SD card.

\ Step 1, load your Forth song file created with YMReg_ForthBuild.py into RAM
\ Step 2, replace "3dfight" with your own song filename here, in the following 2 words:

3dfight-frames CONSTANT TOTALFRAMES

: SONG ( frame register -- value )   \ creat a 2-dimension song array accessed by frames
  >R  10 *  3dfight-song +  R> +  C@
;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
\ Frame Register Loading

0 VALUE FRAME                     \ one set of frequency and volume registers

: voice1Freq!  ( -- )
           FRAME
           DUP 0 SONG             \ get 8-bit fine tune
           SWAP 1 SONG            \ get 4-bit course tune
           8 LSHIFT OR            \ form 12-bit total tune
           DUP F1 =
     IF
           DROP                   \ If frequency not changed do nothing
     ELSE
           DUP TO F1
           1 MAX                  \ AY hardware convention: 0 is treated as 1
           VTUNE SWAP /           \ AY master clock divided by tune --> Hz
           V1 SWAP voice-freq! \ use ledc vocab to load freq
     THEN
;

: voice2Freq!  ( -- )
           FRAME
           DUP 2 SONG             \ get 8-bit fine tune
           SWAP 3 SONG            \ get 4-bit course tune
           8 LSHIFT OR            \ form 12-bit total tune
           DUP F2 =
     IF
           DROP                   \ If frequency not changed do nothing
     ELSE
           DUP TO F2
           1 MAX                  \ AY hardware convention: 0 is treated as 1
           VTUNE SWAP /           \ AY master clock divided by tune --> Hz

           V2 SWAP voice-freq! \ use ledc vocab to load freq
     THEN
;

: voice3Freq!  ( -- )
           FRAME
           DUP 4 SONG             \ get 8-bit fine tune
           SWAP 5 SONG            \ get 4-bit course tune
           8 LSHIFT OR            \ form 12-bit total tune
           DUP F3 =
     IF
           DROP                   \ If frequency not changed do nothing
     ELSE
           DUP TO F3
           1 MAX                  \ AY hardware convention: 0 is treated as 1
           VTUNE SWAP /           \ AY master clock divided by tune --> Hz
           V3 SWAP voice-freq! \ use ledc vocab to load freq
     THEN
;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

\ r7 (SONG index 6) is the AY mixer/enable register. Bits 0-2 are the
\ per-voice TONE enable bits for voice A/B/C (bits 3-5 are noise enable,
```

```
\ left alone -- this board doesn't use the noise generator). These bits
\ are ACTIVE LOW on real AY hardware: 0 = tone on, 1 = tone muted.

: toneEnabled? ( bit# -- flag )   \ true if this voice's tone bit in r7 is active
     FRAME 6 SONG SWAP RSHIFT 1 AND 0=
;

: voice1Vol!  ( -- )
           0 toneEnabled?
           IF
                FRAME 7 SONG $0F AND   \ get 4-bit volume

                AY_Vol                          \ get log 12-bit volume from Table
           ELSE
                0                               \ mixer says this voice's tone is off
           THEN
                DUP A1 =
           IF
                DROP                       \ If volume is the same don't reload it
           ELSE
                DUP TO A1
                CS_V1 SWAP mcp!            \ using SPI load into DAC chip
           THEN
;

: voice2Vol!  ( -- )
           1 toneEnabled?
           IF
                FRAME 8 SONG $0F AND   \ get 4-bit volume
                AY_Vol                          \ get log 12-bit volume from Table
           ELSE
                0                               \ mixer says this voice's tone is off
           THEN
                DUP A2 =
           IF
                DROP                       \ If volume is the same don't reload it
           ELSE
                DUP TO A2
                CS_V2 SWAP mcp!            \ using SPI load into DAC chip
           THEN
;

: voice3Vol!  ( -- )
           2 toneEnabled?
           IF
                FRAME 9 SONG $0F AND   \ get 4-bit volume
                AY_Vol                          \ get log 12-bit volume from Table
           ELSE
                0                               \ mixer says this voice's tone is off
           THEN
                DUP A3 =
           IF
                DROP                       \ If volume is the same don't reload it
           ELSE
                DUP TO A3
                CS_V3 SWAP mcp!            \ using SPI load into DAC chip
           THEN
;
```

```
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
: ay!  ( -- )
          voice1Freq!  voice1Vol!
          voice2Freq!  voice2Vol!
          voice3Freq!  voice3Vol!
;
\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~



\ Playback Controllers

: tempo ( -- )         \ Adjust tempo with POT4
          POT4 analogRead 5 RSHIFT 20 + MS  \ 20-137, normal is 50MS
;

: pausePlay ( -- )  BEGIN SWITCH1 digitalRead UNTIL ;

: map ( pot max -- scaled )  4095 */ ;   \ map pot value 0-4095, to 0-max

: rewind ( -- )  \  when switch2 is pressed rewind to SONG position set by POT1
     SWITCH2 digitalRead NOT IF POT1 analogRead TOTALFRAMES map TO FRAME THEN
;

: tune ( -- )    \ Master Clock 1,000,000 divided by 4 to 36,  Normal is 16
     1000000 POT2 analogRead 7 RSHIFT 4 +  /  TO VTUNE
;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


\ Song Playback

: frameIncrement ( -- )  \ at SONG End wait 2 seconds then start over
          FRAME 1+  DUP
          TOTALFRAMES >
               IF          DROP   ." End of Song " CR    2000 MS    ." Begin Song -- "
                           0
               THEN
          TO FRAME
;


: songplay ( -- )
          init  ." Song Begin      "  0 TO FRAME
          BEGIN
               ay!
               frameIncrement
               pausePlay  rewind  tune     \ check controllers
               tempo                       \ set play speed
          KEY?
          UNTIL
               KEY DROP
               ." Song End"  CR
;

\ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

A playback sound example using the above code is included on the website along with the python code and an archive of many old YM soundfiles for you to use.

Some Closing Thoughts

Why use Forth on the ESP32 instead of C on the Arduino IDE?

The core difference: compile-and-flash vs. talk-to-your-board

With C on the Arduino IDE, the loop is: write code → compile on your computer → upload the whole program → it runs (or doesn't) → if something's wrong, go back to step one. Even a tiny change means a full recompile-and-flash cycle, which can take anywhere from a few seconds to uncomfortably long.

Forth is interactive. Once it's running on a microcontroller, you have a live command line *on the device itself*. You can type a word (Forth's term for a function), press enter, and it executes immediately — no compiling, no uploading. You can test a single line of logic, check a sensor reading, or twiddle a pin in real time, right at the hardware.

Why that matters

1 **Immediate feedback loop.** You can poke at hardware interactively and see what happens immediately. That's a much gentler way to learn what a register, a pin, or a timer actually *does* than reading a datasheet and hoping your C code is right before you flash it.

2 **Build up from tiny, testable pieces.** In Forth you define small words (: blink-led ... ;) and test each one at the prompt before combining them into bigger words. It naturally teaches incremental, bottom-up thinking — you're never debugging 200 lines at once.

3 **You can inspect the running system.** Since the interpreter is live, you can ask the device "what's the value of this variable right now?" without adding Serial.print statements and reflashing.

4 **Extremely small footprint.** Forth systems can be tiny — useful on very resource-constrained chips, and it demystifies what's actually happening under

the hood (stacks, memory) in a way C's abstractions can hide.

5 **You're extending the language, not just calling it.** Every new word you define becomes a first-class part of the language, indistinguishable from the built-ins. This reframes "programming" as "growing your own vocabulary" rather than "assembling calls into a fixed API" — which tends to be a very sticky mental model once it clicks.

Tradeoffs

Forth's syntax (postfix/RPN, stack-based) is a real hurdle at first — 3 4 + instead of 3 + 4 trips people up before it clicks. And the Arduino IDE's ecosystem (libraries, examples, community support) is overwhelmingly C/C++-oriented, so Forth means fewer plug-and-play libraries for things like MIDI or audio codecs — you'll often be writing more from scratch, which is a double-edged sword pedagogically: more foundational understanding, but more upfront work.

The interactivity is probably the single biggest win. Being able to sit at a prompt and say "send this MIDI byte" or "read this ADC" and see the result *immediately* turns abstract hardware concepts into something you can poke and prod, which is usually where real understanding starts.

I actually believe that Forth code can be easier to read once you get familiar with the stack manipulations. I hope the code examples shown here give you that impression too.

Here is a short tutorial on Forth Stack manipulations to help with your transition.

Stack Manipulations

```
Forth's stack is just a pile of numbers, last one in is the first one out (LIFO —
Last In, First Out). Words consume numbers off the top and push results back on. Only
the top of the stack is ever immediately available.

Pushing numbers onto the stack

Typing a number just pushes it. Type 33 22 11 and the stack builds up like this, last
one pushed is on the top:

 11   <- top
 22
 33

You can check what's on the stack anytime with .s (a debugging word that prints the
```

stack without touching it). Also, ESP32Forth always prints what's currently on the stack before the cursor in the Serial Terminal.

Forth common practice is to provide stack notation for every word defined in the dictionary, how the word uses the stack. Stack notation is a parenthesized comment with a dash between the stack arguments. To the left of the dash are the stack arguments to be used by the word. To the right are the stack arguments the word will leave on the stack as results. Even if the word does not use the stack the stack notation is (--).

For example, the divide word is presented as / (n1 n2 -- n3). Two numbers are pushed onto the stack before the divide word, in this order: n1 n2, with n2 on the top. The divide word then pops the two values off the stack and calculates n1 divided by n2 (n1/n2), or n1below/n2top. The order is critical. The calculation result, n3 (the integer quotient), is then pushed onto the top of the stack.

Start with:

```
 11   <- top
 22
 33
```

Type / It pops 11 and 22, divides 22 by 11, and then pushes the result 2 onto the stack:

```
 2   <- top
 33
```

Another one: swap (n1 n2 -- n2 n1)

Starting fresh with 33 2:

```
 2   <- top
 33
```

Type swap. It flips the top two:

```
 33   <- top
 2
```

Chaining it all together in one command line. Type 33 22 11 / swap

```
      1.     Push all three:
 11   <- top
 22
 33
```

```
      2.     "/"  pops 11 and 22, pushes 2:
 2   <- top
 33
      3.     swap flips them:
 33   <- top
 2
```

That's the whole model: every word is either putting something on the stack, or taking something off it, pop its input arguments and push its results. Once you can trace a stack diagram like this by hand, the "weirdness" of Forth's syntax mostly disappears — it's really just following the pile up and down.