

# Das grosse Buch für ESP32forth

version 1.3 - 26. Okt. 2023



## Autor

- Marc PETREMANN      [petremann@arduino-forth.com](mailto:petremann@arduino-forth.com)

## Mitarbeiter

- Vaclav POSSELT
- Thomas SCHREIN

## Inhalt

Autor.....	1
Mitarbeiter.....	1
<b>Einführung.....</b>	<b>4</b>
Übersetzungshilfe.....	4
<b>Entdeckung der ESP32-Karte.....</b>	<b>5</b>
Präsentation.....	5
Die Stärken Punkten.....	5
GPIO-Ein-/Ausgänge auf ESP32.....	6
ESP32-Board-Peripheriegeräte.....	8
<b>Warum auf ESP32 in FORTH-Sprache programmieren?.....</b>	<b>9</b>
Präambel.....	9
Grenzen zwischen Sprache und Anwendung.....	10
Was ist ein FORTH-Wort?.....	10
Ein Wort ist eine Funktion?.....	10
FORTH-Sprache im Vergleich zur C-Sprache.....	11
Was FORTH Ihnen im Vergleich zur C-Sprache ermöglicht.....	12
Aber warum ein Stapel statt Variablen?.....	13
Sind Sie überzeugt?.....	13
Gibt es professionelle Bewerbungen, die in FORTH verfasst sind?.....	14
<b>Ein echtes 32-Bit FORTH mit ESP32Forth.....</b>	<b>16</b>
Werte auf dem Datenstapel.....	16
Werte im Gedächtnis.....	16
Textverarbeitung je nach Datengröße oder -typ.....	17
Abschluss.....	18
<b>Wörterbuch / Stapel / Variablen / Konstanten.....</b>	<b>20</b>
Wörterbuch erweitern.....	20
Wörterbuchverwaltung.....	20
Stapel und umgekehrte polnische Notation.....	21
Umgang mit dem Parameterstapel.....	22
Der Return Stack und seine Verwendung.....	23
Speichernutzung.....	23
Variablen.....	24
Konstanten.....	24
Pseudokonstante Werte.....	24
Grundlegende Tools für die Speicherzuweisung.....	25
<b>Textfarben und Anzeigeposition auf dem Terminal.....</b>	<b>26</b>
ANSI-Kodierung von Terminals.....	26
Textfärbung.....	27
Anzeigeposition.....	28
<b>Lokale Variablen mit ESP32Forth.....</b>	<b>30</b>
Einführung.....	30
Der Fake-Stack-Kommentar.....	30

Aktion auf lokale Variablen.....	31
<b>Datenstrukturen für ESP32forth.....</b>	<b>34</b>
Präambel.....	34
Tabellen in FORTH.....	34
Eindimensionales 32-Bit-Datenarray.....	34
Tabellendefinitionswörter.....	35
Lesen und schreiben Sie in eine Tabelle.....	35
Praktisches Beispiel für die Verwaltung eines virtuellen Bildschirms.....	36
Management komplexer Strukturen.....	39
Definition von Sprites.....	41
<b>Reale Zahlen mit ESP32forth.....</b>	<b>44</b>
Die echten mit ESP32forth.....	44
Echte Zahlengenauigkeit mit ESP32forth.....	44
Reale Konstanten und Variablen.....	45
Arithmetische Operatoren für reelle Zahlen.....	45
Mathematische Operatoren für reelle Zahlen.....	46
Logische Operatoren für reelle Zahlen.....	46
Ganzzahlige ↔ reelle Transformationen.....	47
<b>Installieren der OLED-Bibliothek für SSD1306.....</b>	<b>48</b>
<b>Ressourcen.....</b>	<b>50</b>
Auf Englisch.....	50
Auf Französisch.....	50
GitHub.....	50

# Einführung

Seit 2019 verwalte ich mehrere Websites, die sich der FORTH-Sprachentwicklung für ARDUINO- und ESP32-Karten sowie der eForth-Webversion widmen. :

- ARDUINO : <https://arduino-forth.com/>
- ESP32 : <https://esp32.arduino-forth.com/>
- eForth web : <https://eforth.arduino-forth.com/>

Diese Websites sind in zwei Sprachen verfügbar: Französisch und Englisch. Jedes Jahr bezahle ich für das Hosting der Hauptseite arduino-forth.com.

Es wird früher oder später – und zwar so spät wie möglich – passieren, dass ich die Nachhaltigkeit dieser Seiten nicht mehr gewährleisten kann. Die Folge wird sein, dass die von diesen Websites verbreiteten Informationen verschwinden.

Dieses Buch ist die Zusammenstellung von Inhalten meiner Websites. Es wird kostenlos über ein Github-Repository verteilt. Diese Verbreitungsmethode ermöglicht eine größere Nachhaltigkeit als Websites.

Wenn übrigens einige Leser dieser Seiten einen Beitrag leisten möchten, sind sie herzlich willkommen:

- Kapitel vorschlagen ;
- um Fehler zu melden oder Änderungen vorzuschlagen ;
- um bei der Übersetzung zu helfen...

## Übersetzungshilfe

Mit Google Translate können Sie Texte einfach, aber mit Fehlern übersetzen. Deshalb bitte ich um Hilfe bei der Korrektur der Übersetzungen.

In der Praxis stelle ich die bereits übersetzten Kapitel im LibreOffice-Format zur Verfügung. Wenn Sie bei diesen Übersetzungen helfen möchten, besteht Ihre Aufgabe lediglich darin, diese Übersetzungen zu korrigieren und zurückzugeben.

Das Korrigieren eines Kapitels nimmt wenig Zeit in Anspruch, von einer bis zu mehreren Stunden.

**Um mich zu erreichen :**            petremann@arduino-forth.com

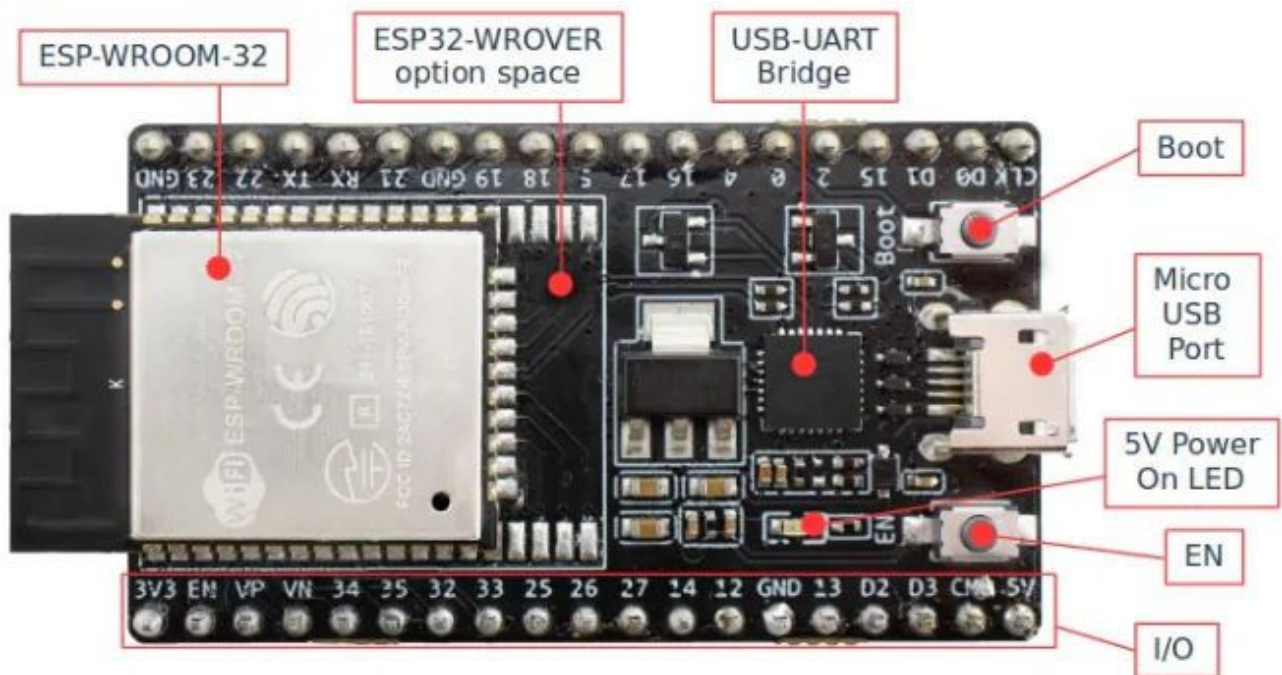
# Entdeckung der ESP32-Karte

## Präsentation

Das ESP32-Board ist kein ARDUINO-Board. Entwicklungstools nutzen jedoch bestimmte Elemente des ARDUINO-Ökosystems, wie beispielsweise die ARDUINO-IDE.

## Die Stärken Punkten

Hinsichtlich der Anzahl der verfügbaren Ports liegt die ESP32-Karte zwischen einem



ARDUINO NANO und ARDUINO UNO. Das Basismodell verfügt über 38 Anschlüsse :

Zu den ESP32-Geräten gehören :

- 18 Analog-Digital-Wandlerkanäle (ADC).
- 3 SPI-Schnittstellen
- 3 UART-Schnittstellen
- 2 I2C-Schnittstellen
- 16 PWM-Ausgangskanäle
- 2 Digital-Analog-Wandler (DAC)
- 2 I2S-Schnittstellen

- 10 kapazitive GPIOs

Die ADC- (Analog-Digital-Wandler) und DAC-Funktionalität (Digital-Analog-Wandler) sind bestimmten statischen Pins zugewiesen. Sie können jedoch entscheiden, welche Pins UART, I2C, SPI, PWM usw. sind. Sie müssen sie nur im Code zuweisen. Dies ist dank der Multiplexing-Funktion des ESP32-Chips möglich.

Die meisten Steckverbinder haben mehrere Verwendungszwecke.

Was das ESP32-Board jedoch auszeichnet, ist, dass es standardmäßig mit WLAN- und Bluetooth-Unterstützung ausgestattet ist, was ARDUINO-Boards nur in Form von Erweiterungen bieten.

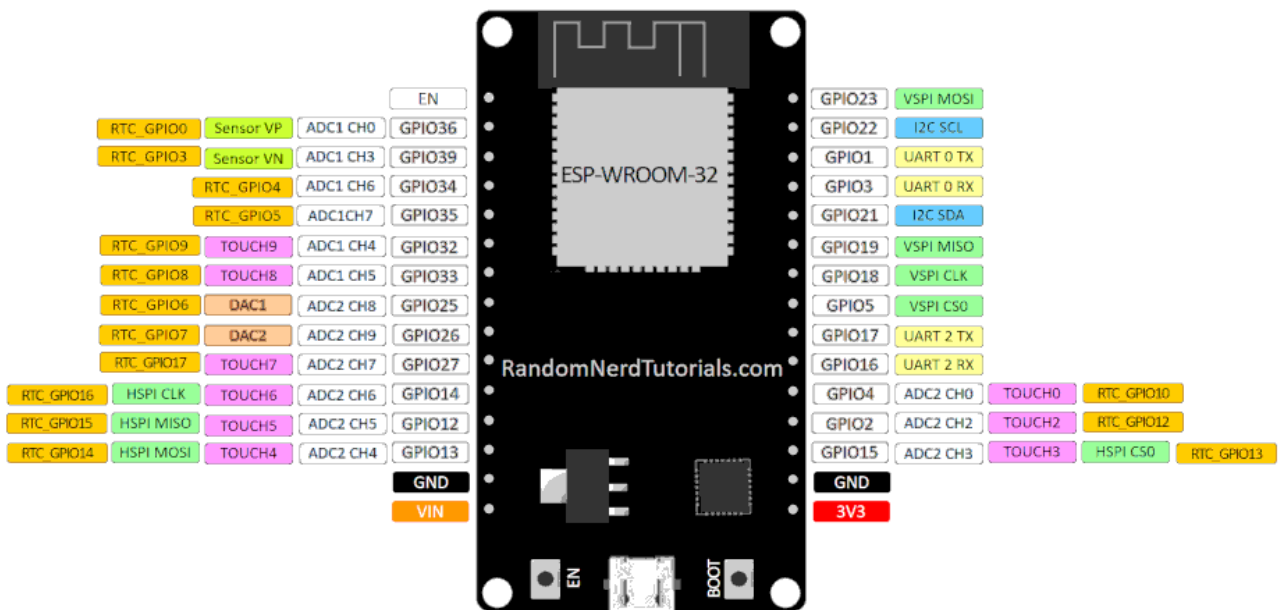
## GPIO-Ein-/Ausgänge auf ESP32

Hier im Foto die ESP32-Karte, anhand derer wir die Rolle der verschiedenen GPIO-Ein-/Ausgänge erklären :



Die Position und Anzahl der GPIO-I/Os kann sich je nach Kartenmarke ändern. In diesem Fall sind nur die Angaben auf der physischen Karte authentisch. Im Bild, untere Reihe, von links nach rechts: CLK, SD0, SD1, G15, G2, G0, G4, G16.....G22, G23, GND.





In diesem Diagramm sehen wir, dass die untere Reihe mit 3V3 beginnt, während sich dieser I/O auf dem Foto am Ende der oberen Reihe befindet. Daher ist es sehr wichtig, sich nicht auf das Diagramm zu verlassen, sondern den korrekten Anschluss der Peripheriegeräte und Komponenten auf der physischen ESP32-Karte noch einmal zu überprüfen.

Entwicklungsboards auf Basis eines ESP32 verfügen neben denen für die Stromversorgung in der Regel über 33 Pins. Einige GPIO-Pins haben besondere Funktionen :

GPIO	Mögliche Namen
6	SCK/CLK
7	SCK/CLK
8	SDO/SD0
9	SDI/SD1
10	SHD/SD2
11	CSC/CMD

Wenn Ihre ESP32-Karte über I/O GPIO6, GPIO7, GPIO8, GPIO9, GPIO10, GPIO11 verfügt, sollten Sie diese auf keinen Fall verwenden, da sie mit dem Flash-Speicher des ESP32 verbunden sind. Wenn Sie sie verwenden, funktioniert der ESP32 nicht.

GPIO1(TX0) und GPIO3(RX0) I/O werden für die Kommunikation mit dem Computer in UART über den USB-Port verwendet. Wenn Sie diese verwenden, können Sie nicht mehr mit der Karte kommunizieren.

GPIO36(VP), GPIO39(VN), GPIO34, GPIO35 I/O können nur als Eingang verwendet werden. Sie verfügen auch nicht über eingebaute interne Pullup- und Pulldown-Widerstände.



Mit dem EN-Anschluss können Sie den Zündstatus des ESP32 über ein externes Kabel steuern. Es wird mit der EN-Taste auf der Karte verbunden. Wenn der ESP32 eingeschaltet ist, liegt er bei 3,3 V. Wenn wir diesen Pin mit Masse verbinden, wird der ESP32 ausgeschaltet. Sie können es verwenden, wenn sich der ESP32 in einer Box befindet und Sie ihn mit einem Schalter ein-/ausschalten möchten.

## **ESP32-Board-Peripheriegeräte**

Um mit Modulen, Sensoren oder elektronischen Schaltkreisen zu interagieren, verfügt der ESP32 wie jeder Mikrocontroller über eine Vielzahl an Peripheriegeräten. Davon gibt es mehr als auf einem klassischen Arduino-Board.

ESP32 verfügt über die folgenden Peripheriegeräte :

- 3 UART-Schnittstellen
- 2 I2C-Schnittstellen
- 3 SPI-Schnittstellen
- 16 PWM-Ausgänge
- 10 kapazitive Sensoren
- 18 analoge Eingänge (ADC)
- 2 DAC-Ausgänge

Einige Peripheriegeräte werden bereits im Grundbetrieb von ESP32 genutzt. Somit gibt es pro Gerät weniger mögliche Schnittstellen.

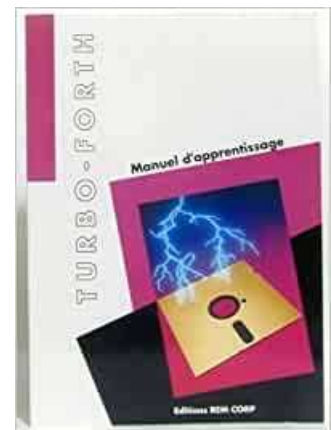
# Warum auf ESP32 in FORTH-Sprache programmieren?

## Präambel

Ich programmiere seit 1983 in FORTH. Ich habe 1996 mit dem Programmieren in FORTH aufgehört. Aber ich habe nie aufgehört, die Entwicklung dieser Sprache zu verfolgen. Ich habe 2019 wieder mit dem Programmieren auf ARDUINO mit FlashForth und dann mit ESP32forth begonnen.

Ich bin Co-Autor mehrerer Bücher über die FORTH-Sprache :

- Introduction au ZX-FORTH (ed Eyrolles - 1984 - ASIN:B0014IGOXO)
- Tours de FORTH (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH pour CP/M et MSDOS (ed Loisetech - 1986)
- TURBO-Forth, manuel d'apprentissage (ed Rem CORP - 1990)
- TURBO-Forth, guide de référence (ed Rem CORP - 1991)



Das Programmieren in der FORTH-Sprache war schon immer ein Hobby, bis mich 1992 der Manager eines Unternehmens kontaktierte, das als Zulieferer für die Automobilindustrie tätig war. Sie hatten ein Interesse an der Softwareentwicklung in der Sprache C. Sie mussten einen Industrieautomaten bestellen.

Die beiden Softwareentwickler dieser Firma programmierten in der Sprache C: TURBO-C von Borland, um genau zu sein. Und ihr Code konnte nicht kompakt und schnell genug sein, um in den 64 Kilobyte großen RAM-Speicher zu passen. Es war 1992 und Flash-Speichererweiterungen gab es noch nicht. In diesen 64 KB RAM mussten wir MS-DOS 3.0 und die Anwendung unterbringen!

Einen Monat lang hatten C-Entwickler das Problem in alle Richtungen umgedreht, sogar Reverse Engineering mit SOURCER (einem Disassembler), um nicht wesentliche Teile des ausführbaren Codes zu entfernen.

Ich habe das mir vorgelegte Problem analysiert. Von Grund auf habe ich innerhalb einer Woche alleine einen perfekt funktionsfähigen Prototyp erstellt, der den Spezifikationen entsprach. Drei Jahre lang, von 1992 bis 1995, habe ich zahlreiche Versionen dieser Anwendung erstellt, die auf den Montagebändern mehrerer Automobilhersteller eingesetzt wurden.

# Grenzen zwischen Sprache und Anwendung

Alle Programmiersprachen werden auf diese Weise geteilt :

- ein Interpreter und ausführbarer Quellcode: BASIC, PHP, MySQL, JavaScript usw. Die Anwendung ist in einer oder mehreren Dateien enthalten, die bei Bedarf interpretiert werden. Das System muss den Interpreter, der den Quellcode ausführt, dauerhaft hosten ;
- ein Compiler und/oder Assembler: C, Java usw. Einige Compiler generieren nativen Code, also speziell auf einem System ausführbar. Andere, wie Java, kompilieren ausführbaren Code auf einer virtuellen Java-Maschine.

Eine Ausnahme bildet die FORTH-Sprache. Es integriert :

- ein Dolmetscher, der jedes Wort in der FORTH-Sprache ausführen kann
- ein Compiler, der das Wörterbuch der FORTH-Wörter erweitern kann

## Was ist ein FORTH-Wort?

Ein FORTH-Wort bezeichnet einen beliebigen Wörterbuchausdruck, der aus ASCII-Zeichen besteht und bei der Interpretation und/oder Kompilierung verwendet werden kann: Mit Wörter können Sie alle Wörter im FORTH-Wörterbuch auflisten.

Bestimmte FORTH-Wörter können nur bei der Kompilierung verwendet werden: **if else then** zum Beispiel.

Bei der FORTH-Sprache besteht das wesentliche Prinzip darin, dass wir keine Anwendung erstellen. In FORTH erweitern wir das Wörterbuch! Jedes neue Wort, das Sie definieren, ist ebenso Teil des FORTH-Wörterbuchs wie alle beim Start von FORTH vordefinierten Wörter. Beispiel :

```
: typeToLoRa ( -- )
  0 echo !      \ Deaktivieren Sie das Echo der Terminalanzeige
  ['] serial2-type is type
;
: typeToTerm ( -- )
  ['] default-type is type
  -1 echo !     \ Aktiviert das Display-Echo des Terminals
;
```

Wir erstellen zwei neue Wörter: **typeToLoRa** und **typeToTerm**, die das Wörterbuch vordefinierter Wörter vervollständigen.

## Ein Wort ist eine Funktion?

Ja und nein. Tatsächlich kann ein Wort eine Konstante, eine Variable, eine Funktion sein... Hier in unserem Beispiel die folgende Sequenz :

```
: typeToLoRa ...code... ;
```

hätte sein Äquivalent in der C-Sprache :

```
void typeToLoRa() { ...code... }
```

In der FORTH-Sprache gibt es keine Grenze zwischen Sprache und Anwendung.

In FORTH können Sie wie in der Sprache C jedes bereits definierte Wort in der Definition eines neuen Worts verwenden.

Ja, aber warum dann FORTH statt C?

Ich habe diese Frage erwartet.

In der C-Sprache kann auf eine Funktion nur über die Hauptfunktion main() zugegriffen werden. Wenn diese Funktion mehrere Zusatzfunktionen integriert, wird es bei einer Fehlfunktion des Programms schwierig, einen Parameterfehler zu finden.

Im Gegenteil, mit FORTH ist es möglich, über den Interpreter jedes vordefinierte oder von Ihnen definierte Wort auszuführen, ohne das Hauptwort des Programms durchlaufen zu müssen.

Der FORTH-Interpreter ist über ein Terminalprogramm und eine USB-Verbindung zwischen der ESP32-Karte und dem PC sofort auf der ESP32-Karte zugänglich.

Die Kompilierung von in FORTH-Sprache geschriebenen Programmen erfolgt in der ESP32-Karte und nicht auf dem PC. Es erfolgt kein Upload. Beispiel:

```
: >gray ( n -- n' )  
    dup 2/ xor      \ n' = n xor ( 1 logische Verschiebung nach  
rechts )  
    ;
```

Diese Definition wird per Kopieren/Einfügen in das Terminal übertragen. Der FORTH-Interpreter/Compiler analysiert den Stream und kompiliert das neue Wort **>gray**.

In der Definition von **>gray** sehen wir die Sequenz **dup 2/ xor**. Um diese Sequenz zu testen, geben Sie sie einfach in das Terminal ein. Um **>gray** auszuführen, geben Sie einfach dieses Wort in das Terminal ein, gefolgt von der Zahl, die umgewandelt werden soll.

## FORTH-Sprache im Vergleich zur C-Sprache

Das ist der Teil, den ich am wenigsten mag. Ich vergleiche die FORTH-Sprache nicht gern mit der C-Sprache. Aber da fast alle Entwickler die C-Sprache verwenden, werde ich die Übung ausprobieren.

Hier ist ein Test mit **if()** in C-Sprache:

```
if(j > 13){                // Wenn alle Bits empfangen wurden  
    rc5_ok = 1;           // Der Dekodierungsprozess ist OK
```

```

detachInterrupt(0); // Externen Interrupt deaktivieren (INT0)
return;
}

```

Testen Sie mit if in FORTH-Sprache (Code-Snippet) :

```

var-j @ 13 >          \ Wenn alle Bits empfangen wurden
  if
    1 rc5_ok !        \ Der Dekodierungsprozess ist OK
    di                \ Externen Interrupt deaktivieren (INT0)
    exit
  then

```

Hier ist die Initialisierung von Registern in C-Sprache :

```

void setup() {
  // Konfigurieren des Timer1-Moduls
  TCCR1A = 0;
  TCCR1B = 0;          // Deaktiviert das Timer1-Modul
  TCNT1 = 0;           // Setzt den Vorladewert von Timer1 auf 0
  (reset)
  TIMSK1 = 1;          // Überlauf-Interrupt aktivieren Timer1
}

```

Die gleiche Definition in der FORTH-Sprache:

```

: setup ( -- )
  \ Konfigurieren des Timer1-Moduls
  0 TCCR1A !
  0 TCCR1B !          \ Deaktiviert das Timer1-Modul
  0 TCNT1 !           \ Setzt den Vorladewert von Timer1 auf 0 (reset)
  1 TIMSK1 !          \ Überlauf-Interrupt aktivieren Timer1
;

```

## Was FORTH Ihnen im Vergleich zur C-Sprache ermöglicht

Wir verstehen, dass FORTH sofort Zugriff auf alle Wörter im Wörterbuch bietet, aber nicht nur darauf. Über den Interpreter greifen wir auch auf den gesamten Speicher der ESP32-Karte zu. Stellen Sie eine Verbindung zum ARDUINO-Board her, auf dem FlashForth installiert ist, und geben Sie dann einfach Folgendes ein:

```
hex here 100 dump
```

Sie sollten dies auf dem Terminalbildschirm finden :

```

3FFEE964          DF DF 29 27 6F 59 2B 42 FA CF 9B 84
3FFEE970          39 4E 35 F7 78 FB D2 2C A0 AD 5A AF 7C 14 E3 52
3FFEE980          77 0C 67 CE 53 DE E9 9F 9A 49 AB F7 BC 64 AE E6
3FFEE990          3A DF 1C BB FE B7 C2 73 18 A6 A5 3F A4 68 B5 69
3FFEE9A0          F9 54 68 D9 4D 7C 96 4D 66 9A 02 BF 33 46 46 45

```

3FFEE9B0	45 39 33 33 2F 0D 08 18 BF 95 AF 87 AC D0 C7 5D
3FFEE9C0	F2 99 B6 43 DF 19 C9 74 10 BD 8C AE 5A 7F 13 F1
3FFEE9D0	9E 00 3D 6F 7F 74 2A 2B 52 2D F4 01 2D 7D B5 1C
3FFEE9E0	4A 88 88 B5 2D BE B1 38 57 79 B2 66 11 2D A1 76
3FFEE9F0	F6 68 1F 71 37 9E C1 82 43 A6 A4 9A 57 5D AC 9A
3FFEEA00	4C AD 03 F1 F8 AF 2E 1A B4 67 9C 71 25 98 E1 A0
3FFEEA10	E6 29 EE 2D EF 6F C7 06 10 E0 33 4A E1 57 58 60
3FFEEA20	08 74 C6 70 BD 70 FE 01 5D 9D 00 9E F7 B7 E0 CA
3FFEEA30	72 6E 49 16 0E 7C 3F 23 11 8D 66 55 EC F6 18 01
3FFEEA40	20 E7 48 63 D1 FB 56 77 3E 9A 53 7D B6 A7 A5 AB
3FFEEA50	EA 65 F8 21 3D BA 54 10 06 16 E6 9E 23 CA 87 25
3FFEEA60	E7 D7 C4 45

Dies entspricht dem Inhalt des Flash-Speichers.

Und die C-Sprache konnte das nicht?

Ja, aber nicht so einfach und interaktiv wie in der FORTH-Sprache.

## Aber warum ein Stapel statt Variablen?

Der Stack ist ein Mechanismus, der auf fast allen Mikrocontrollern und Mikroprozessoren implementiert ist. Sogar die C-Sprache nutzt einen Stack, aber Sie haben keinen Zugriff darauf.

Nur die FORTH-Sprache bietet vollständigen Zugriff auf den Datenstapel. Um beispielsweise eine Addition durchzuführen, stapeln wir zwei Werte, führen die Addition aus und zeigen das Ergebnis an: **2 5 + .** zeigt **7** an.

Es ist ein wenig destabilisierend, aber wenn Sie den Mechanismus des Datenstapels verstehen, werden Sie seine beeindruckende Effizienz sehr zu schätzen wissen.

Mit dem Datenstapel können Daten viel schneller zwischen FORTH-Worten übertragen werden als durch die Verarbeitung von Variablen wie in der C-Sprache oder einer anderen Sprache, die Variablen verwendet.

## Sind Sie überzeugt?

Persönlich bezweifle ich, dass dieses einzelne Kapitel Sie endgültig zum Programmieren in der FORTH-Sprache bekehren wird. Wenn Sie ESP32-Boards beherrschen möchten, haben Sie zwei Möglichkeiten :

- Programmieren Sie das Programm in C-Sprache und nutzen Sie die zahlreichen verfügbaren Bibliotheken. Sie bleiben jedoch an die Möglichkeiten dieser Bibliotheken gebunden. Die Anpassung von Codes an die C-Sprache erfordert echte Programmierkenntnisse in der C-Sprache und die Beherrschung der Architektur von ESP32-Karten. Die Entwicklung komplexer Programme wird immer ein Problem sein.

- Probieren Sie das FORTH-Abenteuer aus und erkunden Sie eine neue und aufregende Welt. Natürlich wird es nicht einfach sein. Sie müssen die Architektur von ESP32-Karten, die Register und die Registerflags im Detail verstehen. Im Gegenzug erhalten Sie Zugang zu einer Programmierung, die perfekt zu Ihren Projekten passt.

## Gibt es professionelle Bewerbungen, die in FORTH verfasst sind?

Oh ja! Beginnend mit dem HUBBLE-Weltraumteleskop, dessen bestimmte Komponenten in der FORTH-Sprache geschrieben wurden.

Der deutsche TGV ICE (Intercity Express) nutzt RTX2000-Prozessoren zur Steuerung von Motoren über Leistungshalbleiter. Die Maschinensprache des RTX2000-Prozessors ist die FORTH-Sprache.

Derselbe RTX2000-Prozessor wurde für die Philae-Sonde verwendet, die versuchte, auf einem Kometen zu landen.

Die Wahl der FORTH-Sprache für professionelle Anwendungen erweist sich als interessant, wenn wir jedes Wort als Blackbox betrachten. Jedes Wort muss einfach sein, daher eine relativ kurze Definition haben und von wenigen Parametern abhängen.

Während der Debugging-Phase ist es einfach, alle möglichen Werte zu testen, die von diesem Wort verarbeitet werden. Sobald dieses Wort vollkommen zuverlässig ist, wird es zu einer Blackbox, also zu einer Funktion, deren ordnungsgemäßes Funktionieren wir absolut vertrauen können. Von Wort zu Wort ist es in FORTH einfacher, ein komplexes Programm zuverlässig zu machen als in jeder anderen Programmiersprache.

Aber wenn es uns an Genauigkeit mangelt, wenn wir Gasanlagen bauen, ist es auch sehr leicht, dass eine Anwendung schlecht funktioniert oder sogar völlig abstürzt!

Schließlich ist es in der FORTH-Sprache möglich, die von Ihnen definierten Wörter in jeder menschlichen Sprache zu schreiben. Allerdings sind die verwendbaren Zeichen auf den ASCII-Zeichensatz zwischen 33 und 127 beschränkt. So könnten wir die Wörter **high** und **low** symbolisch umschreiben :

```
\ Aktiver Port-Pin, andere nicht ändern.
: __/ ( pinmask portadr -- )
  mset
;
\ Das Deaktivieren eines Port-Pins hat keine Auswirkungen auf die
anderen.
: \__ ( pinmask portadr -- )
  mclr
;
```



Ab diesem Moment können Sie zum Einschalten der LED Folgendes eingeben :

```
_0_ ___/ \ allume LED
```

Ja! Die Sequenz **\_0\_ \_\_\_/** ist in FORTH-Sprache!

Mit ESP32forth stehen Ihnen hier alle Zeichen zur Verfügung, die ein FORTH-Wort bilden können:

```
~}|{zyxwvutsrqponmlkjihgfedcba`_  
^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?  
>=<;:9876543210/.- ,+*)('&%$#"!
```

Gute Programmierung.

# Ein echtes 32-Bit FORTH mit ESP32Forth

ESP32Forth ist ein echtes 32-Bit FORTH. Was bedeutet das ?

Die FORTH-Sprache bevorzugt die Manipulation ganzzahliger Werte. Diese Werte können Literalwerte, Speicheradressen, Registerinhalte usw. sein.

## Werte auf dem Datenstapel

Wenn ESP32Forth startet, ist der FORTH-Interpreter verfügbar. Wenn Sie eine beliebige Zahl eingeben, wird diese als 32-Bit-Ganzzahl auf dem Stapel abgelegt:

```
35
```

Wenn wir einen anderen Wert stapeln, wird dieser ebenfalls gestapelt. Der vorherige Wert wird um eine Position nach unten verschoben:

```
45
```

Um diese beiden Werte zu addieren, verwenden wir ein Wort, hier **+** :

```
+
```

Unsere beiden 32-Bit-Ganzzahlwerte werden addiert und das Ergebnis auf dem Stapel abgelegt. Um dieses Ergebnis anzuzeigen, verwenden wir das Wort **.** :

```
. \ zeigt 80 an
```

In der FORTH-Sprache können wir alle diese Operationen in einer einzigen Zeile konzentrieren:

```
35 45 +. \ 80 anzeigen
```

Im Gegensatz zur C-Sprache definieren wir keinen **int8-**, **int16-** oder **int32- Typ** .

Mit ESP32Forth wird ein ASCII-Zeichen durch eine 32-Bit-Ganzzahl bezeichnet, deren Wert jedoch auf [32..256[ begrenzt ist. Beispiel :

```
67 emit \ zeigt C
```

## Werte im Gedächtnis

Mit ESP32Forth können Sie Konstanten und Variablen definieren. Ihr Inhalt liegt immer im 32-Bit-Format vor. Aber es gibt Situationen, in denen uns das nicht unbedingt passt. Nehmen wir ein einfaches Beispiel und definieren ein Morsecode-Alphabet. Wir brauchen nur ein paar Bytes:

- eines, um die Anzahl der Morsezeichen zu definieren
- ein oder mehrere Bytes für jeden Buchstaben des Morsecodes

```
create mA ( -- addr )
  2 c,
  char . c,   char - c,

create mB ( -- addr )
  4 c,
  char - c,   char . c,   char . c,   char . c,

create mC ( -- addr )
  4 c,
  char - c,   char . c,   char - c,   char . c,
```

Hier definieren wir nur 3 Wörter, **mA** , **mB** und **mC** . In jedem Wort werden mehrere Bytes gespeichert. Die Frage ist : Wie werden wir die Informationen in diesen Worten abrufen ?

Die Ausführung eines dieser Wörter hinterlegt einen 32-Bit-Wert, einen Wert, der der Speicheradresse entspricht, an der wir unsere Morsecode-Informationen gespeichert haben. Es ist das Wort **c@** , das wir verwenden werden, um den Morsecode aus jedem Buchstaben zu extrahieren :

```
mA c@ . \ zeigt 2 an
mB c@ . \ zeigt 4 an
```

Das erste so extrahierte Byte wird zur Verwaltung einer Schleife zur Anzeige des Morsecodes eines Buchstabens verwendet :

```
: .morse ( addr -- )
  dup 1+ swap c@ 0 do
    dup i + c@ emit
  loop
  drop
;
mA .morse \ zeigt .-
mB .morse \ zeigt -...
mC .morse \ zeigt -.-.
```

Es gibt sicherlich viele elegantere Beispiele. Hier soll eine Möglichkeit gezeigt werden, 8-Bit-Werte, unsere Bytes, zu manipulieren, während wir diese Bytes auf einem 32-Bit-Stack verwenden.

## Textverarbeitung je nach Datengröße oder -typ

In allen anderen Sprachen gibt es ein generisches Wort wie **echo** (in PHP), das jede Art von Daten anzeigt. Ob Integer, Real, String, wir verwenden immer das gleiche Wort.

Beispiel in PHP-Sprache:

```
$bread = "Gebackenes Brot";
$preis = 2,30;
echo $bread . " : " . $preis;
// zeigt Gebackenes Brot: 2,30
```

Für alle Programmierer ist diese Vorgehensweise DER STANDARD! Wie würde FORTH dieses Beispiel in PHP umsetzen?

```
: bread s" Baked bread" ;
: price s" 2.30" ;
bread type    s" : " type    price type
\ display    Baked bread: 2.30
```

Hier sagt uns der **type** , dass wir gerade eine Zeichenfolge verarbeitet haben.

Wo PHP (oder eine andere Sprache) über eine generische Funktion und einen Parser verfügt, kompensiert FORTH dies mit einem einzigen Datentyp, aber angepassten Verarbeitungsmethoden, die uns über die Art der verarbeiteten Daten informieren.

Hier ist ein absolut trivialer Fall für FORTH, bei dem eine Anzahl von Sekunden im Format HH:MM:SS angezeigt wird:

```
: :##
  # 6 base !
  # decimal
  [char] : hold
;
: .hms ( n -- )
  <# :## :## # # #> type
;
4225 .hms \ zeigt: 01:10:25
```

Ich liebe dieses Beispiel, weil bisher **KEINE ANDERE PROGRAMMIERSPRACHE** in der Lage ist, diese HH:MM:SS-Konvertierung so elegant und prägnant durchzuführen.

Sie haben verstanden, das Geheimnis von FORTH liegt in seinem Wortschatz.

## Abschluss

FORTH hat keine Datentypisierung. Alle Daten durchlaufen einen Datenstapel. Jede Position im Stapel ist IMMER eine 32-Bit-Ganzzahl!

### Das ist alles, was man wissen muss.

Puristen hyperstrukturierter und ausführlicher Sprachen wie C oder Java werden sicherlich Häresie ausrufen. Und hier erlaube ich mir, sie zu beantworten : Warum müssen Sie Ihre Daten eingeben ?

Denn in dieser Einfachheit liegt die Stärke von FORTH: ein einzelner Datenstapel mit einem untypisierten Format und sehr einfachen Operationen.

Und ich zeige Ihnen, was viele andere Programmiersprachen nicht können, nämlich neue Definitionswörter zu definieren :

```
: morse: ( comp: c -- | exec -- )
  create
    c,
  does>
    dup 1+ swap c@ 0 do
      dup i + c@ emit
    loop
  drop space
;
2 morse: mA      char . c,   char - c,
4 morse: mB      char - c,   char . c,   char . c,   char . c,
4 morse: mC      char - c,   char . c,   char - c,   char . c,
mA mB mC      \ zeigt   .- -... -.-.
```

Hier ist das Wort **morse:** zu einem Definitionswort geworden, ebenso wie **constant** oder **variable** ...

Denn FORTH ist mehr als eine Programmiersprache. Es handelt sich um eine Metasprache, also um eine Sprache zum Aufbau einer eigenen Programmiersprache....

# Wörterbuch / Stapel / Variablen / Konstanten

## Wörterbuch erweitern

Forth gehört zur Klasse der gewebten Interpretationssprachen. Das bedeutet, dass es auf der Konsole eingegebene Befehle interpretieren sowie neue Unterroutinen und Programme kompilieren kann.

Der Forth-Compiler ist Teil der Sprache und spezielle Wörter werden verwendet, um neue Wörterbucheinträge (d. h. Wörter) zu erstellen. Die wichtigsten sind **:** (eine neue Definition beginnen) und **;** (beendet die Definition). Versuchen wir es, indem wir Folgendes eingeben:

```
: *+ * + ;
```

Was ist passiert? Die Aktion von: besteht darin, einen neuen Wörterbucheintrag mit dem Namen **\*+** zu erstellen und vom Interpretationsmodus in den Kompilierungsmodus zu wechseln. Im Kompilierungsmodus sucht der Interpreter nach Wörtern und anstatt sie auszuführen, installiert er Zeiger auf ihren Code. Wenn es sich bei dem Text um eine Zahl handelt, legt ESP32forth die Zahl nicht auf den Stapel, sondern erstellt sie im für das neue Wort zugewiesenen Wörterbuchraum. Dabei folgt er einem speziellen Code, der die gespeicherte Zahl bei jeder Ausführung des Worts auf den Stapel legt. Die Ausführungsaktion von **\*+** besteht daher darin, die zuvor definierten Wörter **\*** und **+** nacheinander auszuführen.

Das Wort **;** ist besonders. Es ist ein unmittelbares Wort und wird immer ausgeführt, auch wenn sich das System im Kompilierungsmodus befindet. Was bedeutet **;** ist zweifach. Erstens wird Code installiert, der die Kontrolle an die nächste externe Ebene des Interpreters zurückgibt, und zweitens kehrt es vom Kompilierungsmodus in den Interpretationsmodus zurück.

Probieren Sie jetzt Ihr neues Wort aus:

```
dezimal 5 6 7 *+ . \ zeigt 47 ok<#,ram> an
```

Dieses Beispiel veranschaulicht zwei Hauptarbeitsaktivitäten in Forth: das Hinzufügen eines neuen Worts zum Wörterbuch und das Ausprobieren, sobald es definiert wurde.

## Wörterbuchverwaltung

Das Wort **forget** gefolgt vom zu löschenden Wort entfernt alle Wörterbucheinträge, die Sie seit diesem Wort gemacht haben:

```
: test1 ;
```

```
: test2 ;  
: test3 ;  
forget test2 \ test2 und test3 aus dem Wörterbuch löschen
```

## Stapel und umgekehrte polnische Notation

Forth verfügt über einen explizit sichtbaren Stapel, der zum Übergeben von Zahlen zwischen Wörtern (Befehlen) verwendet wird. Die effektive Verwendung von Forth zwingt Sie dazu, im Stapel zu denken. Das kann am Anfang schwierig sein, aber wie bei allem wird es mit der Übung viel einfacher.

In FORTH ist der Stapel analog zu einem Kartenstapel mit darauf geschriebenen Zahlen. Zahlen werden immer oben auf dem Stapel hinzugefügt und oben vom Stapel entfernt. ESP32forth integriert zwei Stacks: den Parameter-Stack und den Feedback-Stack, die jeweils aus einer Reihe von Zellen bestehen, die 16-Bit-Zahlen aufnehmen können.

Die FORTH-Eingabezeile:

```
dezimal 2 5 73 -16
```

lässt den Parameterstapel unverändert

Zelle	Inhalt	Kommentar
0	-16	(TOS) Oben rechts
1	73	(NOS) Als nächstes im Stapel
2	5	
3	2	

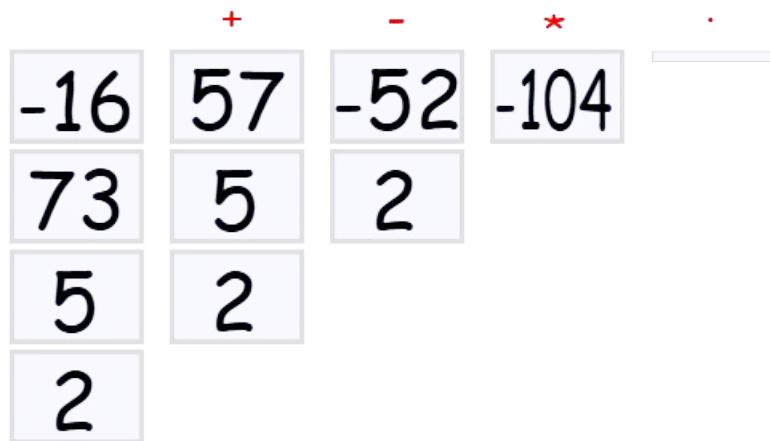
In Forth-Datenstrukturen wie Stapeln, Arrays und Tabellen verwenden wir normalerweise eine auf Null basierende relative Nummerierung. Beachten Sie, dass bei der Eingabe einer Zahlenfolge auf diese Weise die Zahl ganz rechts *zum TOS wird* und die Zahl ganz links unten im Stapel liegt.

Angenommen, wir folgen der ursprünglichen Eingabezeile mit der Zeile

```
+ - * .
```

Die Operationen würden aufeinanderfolgende Stapeloperationen erzeugen:





Nach den beiden Zeilen zeigt die Konsole Folgendes an:

```
decimal 2 5 73 -16    \ zeigt an: 2 5 73 -16 ok
+ - * .               \ zeigt an: -104 ok
```

Beachten Sie, dass ESP32forth die Stapелеlemente bei der Interpretation jeder Zeile bequem anzeigt und dass der Wert -16 als 32-Bit-Ganzzahl ohne Vorzeichen angezeigt wird. Darüber hinaus ist das Wort `.` verbraucht den Datenwert -104 und lässt den Stapel leer. Wenn wir ausführen. Auf dem nun leeren Stack bricht der externe Interpreter mit einem Stack-Pointer-Fehler `STACK UNDERFLOW ERROR` ab.

Die Programmierschreibweise, bei der die Operanden zuerst erscheinen, gefolgt von den Operatoren, wird Reverse Polish Notation (RPN) genannt.

## Umgang mit dem Parameterstapel

Da es sich um ein stapelbasiertes System handelt, muss ESP32forth Möglichkeiten bieten, Zahlen auf den Stapel zu legen, sie zu entfernen und ihre Reihenfolge neu zu ordnen. Wir haben bereits gesehen, dass wir Zahlen einfach durch Eintippen auf den Stapel legen können. Wir können auch Zahlen in die Definition eines FORTH-Wortes integrieren.

Durch das Wort `drop` wird eine Zahl von der obersten Stelle des Stapels entfernt, sodass die nächste Zahl oben liegt. `swap` Worttausch werden die ersten beiden Zahlen ausgetauscht. `dup` kopiert die Zahl oben und verschiebt alle anderen Zahlen nach unten. `rot` dreht die ersten 3 Zahlen. Diese Aktionen werden im Folgenden vorgestellt.



## Der Return Stack und seine Verwendung

Beim Kompilieren eines neuen Wortes stellt ESP32forth Verknüpfungen zwischen dem aufrufenden Wort und zuvor definierten Wörtern her, die bei der Ausführung des neuen Wortes aufgerufen werden sollen. Dieser Verknüpfungsmechanismus verwendet zur Laufzeit den Rstack. Die Adresse des nächsten aufzurufenden Wortes wird auf dem hinteren Stapel abgelegt, sodass das System nach Abschluss der Ausführung des aktuellen Worts weiß, wohin es zum nächsten Wort wechseln muss. Da Wörter verschachtelt werden können, muss ein Stapel dieser Rücksprungadressen vorhanden sein.

Der Benutzer dient nicht nur als Reservoir für Rücksprungadressen, sondern kann auch den Rückgabestapel speichern und von dort abrufen. Dabei muss jedoch sorgfältig vorgegangen werden, da der Rückgabestapel für die Programmausführung unerlässlich ist. Wenn Sie den Rückgabeakku zur vorübergehenden Speicherung verwenden, müssen Sie ihn in seinen ursprünglichen Zustand zurückversetzen, da es sonst wahrscheinlich zu einem Absturz des ESP32forth-Systems kommt. Trotz der Gefahr gibt es Zeiten, in denen die Verwendung von Backstack als temporärer Speicher Ihren Code weniger komplex machen kann.

Zum Speichern auf dem Stapel verwenden Sie **>r** , um den oberen Rand des Parameterstapels an den oberen Rand des Rückgabestapels zu verschieben. Um einen Wert abzurufen, verschiebt **r>** den obersten Wert vom Stapel zurück an die Spitze des Parameterstapels. Um einfach einen Wert oben vom Stapel zu entfernen, gibt es das Wort **rdrop** . Das Wort **r@** kopiert den oberen Teil des Stapels zurück in den Parameterstapel.

## Speichernutzung

**@** (fetch) aus dem Speicher auf den Stapel geholt und mit dem Wort **!** von oben im Speicher abgelegt. (blind). **@** erwartet eine Adresse auf dem Stapel und ersetzt die Adresse durch ihren Inhalt. **!** erwartet eine Nummer und eine Adresse, um es zu speichern. Die Nummer wird an dem Speicherort abgelegt, auf den sich die Adresse bezieht, wobei dabei beide Parameter verbraucht werden.

Vorzeichenlose Zahlen, die 8-Bit-Werte (Byte) darstellen, können in zeichengroße Zeichen eingefügt werden. Speicherzellen mit **c@** und **c!** .

```
create testVar
  cell allot
  $f7 testVar c!
  testVar c@ . \ zeigt 247 an
```

## Variablen

Eine Variable ist ein benannter Speicherort im Speicher, der eine Zahl, beispielsweise das Zwischenergebnis einer Berechnung, außerhalb des Stapels speichern kann. Zum Beispiel :

```
Variable x
```

erstellt einen Speicherort mit dem Namen **x** , der ausgeführt wird und die Adresse seines Speicherorts oben im Stapel belässt:

```
X . \ zeigt die Adresse an
```

Wir können die Daten dann an dieser Adresse abholen oder speichern:

```
Variable x
3 x !
X @ . \ zeigt an: 3
```

## Konstanten

Eine Konstante ist eine Zahl, die Sie während der Ausführung eines Programms nicht ändern möchten. Das Ergebnis der Ausführung des mit einer Konstanten verbundenen Wortes ist der Wert der auf dem Stapel verbleibenden Daten.

```
\ definiert VSPI-Pins
19 constant VSPI_MISO
23 constant VSPI_MOSI
18 constant VSPI_SCLK
05 constant VSPI_CS

\ legt die SPI-Portfrequenz fest
4000000 Konstante SPI_FREQ

\ SPI-Vokabular auswählen
only FORTH SPI also

\ initialisiert den SPI-Port
: init.VSPI ( -- )
  VSPI_CS OUTPUT pinMode
  VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin
  SPI_FREQ SPI.setFrequency
;
```

## Pseudokonstante Werte

Ein mit value definierter Wert ist ein Hybridtyp aus Variable und Konstante. Wir legen einen Wert fest, initialisieren ihn und er wird wie eine Konstante aufgerufen. Wir können einen Wert auch ändern, so wie wir eine Variable ändern können.

```
decimal
13 value thirteen
thirteen . \ Anzeige: 13
```

```
47 to thirteen
thirteen .      \ Anzeige: 47
```

Das Wort **to** funktioniert auch in Wortdefinitionen und ersetzt den darauf folgenden Wert durch den Wert, der sich gerade ganz oben im Stapel befindet. Sie müssen darauf achten, dass auf **to** ein durch value definierter Wert folgt und nicht etwas anderes.

## Grundlegende Tools für die Speicherzuweisung

Die Wörter **create** und **allot** sind die grundlegenden Werkzeuge zum Reservieren von Speicherplatz und zum Anbringen einer Bezeichnung. Die folgende Transkription zeigt beispielsweise einen neuen Eintrag **graphic-array** im Wörterbuch :

```
create graphic-array ( --- addr )
  %00000000 c,
  %00000010 c,
  %00000100 c,
  %00001000 c,
  %00010000 c,
  %00100000 c,
  %01000000 c,
  %10000000 c,
```

Bei der Ausführung überträgt das Wort **graphic-array** die Adresse des ersten Eintrags.

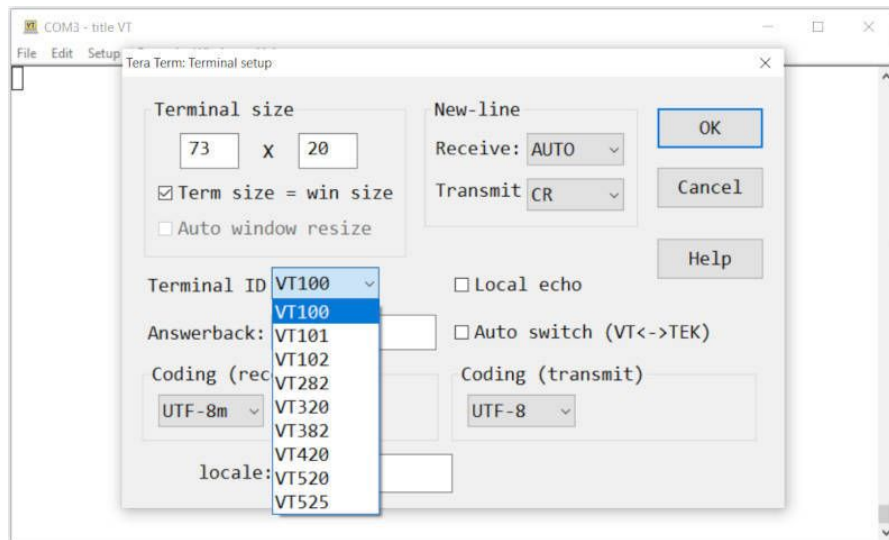
Wir können jetzt mit den zuvor erläuterten Abruf- und Speicherwörtern auf den dem **graphic-array** zugewiesenen Speicher zugreifen. Um die Adresse des dritten Bytes zu berechnen, das dem **graphic-array**, können wir **graphic-array 2 +** schreiben , wobei wir bedenken, dass die Indizes bei 0 beginnen.

```
30 graphic-array 2 + c!
graphic-array 2 + c@ .      \ affiche 30
```

# Textfarben und Anzeigeposition auf dem Terminal

## ANSI-Kodierung von Terminals

Wenn Sie Terminalsoftware zur Kommunikation mit ESP32forth verwenden, besteht eine gute Chance, dass dieses Terminal ein VT-Terminal oder ein gleichwertiges Terminal emuliert. Hier ist TeraTerm so konfiguriert, dass es ein VT100-Terminal emuliert:



Diese Terminals verfügen über zwei interessante Funktionen :

- Färben Sie den Seitenhintergrund und den anzuzeigenden Text
- Positionieren Sie den Anzeigecursor

Beide Funktionen werden durch ESC-Sequenzen (Escape-Sequenzen) gesteuert. So sind die Wörter **bg** und **fg** in ESP32forth definiert:

```
forth definitions ansi
: fg ( n -- ) esc ." [38;5;" n. ." m" ;
: bg ( n -- ) esc ." [48;5;" n. ." m" ;
: normal   esc ." [0m" ;
: at-xy ( x y -- ) esc ." [" 1+ n. ." ;" 1+ n. ." H" ;
: page    esc ." [2J" esc ." [H" ;
```

**normal** Wort überschreibt die durch **bg** und **fg** definierten Farbsequenzen.

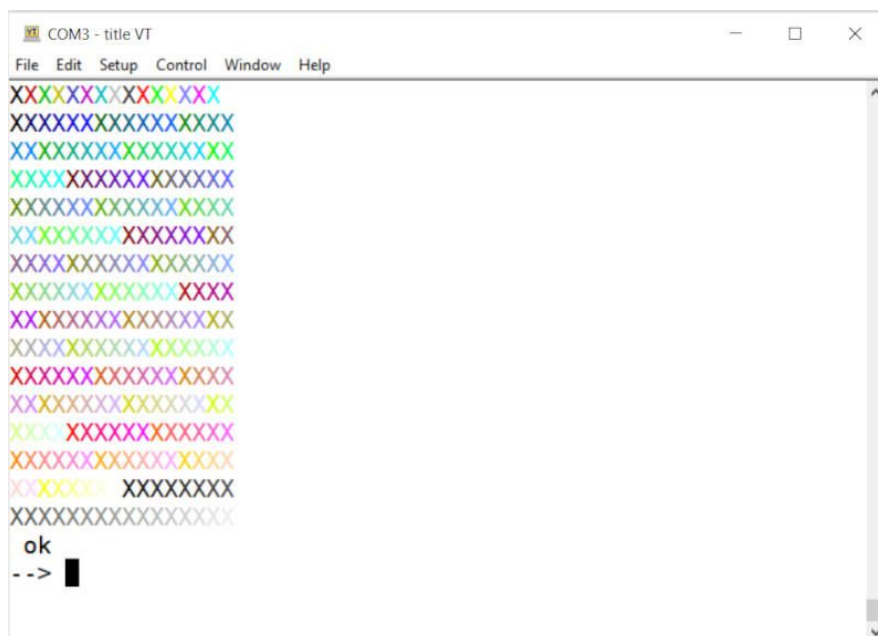
Das Wort **page** löscht den Bildschirm des Terminals und positioniert den Cursor in der oberen linken Ecke des Bildschirms.

# Textfärbung

Sehen wir uns zunächst an, wie man den Text einfärbt :

```
: testFG ( -- )
  page
  16 0 do
    16 0 do
      j 16 * i + fg
      ." X"
    loop
  cr
loop
normal
;
```

Ausführen von **testFG** wird Folgendes angezeigt:



Um die Hintergrundfarben zu testen, gehen wir wie folgt vor:

```
: testBG ( -- )
  page
  16 0 do
    16 0 do
      j 16 * i + bg
      space space
    loop
  cr
loop
normal
;
```

Ausführen von **testBG** wird Folgendes angezeigt :



## Anzeigeposition

Das Terminal ist die einfachste Lösung zur Kommunikation mit ESP32forth. Mit ANSI-Escape-Sequenzen lässt sich die Darstellung von Daten leicht verbessern.

```

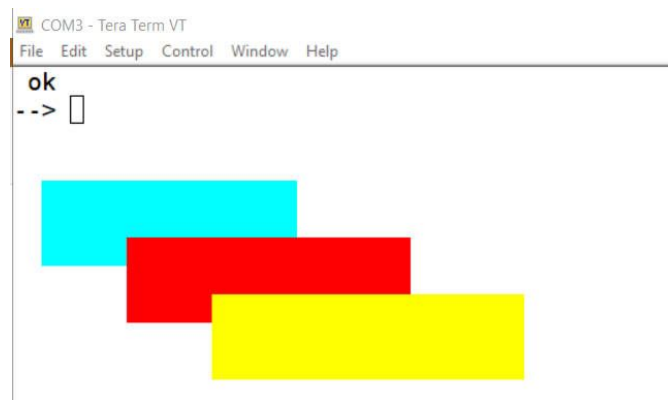
09 constant red
11 constant yellow
14 constant cyan
15 constant whyte
: box { x0 y0 xn yn color -- }
  color bg
  yn y0 - 1+ \ determine height
  0 do
    x0 y0 i + at-xy
    xn x0 - spaces
  loop
  normal
;

: 3boxes ( -- )
  page
  2 4 20 6 cyan box
  8 6 28 8 red box
  14 8 36 10 yellow box
  0 0 at-xy
;

```



Das Ausführen von **3boxes** zeigt Folgendes :



Sie sind nun in der Lage, einfache und effektive Schnittstellen zu erstellen, die die Interaktion mit den von ESP32forth kompilierten FORTH-Definitionen ermöglichen.

# Lokale Variablen mit ESP32Forth

## Einführung

Die FORTH-Sprache verarbeitet Daten hauptsächlich über den Datenstapel. Dieser sehr einfache Mechanismus bietet eine unübertroffene Leistung. Umgekehrt kann es schnell komplex werden, den Datenfluss zu verfolgen. Lokale Variablen bieten eine interessante Alternative.

## Der Fake-Stack-Kommentar

( und ) umrahmten Stapelkommentare aufgefallen sein . Beispiel:

```
\ addiert zwei vorzeichenlose Werte, hinterlässt Summe
\ und Übertrag auf dem Stapel
: um+ (u1 u2 -- Summenübertrag)
  \ hier die Definition
;
```

Hier hat der Kommentar ( **u1 u2 -- sum Carry** ) absolut keine Auswirkung auf den Rest des FORTH-Codes. Das ist reiner Kommentar.

Bei der Vorbereitung einer komplexen Definition besteht die Lösung darin, lokale Variablen zu verwenden, die durch { und } eingerahmt sind. Beispiel :

```
: 2OVER { a b c d }
  a b c d a b
;
```

Wir definieren vier lokale Variablen **abc** und **d** .

Die Wörter { und } ähneln den Wörtern ( und ) , haben aber überhaupt nicht die gleiche Wirkung. Codes zwischen { und } sind lokale Variablen. Die einzige Einschränkung: Verwenden Sie keine Variablennamen, die FORTH-Wörter aus dem FORTH-Wörterbuch sein könnten. Wir hätten unser Beispiel auch so schreiben können:

```
: 2OVER { varA varB varC varD }
  varA varB varC varD varA varB
;
```

Jede Variable nimmt den Wert des Datenstapels in der Reihenfolge ihrer Hinterlegung auf dem Datenstapel an. hier geht 1 in **varA** , 2 in **varB** usw.:

```
--> 1 2 3 4
ok
1 2 3 4 --> 2over
```

```
ok
1 2 3 4 1 2 -->
```

Unser Fake-Stack-Kommentar kann wie folgt vervollständigt werden:

```
: 20VER { varA varB varC varD -- varA varB }
.....
```

Die folgenden Zeichen `--` haben keine Wirkung. Der einzige Sinn besteht darin, unseren Fake-Kommentar wie einen echten Stack-Kommentar aussehen zu lassen.

## Aktion auf lokale Variablen

Lokale Variablen verhalten sich genau wie durch Werte definierte Pseudovariablen.

Beispiel:

```
: 3x+1 { var -- sum }
  var 3 * 1 +
;
```

Hat den gleichen Effekt wie dieser:

```
0 value var
: 3x+1 ( var -- sum )
  to var
  var 3 * 1 +
;
```

In diesem Beispiel wird `var` explizit durch den Wert definiert.

Wir weisen einer lokalen Variablen mit dem Wort `to` oder `+to` einen Wert zu, um den Inhalt einer lokalen Variablen zu erhöhen. In diesem Beispiel fügen wir im Code unseres Wortes eine auf Null initialisierte lokale Variable `result` hinzu:

```
: a+bEXP2 { varA varB -- (a+b)EXP2 }
  0 { result }
  varA varA *      to result
  varB varB *      +to result
  varA varB * 2 * +to result
  result
;
```

Ist es nicht besser lesbar?

```
: a+bEXP2 ( varA varB -- result )
  2dup
  * 2 * >r
  dup *
  swap dup * +
  r> +
;
```

Hier ist ein letztes Beispiel, die Definition des Wortes **um+** , das zwei vorzeichenlose Ganzzahlen addiert und die Summe und den Überlaufwert dieser Summe auf dem Datenstapel belässt:

```
\ Addiert zwei ganze Zahlen ohne Vorzeichen,
\ hinterlässt Summe und Übertrag auf dem Stapel
: um+ { u1 u2 -- sum carry }
  0 { sum }
  cell for
    aft
      u1 $100 /mod to u1
      u2 $100 /mod to u2
      +
      cell 1- i - 8 * lshift +to sum
    then
  next
  sum
  u1 u2 + abs
;
```

Hier ist ein komplexeres Beispiel für das Umschreiben von **DUMP** mithilfe lokaler Variablen:

```
\ lokale Variablen in DUMP:
\ START_ADDR      \ erste Adresse für Dump
\ END_ADDR        \ letzte Adresse für Dump
\ ØSTART_ADDR     \ erste Adresse für Schleife im Dump
\ LINES           \ Anzahl der Zeilen für die Dump-Schleife
\ myBASE          \ aktuelle numerische Basis
internals
: dump ( start len -- )
  cr cr ." --addr--- "
  ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  -----
chars-----"
  2dup + { END_ADDR }          \ letzte Adresse für den Dump
speichern
  swap { START_ADDR }         \ START-Adresse zum Dump
speichern
  START_ADDR 16 / 16 * { ØSTART_ADDR } \ calc. Adresse für
Schleifenstart
  16 / 1+ { LINES }
  base @ { myBASE }           aktuelle Basis speichern
  hex
    \ äußere Schleife
  LINES 0 do
    ØSTART_ADDR i 16 * +      \ calc start address for current
line
    cr <# # # # # [char] - hold # # # # #> type
    space space              \ and display address
    \ first inner loop, display bytes
    16 0 do
      \ calculate real address
      ØSTART_ADDR j 16 * i + +
```

```

        ca@ <# # # #> type space \ display byte in format: NN
    loop
    space
    \ second inner loop, display chars
    16 0 do
        \ calculate real address
        0START_ADDR j 16 * i + +
        \ display char if code in interval 32-127
        ca@      dup 32 < over 127 > or
        if        drop [char] . emit
        else      emit
        then
    loop
    loop
    myBASE base !           \ restore current base
    cr cr
;
forth

```

Die Verwendung lokaler Variablen vereinfacht die Datenmanipulation auf Stacks erheblich. Der Code ist besser lesbar. Beachten Sie, dass es nicht notwendig ist, diese lokalen Variablen vorab zu deklarieren. Es reicht aus, sie bei der Verwendung anzugeben, zum Beispiel: **base @ { myBASE } .**

**WARNUNG:** Wenn Sie lokale Variablen in einer Definition verwenden, verwenden Sie nicht mehr die Wörter **>r** und **r>** , da sonst die Gefahr besteht, dass die Verwaltung lokaler Variablen unterbrochen wird. Schauen Sie sich einfach die Dekompilierung dieser Version von **DUMP** an , um den Grund für diese Warnung zu verstehen:

```

dump cr cr s" --addr--- " type
s" 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  -----chars-----" type
2dup + >R SWAP >R -4 local@ 16 / 16 * >R 16 / 1+ >R base @ >R
hex -8 local@ 0 (do) -20 local@ R@ 16 * + cr
<# # # # # 45 hold # # # # #> type space space
16 0 (do) -28 local@ j 16 * R@ + + CA@ <# # # #> type space 1 (+loop)
0BRANCH rdrop rdrop space 16 0 (do) -28 local@ j 16 * R@ + + CA@ DUP 32 < OVER 127 > OR
0BRANCH DROP 46 emit BRANCH emit 1 (+loop) 0BRANCH rdrop rdrop 1 (+loop)
0BRANCH rdrop rdrop -4 local@ base ! cr cr rdrop rdrop rdrop rdrop rdrop ;

```

# Datenstrukturen für ESP32forth

## Präambel

ESP32forth ist eine 32-Bit-Version der FORTH-Sprache. Diejenigen, die FORTH seit seinen Anfängen praktizieren, haben mit 16-Bit-Versionen programmiert. Diese Datengröße wird durch die Größe der auf dem Datenstapel abgelegten Elemente bestimmt. Um die Größe der Elemente in Bytes herauszufinden, müssen Sie das Wort `cell` ausführen. Führen Sie dieses Wort für ESP32forth aus:

```
cell . \ zeigt 4 an
```

Der Wert 4 bedeutet, dass die Größe der auf dem Datenstapel platzierten Elemente 4 Bytes oder  $4 \times 8 \text{ Bits} = 32 \text{ Bits}$  beträgt.

Bei einer 16-Bit-Forth-Version stapelt `cell` den Wert 2. Wenn Sie eine 64-Bit-Version verwenden, stapelt `cell` ebenfalls den Wert 8.

## Tabellen in FORTH

Beginnen wir mit relativ einfachen Strukturen: Tabellen. Wir werden nur ein- oder zweidimensionale Arrays diskutieren.

### Eindimensionales 32-Bit-Datenarray

Dies ist die einfachste Art von Tabelle. Um eine Tabelle dieses Typs zu erstellen, verwenden wir das Wort **create**, gefolgt vom Namen der zu erstellenden Tabelle:

```
create temperatures
    34 ,    37 ,    42 ,    36 ,    25 ,    12 ,
```

In dieser Tabelle speichern wir 6 Werte: 34, 37...12. Um einen Wert abzurufen, verwenden Sie einfach das Wort `@`, indem Sie die nach `temperatures` gestapelte Adresse mit dem gewünschten Offset erhöhen:

```
temperatures    \ empile addr
0 cell *        \ calcule décalage 0
+               \ ajout décalage à addr
@ .             \ affiche 34

temperatures    \ empile addr
1 cell *        \ calcule décalage 1
+               \ ajout décalage à addr
@ .             \ affiche 37
```

Wir können den Zugangscode auf den gewünschten Wert faktorisieren, indem wir ein Wort definieren, das diese Adresse berechnet:

```
: temp@ ( index -- value )
  cell * temperatures + @
;
0 temp@ . \ affiche 34
2 temp@ . \ affiche 42
```

Sie werden feststellen, dass für n in dieser Tabelle gespeicherte Werte, hier 6 Werte, der Zugriffsindex immer im Intervall [0..n-1] liegen muss.

## Tabellendefinitionswörter

So erstellen Sie eine Wortdefinition für eindimensionale Ganzzahl-Arrays:

```
: array ( comp: -- | exec: index -- addr )
  create
  does>
    swap cell * +
;
array myTemps
  21 , 32 , 45 , 44 , 28 , 12 ,
0 myTemps @ . \ zeigt 21
5 myTemps @ . \ zeigt 12
```

**array** -Variante zu erstellen, um unsere Daten kompakter zu verwalten :

```
: arrayC (comp: -- | exec: index -- addr)
erstellen
tut>
+
;
arrayC myCTime
21c, 32c, 45c, 44c, 28c, 12c,
0 myCTemps c@. \anzeige 21
5 myCTemps c@. \anzeige 12
```

Bei dieser Variante werden die gleichen Werte auf viermal weniger Speicherplatz gespeichert.

## Lesen und schreiben Sie in eine Tabelle

Es ist durchaus möglich, ein leeres Array mit n Elementen zu erstellen und Werte in dieses Array zu schreiben und zu lesen:

```
: arrayC ( comp: -- | exec: index -- addr )
  create
  does>
    +
;
arrayC myCTemps
```



```

    21 c,    32 c,    45 c,    44 c,    28 c,    12 c,
0 myCTemps c@ .    \ display 21
5 myCTemps c@ .    \ display 12

```

In unserem Beispiel enthält das Array 6 Elemente. Mit ESP32forth steht genügend Speicherplatz zur Verfügung, um deutlich größere Arrays, beispielsweise mit 1.000 oder 10.000 Elementen, zu verarbeiten. Es ist einfach, mehrdimensionale Tabellen zu erstellen. Beispiel für ein zweidimensionales Array:

```

63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
    SCR_WIDTH SCR_HEIGHT * allot          \ réserve 63 * 16 octets
mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill \ remplis avec 'space'

```

Hier definieren wir eine zweidimensionale Tabelle namens **mySCREEN**, die einen virtuellen Bildschirm mit 16 Zeilen und 63 Spalten darstellt.

Reservieren Sie einfach einen Speicherplatz, der das Produkt der Abmessungen X und Y der zu verwendenden Tabelle ist. Sehen wir uns nun an, wie man dieses zweidimensionale Array verwaltet:

```

: xySCRaddr { x y -- addr }
    SCR_WIDTH y *
    x + mySCREEN +
;
: SCR@ ( x y -- c )
    xySCRaddr c@
;
: SCR! ( c x y -- )
    xySCRaddr c!
;
char X 15 5 SCR!    \ speichert char X am Hals 15 Zeile 5
15 5 SCR@ emit      \ wird angezeigt

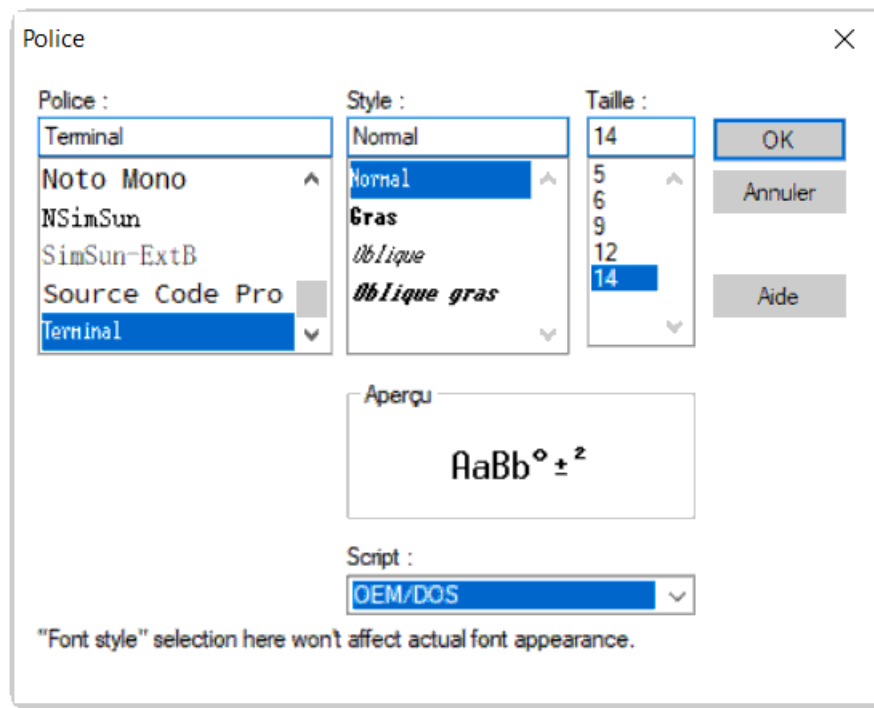
```

## Praktisches Beispiel für die Verwaltung eines virtuellen Bildschirms

Bevor wir mit unserem Beispiel zur Verwaltung eines virtuellen Bildschirms fortfahren, sehen wir uns an, wie man den Zeichensatz des TERA TERM-Terminals ändert und anzeigt.

TERA TERM starten:

- Klicken Sie in der Menüleiste auf **Setup**
- Wählen Sie **Font** und **Font...**
- Konfigurieren Sie die Schriftart unten:



So zeigen Sie die Tabelle der verfügbaren Zeichen an:

```
: tableChars ( -- )
  base @ >r hex
  128 32 do
    16 0 do
      j i + dup . space emit space space
    loop
  cr
  16 +loop
  256 160 do
    16 0 do
      j i + dup . space emit space space
    loop
  cr
  16 +loop
  cr
  r> base !
;
tableChars
```

Hier ist das Ergebnis der Ausführung von **tableChars**:





In unserem virtuellen Bildschirmbeispiel zeigen wir, dass die Verwaltung eines zweidimensionalen Arrays eine konkrete Anwendung hat. Unser virtueller Bildschirm ist zum Schreiben und Lesen zugänglich. Hier zeigen wir unseren virtuellen Bildschirm im Terminalfenster an. Diese Anzeige ist alles andere als effizient. Auf einem echten OLED-Bildschirm kann es aber deutlich schneller gehen.

## Management komplexer Strukturen

ESP32forth verfügt über das Strukturvokabular. Der Inhalt dieses Vokabulars ermöglicht die Definition komplexer Datenstrukturen.

Hier ist eine triviale Beispielstruktur :

```
structures
struct YMDHMS
  ptr field >year
  ptr field >month
  ptr field >day
  ptr field >hour
  ptr field >min
  ptr field >sec
```

Hier definieren wir die YMDHMS-Struktur. Diese Struktur verwaltet die Zeiger **>year** **>month** **>day** **>hour** **>min** und **>sec** .

des **YMDHMS** -Wortes besteht darin, die Zeiger in der komplexen Struktur zu initialisieren und zu guptieren. So werden diese Zeiger verwendet :

```
create DateTime
  YMDHMS allot

2022 DateTime >year  !
  03 DateTime >month !
  21 DateTime >day   !
```

```

22 DateTime >hour    !
36 DateTime >min     !
15 DateTime >sec     !

: .date ( date -- )
  >r
  ." YEAR: " r@ >year    @ . cr
  ." MONTH: " r@ >month  @ . cr
  ." DAY: " r@ >day      @ . cr
  ." HH: " r@ >hour      @ . cr
  ." MM: " r@ >min       @ . cr
  ." SS: " r@ >sec       @ . cr
  r> drop
;

DateTime .date

```

Wir haben das Wort **DateTime** definiert, das eine einfache Tabelle aus 6 aufeinanderfolgenden 32-Bit-Zellen ist. Der Zugriff auf jede Zelle erfolgt über den entsprechenden Zeiger. Wir können den zugewiesenen Speicherplatz unserer **YMDHMS**-Struktur neu definieren, indem wir das Wort **i8** verwenden, um auf Bytes zu verweisen:

```

structures
struct cYMDHMS
  ptr field >year
  i8 field >month
  i8 field >day
  i8 field >hour
  i8 field >min
  i8 field >sec

create cDateTime
  cYMDHMS allot

2022 cDateTime >year    !
03 cDateTime >month    c!
21 cDateTime >day      c!
22 cDateTime >hour     c!
36 cDateTime >min      c!
15 cDateTime >sec      c!

: .cDate ( date -- )
  >r
  ." YEAR: " r@ >year    @ . cr
  ." MONTH: " r@ >month  c@ . cr
  ." DAY: " r@ >day      c@ . cr
  ." HH: " r@ >hour      c@ . cr
  ." MM: " r@ >min       c@ . cr
  ." SS: " r@ >sec       c@ . cr
  r> drop
;

cDateTime .cDate      \ zeigt:
\ YEAR: 2022
\ MONTH: 3

```

```

\   DAY: 21
\   HH: 22
\   MM: 36
\   SS: 15

```

In dieser **cYMDHMS**-Struktur haben wir das Jahr im 32-Bit-Format beibehalten und alle anderen Werte auf 8-Bit-Ganzzahlen reduziert. Wir sehen im .cDate-Code, dass die Verwendung von Zeigern einen einfachen Zugriff auf jedes Element unserer komplexen Struktur ermöglicht....

## Definition von Sprites

Wir haben zuvor einen virtuellen Bildschirm als zweidimensionales Array definiert. Die Dimensionen dieses Arrays werden durch zwei Konstanten definiert. Erinnerung an die Definition dieses virtuellen Bildschirms:

```

63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
    SCR_WIDTH SCR_HEIGHT * allot
mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill

```

Der Nachteil dieser Programmiermethode besteht darin, dass die Dimensionen in Konstanten, also außerhalb der Tabelle, definiert werden. Interessanter wäre es, die Abmessungen der Tabelle in die Tabelle einzubetten. Dazu definieren wir eine an diesen Fall angepasste Struktur :

```

structures
struct cARRAY
    i8 field >width
    i8 field >height
    i8 field >content

create myVscreen    \ definit un ecran 8x32 octets
    32 c,            \ compile width
    08 c,            \ compile height
    myVscreen >width c@
    myVscreen >height c@ * allot

```

Um ein Software-Sprite zu definieren, geben wir ganz einfach diese Definition weiter:

```

: sprite: ( width height -- )
    create
        swap c, c, \ compile width et height
    does>
;
2 1 sprite: blackChars
    $db c, $db c,
2 1 sprite: greyChars
    $b2 c, $b2 c,
blackChars >content 2 type    \ affiche contenu du sprite blackChars

```

Voici comment définir un sprite 5 x 7 octets:

```
5 7 sprite: char3
    $20 c, $db c, $db c, $db c, $20 c,
    $db c, $20 c, $20 c, $20 c, $db c,
    $20 c, $20 c, $20 c, $20 c, $db c,
    $20 c, $db c, $db c, $db c, $20 c,
    $20 c, $20 c, $20 c, $20 c, $db c,
    $db c, $20 c, $20 c, $20 c, $db c,
    $20 c, $db c, $db c, $db c, $20 c,
```

Um das Sprite von einer xy-Position im Terminalfenster aus anzuzeigen, reicht eine einfache Schleife:

```
: .sprite { xpos ypos sprAddr -- }
    sprAddr >height c@ 0 do
        xpos ypos at-xy
        sprAddr >width c@ i * \ calculate offset in sprite datas
        sprAddr >content + \ calculate real address for line n
in sprite datas
    sprAddr >width c@ type \ display line
    1 +to ypos \ increment y position
loop
;

0 constant blackColor
1 constant redColor
4 constant blueColor
10 02 char3 .sprite
redColor fg
16 02 char3 .sprite
blueColor fg
22 02 char3 .sprite
blackColor fg
cr cr
```

Ergebnis der Anzeige unseres Sprites:

```
COM3 - Tera Term VT
File Edit Setup Control Window Help
ok
-->
ok
-->
ok
-->
ok
-->
ok
-->
ok
--> blackColor fg
ok
```



Ich hoffe, der Inhalt dieses Kapitels hat Ihnen einige interessante Ideen gegeben, die Sie gerne teilen möchten ...



## Reale Zahlen mit ESP32forth

Wenn wir die Operation **1 3 /** in der FORTH-Sprache testen, ist das Ergebnis 0.

Es ist nicht überraschend. Grundsätzlich verwendet ESP32forth über den Datenstack nur 32-Bit-Ganzzahlen. Ganzzahlen bieten bestimmte Vorteile:

- Geschwindigkeit der Verarbeitung;
- Ergebnis von Berechnungen ohne Driftgefahr bei Iterationen;
- für fast alle Situationen geeignet.

Auch bei trigonometrischen Berechnungen können wir eine Tabelle mit ganzen Zahlen verwenden. Erstellen Sie einfach eine Tabelle mit 90 Werten, wobei jeder Wert dem Sinus eines Winkels multipliziert mit 1000 entspricht.

Aber auch ganze Zahlen haben Grenzen:

- unmögliche Ergebnisse für einfache Divisionsberechnungen, wie unser 1/3-Beispiel;
- erfordert komplexe Manipulationen, um physikalische Formeln anzuwenden.

Seit Version 7.0.6.5 enthält ESP32forth Operatoren, die sich mit reellen Zahlen befassen.

Reelle Zahlen werden auch Gleitkommazahlen genannt.

## Die echten mit ESP32forth

Um reelle Zahlen zu unterscheiden, müssen sie mit dem Buchstaben „e“ enden:

```
3          \ push 3 on the normal stack
3e         \ push 3 on the real stack
5.21e f.   \ display 5.210000
```

Es ist das Wort **F.** Dadurch können Sie eine reelle Zahl anzeigen, die sich oben auf dem reellen Stapel befindet.

## Echte Zahlengenauigkeit mit ESP32forth

Mit dem Wort **set-precision** können Sie die Anzahl der Dezimalstellen angeben, die nach dem Dezimalpunkt angezeigt werden sollen. Sehen wir uns das mit der Konstante **pi an** :

```
pi f.      \ display 3.141592
4 set-precision
```

```
pi f.      \ display 3.1415
```

Die Grenzgenauigkeit für die Verarbeitung reeller Zahlen mit ESP32forth beträgt sechs Dezimalstellen :

```
12 set-precision  
1.987654321e f.      \ display 1.987654668777
```

Wenn wir die Anzeigegenauigkeit reeller Zahlen unter 6 reduzieren, werden die Berechnungen immer noch mit einer Genauigkeit von 6 Nachkommastellen durchgeführt.

## Reale Konstanten und Variablen

Eine echte Konstante wird mit dem Wort **fconstant** definiert :

```
0.693147e fconstant ln2  \ natural logarithm of 2
```

**fvariable** definiert :

```
fvariable intensity  
170e 12e F/ intensity SF!  \ I=P/U  ---  P=170w  U=12V  
intensity SF@ f.          \ display 14.166669
```

ACHTUNG: Alle reellen Zahlen durchlaufen den **Stapel reeller Zahlen** . Bei einer realen Variablen durchläuft nur die Adresse den Datenstapel, die auf den realen Wert zeigt.

Das Wort **SF!** speichert einen reellen Wert an der Adresse oder Variablen, auf die seine Speicheradresse zeigt. Durch die Ausführung einer echten Variablen wird die Speicheradresse auf dem klassischen Datenstapel platziert.

Das Wort **SF@** stapelt den realen Wert, auf den seine Speicheradresse zeigt.

## Arithmetische Operatoren für reelle Zahlen

ESP32Forth verfügt über vier arithmetische Operatoren **F+ F- F\* F/** :

```
1.23e 4.56e F+ f.  \ display 5.790000    1.23-4.56  
1.23e 4.56e F- f.  \ display -3.330000    1.23-4.56  
1.23e 4.56e F* f.  \ display 5.608800     1.23*4.56  
1.23e 4.56e F/ f.  \ display 0.269736     1.23/4.56
```

ESP32forth hat auch diese Worte:

- **1/F** berechnet den Kehrwert einer reellen Zahl;
- **fsqrt** berechnet die Quadratwurzel einer reellen Zahl.

```
5. 1/F f. \display 0,200000 1/5  
5. fsqrt f. \ display 2.236068 sqrt(5)
```

## Mathematische Operatoren für reelle Zahlen

ESP32forth verfügt über mehrere mathematische Operatoren:

- **F\*\*** erhöht einen echten  $r\_val$  auf die Potenz  $r\_exp$
- **FATAN2** berechnet den Winkel im Bogenmaß aus der Tangente.
- **FCOS** ( $r1 - r2$ ) Berechnet den Kosinus eines Winkels, ausgedrückt im Bogenmaß.
- **FEXP** ( $\ln-r - r$ ) berechnet den reellen Wert, der  $e^{EXP\ r}$  entspricht
- **FLN** ( $r - \ln-r$ ) berechnet den natürlichen Logarithmus einer reellen Zahl.
- **FSIN** ( $r1 - r2$ ) berechnet den Sinus eines Winkels, ausgedrückt im Bogenmaß.
- **FSINCOS** ( $r1 - rcos\ rsin$ ) berechnet den Kosinus und Sinus eines Winkels, ausgedrückt im Bogenmaß.

Einige Beispiele :

```
2e 3e f** f.    \ display 8.000000
2e 4e f** f.    \ display 16.000000
10e 1.5e f** f.  \ display 31.622776

4.605170e FEXP F.    \ display 100.000018

pi 4e f/
FSINCOS f. f.    \ display 0.707106 0.707106
pi 2e f/
FSINCOS f. f.    \ display 0.000000 1.000000
```

## Logische Operatoren für reelle Zahlen

Mit ESP32forth können Sie auch Logiktests an realen Daten durchführen:

- **F0<** ( $r -- fl$ ) testet, ob eine reelle Zahl kleiner als Null ist.
- **F0=** ( $r - fl$ ) zeigt wahr an, wenn der reelle Wert Null ist.
- **f<** ( $r1\ r2 -- fl$ )  $fl$  ist wahr, wenn  $r1 < r2$ .
- **f<=** ( $r1\ r2 -- fl$ )  $fl$  ist wahr, wenn  $r1 \leq r2$ .
- **f<>** ( $r1\ r2 -- fl$ )  $fl$  ist wahr, wenn  $r1 \neq r2$ .
- **f=** ( $r1\ r2 -- fl$ )  $fl$  ist wahr, wenn  $r1 = r2$ .
- **f>** ( $r1\ r2 -- fl$ )  $fl$  ist wahr, wenn  $r1 > r2$ .
- **f>=** ( $r1\ r2 -- fl$ )  $fl$  ist wahr, wenn  $r1 \geq r2$ .

## Ganzzahlige ↔ reelle Transformationen

ESP32forth verfügt über zwei Wörter, um ganze Zahlen in reelle Zahlen umzuwandeln und umgekehrt:









- **F>S** (r -- n) wandelt eine reelle Zahl in eine ganze Zahl um. Belassen Sie den ganzzahligen Teil im Datenstapel, wenn die reelle Zahl Dezimalteile hat.
- **S>F** (n -- r: r) wandelt eine ganze Zahl in eine reelle Zahl um und überträgt diese reelle Zahl auf den reellen Stapel.

Beispiel :

```
35 S>F
F.    \ display 35.000000
3.5e F>S .    \ display 3
```

# Installieren der OLED-Bibliothek für SSD1306

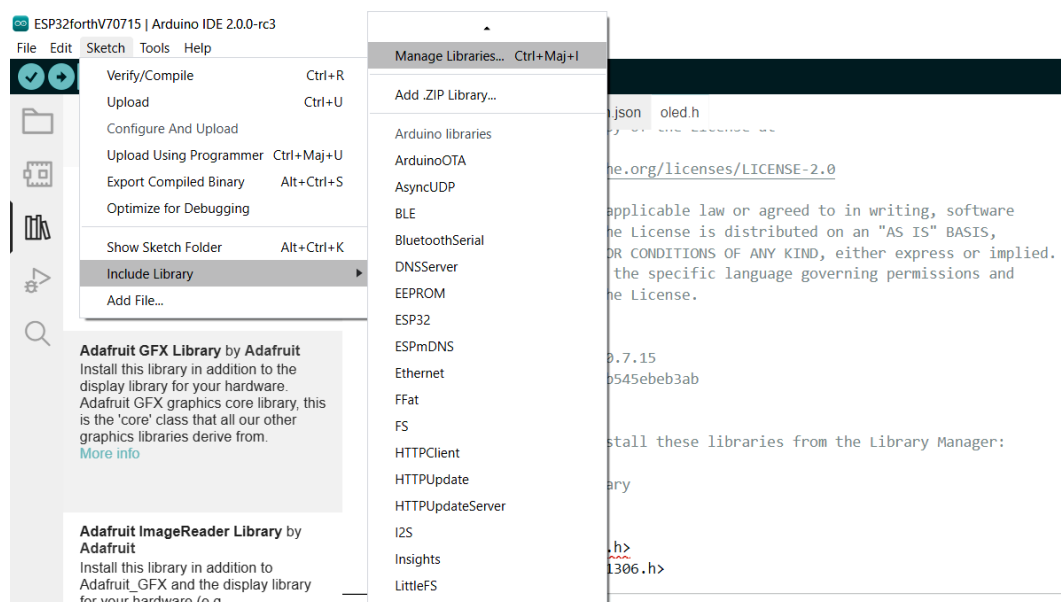
Seit ESP32forth Version 7.0.7.15 sind die Optionen im **optional** Ordner verfügbar:

Téléchargements > ESP32forth-7.0.7.15(1).zip > ESP32forth > optional		
Nom	Type	
 assemblers.h	Fichier H	
 camera.h	Fichier H	
 interrupts.h	Fichier H	
 oled.h	Fichier H	
 README-optional.txt	Document texte	
 rmt.h	Fichier H	
 serial-bluetooth.h	Fichier H	
 spi-flash.h	Fichier H	

Um das **oled** Vokabular zu erhalten, kopieren Sie die Datei **oled.h** in den Ordner, der die Datei **ESP32forth.ino** enthält.

Starten Sie dann ARDUINO IDE und wählen Sie die neueste **ESP32forth.ino**-Datei aus.

Wenn die OLED-Bibliothek nicht installiert wurde, klicken Sie in der ARDUINO IDE auf *Sketch* und wählen Sie *Include Library* und dann *Manage Libraries*.



Suchen Sie in der linken Seitenleiste nach der Bibliothek **Adafruit SSD1306 by Adafruit**.

Sie können nun den Sketch compilieren und uploaden. Klicken Sie dazu auf *Sketch* und dann *Upload*.

Wenn der Sketch auf das ESP32-Board hochgeladen ist, starten Sie das TeraTerm-Terminal. Überprüfen Sie, ob das **oled**-Vokabular vorhanden ist :

```
oled vlist \ display:
OledInit SSD1306_SWITCHCAPVCC SSD1306_EXTERNALVCC WHITE BLACK OledReset
HEIGHT WIDTH OledAddr OledNew OledDelete OledBegin OledHOME OledCLS OledTextc
OledPrintln OledNumln OledNum OledDisplay OledPrint OledInvert OledTextsize
OledSetCursor OledPixel OledDrawL OledCirc OledCircF OledRect OledRectF
OledRectR OledRectRF oled-builtins
```

# Ressourcen

## Auf Englisch

- **ESP32forth** Seite verwaltet von Brad NELSON, dem Erfinder von ESP32forth. Dort finden Sie alle Versionen (ESP32, Windows, Web, Linux...)  
<https://esp32forth.appspot.com/ESP32forth.html>

•

## Auf Französisch

- **ESP32 Forth** Website in zwei Sprachen (Französisch, Englisch) mit vielen Beispielen  
<https://esp32.arduino-forth.com/>

## GitHub

- **Ueforth** Ressourcen verwaltet von Brad NELSON. Enthält alle Forth- und C-Sprach Quelldateien für ESP32forth  
<https://github.com/flagxor/ueforth>
- **ESP32forth** Quellcodes und Dokumentation für ESP32forth. Ressourcen verwaltet von Marc PETREMANN  
<https://github.com/MPETREMANN11/ESP32forth>
- **ESP32forthStation** Ressourcen verwaltet von Ulrich HOFFMAN. Eigenständiger Forth-Computer mit LillyGo TTGO VGA32-Einplatinencomputer und ESP32forth  
<https://github.com/uho/ESP32forthStation>
- **ESP32Forth** Ressourcen verwaltet von F. J. RUSSO  
<https://github.com/FJRusso53/ESP32Forth>
- **esp32forth-addons** Ressourcen verwaltet von Peter FORTH  
<https://github.com/PeterForth/esp32forth-addons>
- **Esp32forth-org** Code-Repository für Mitglieder der Forth2020- und ESP32forth-Gruppen  
<https://github.com/Esp32forth-org>

•

## Index

Anzeigeposition.....	26	normal.....	26	Textfarben.....	26
bg.....	26	oled.....	48	to.....	31
constant.....	24	page.....	26	value.....	
F.....	44	pi.....	44	.....	24
F>S.....	47	S>F.....	47	Variable.....	24
fconstant.....	45	set-precision.....	44	{.....	30
fg.....	26	SF!.....	45	}.....	30
FORTH-Wort.....	10	SF@.....	45	+to.....	31
fvariable.....	45	struct.....	39		
Kommentar.....	30	structures.....	39, 41		