



University of  
East London

# Data Communication on Automated Systems

Author  
Chris Liourtas

*A thesis submitted in partial fulfillment of the requirements for the degree of  
Cybersecurity and Networks*

Degree Name  
BSc (Hons) Cyber Security and Networks

UNIVERSITY NAME  
University of East London

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Εισαγωγή</b>                                  | <b>3</b>  |
| 1.1      | Συστήματα Αυτοματισμού . . . . .                 | 4         |
| 1.2      | Embedded Systems . . . . .                       | 6         |
| 1.2.1    | Βασικές αρχές υπολογιστικών συστημάτων . . . . . | 6         |
| 1.2.2    | ESP ως ενσωματωμένο σύστημα . . . . .            | 9         |
| 1.2.3    | Μεταγλώττιση και Τύποι . . . . .                 | 9         |
| <b>2</b> | <b>Υλοποίηση C</b>                               | <b>11</b> |
| 2.1      | FreeRTOS . . . . .                               | 13        |
| <b>3</b> | <b>Υλοποίηση Rust</b>                            | <b>15</b> |
| 3.1      | Λογική και Design Patterns . . . . .             | 17        |
| 3.1.1    | Γραμμική Λογική . . . . .                        | 17        |
| 3.1.2    | Builder Pattern . . . . .                        | 19        |
| 3.1.3    | Asynchronous Rust . . . . .                      | 22        |
| 3.1.4    | HAL Patterns . . . . .                           | 29        |
| 3.2      | Γράφοντας το Project . . . . .                   | 30        |
| 3.2.1    | HC-SR04 Driver . . . . .                         | 31        |
| 3.2.2    | MQTT Driver . . . . .                            | 34        |
| 3.2.3    | Main Program . . . . .                           | 40        |

## **Abstract**

This is an extremely cool abstract.

## 1 Εισαγωγή

Τα συστήματα αυτοματισμού είναι πλέον μέρος της καθημερινότητας μας. Συναντάμε συνέχεια, χωρίς απαραίτητα να το συνειδητοποιούμε κάποιου είδους σύστημα το οποίο είναι προγραμματισμένο να εκτελεί μια διεργασία με βάση ορισμένες συνθήκες του εξωτερικού του περιβάλλοντος. Το παραπάνω έχει μεγάλο αντίκτυπο στον κόσμο το οποίο γίνεται ξεκάθαρο με την ραγδαία εξέλιξη του IoT (Internet of Things). Περιτριγυρίζομαστε από συσκευές οι οποίες λειτουργούν ως συστήματα αυτοματισμού αλλά παράλληλα επικοινωνούν με το Cloud.

Λόγω των περιορισμένων πόρων των συσκευών IoT έχουμε υιοθετήσει συγκεκριμένα standards για τις γλώσσες προγραμματισμού και κατά συνέπεια τα οικοσυστήματα που χρησιμοποιούμε για την κατασκευή τους. Συνήθως αυτά τα standards είναι ένας συμβιβασμός μεταξύ ευκολίας προγραμματισμού και πλήρους βελτιστοποίησης των πόρων της συσκευής. Με ευκολία προγραμματισμού εννοούμε για παράδειγμα την αυτόματη διαχείριση μνήμης (Garbage Collection). Δεν είναι σπάνιο πλέον να βρίσκεται κάπου ένας μικροελεγκτής ο οποίος είναι ικανός να τρέξει ένα runtime της Python ή της Java διευκολύνοντας κατά μεγάλο βαθμό την διαδικασία υλοποίησης και της ασφάλειας όσο αναφορά την μνήμη. Έχουν όμως προφανώς συνέπειες στην απόδοση του προγράμματος, τόσο μεγάλες που ανάλογα με τις ανάγκες του έργου ενδέχεται να μην είναι ρεαλιστικές επιλογές. Μάλιστα οι επιλογές τέτοιου τύπου γλωσσών έχουν συχνά άλλα προβλήματα. Σε μεγάλα projects δεν είναι τόσο καλή πρακτική η χρήση dynamic typing της Python καθώς ανοίγουν την πόρτα για πολλά logic bombs ή άλλα λογικά σφάλματα. Από την άλλη στην Java, αν και παρέχει στατικά types ο τρόπος διαχείρισης σφαλμάτων είναι, στην καλύτερη περίπτωση, περιοριστικός. Για αυτούς τους λόγους αρχικά φαίνεται ότι η C και η C++ κυριαρχούν στην πλειοψηφία των καταστάσεων σε αυτόν τον τομέα, δίνοντας πλήρης έλεγχο στον προγραμματιστή επιτρέποντας του να γράψει τον καλύτερο δυνατό κώδικα. Με την σειρά του αυτό ανοίγει άλλες πόρτες σφαλμάτων και πιθανών κινδύνων ασφάλειας. Σε μια πασίγνωστη παρουσίαση της Microsoft υποστηρίζετε ότι το 70% των ευαλωτών ασφάλειας τους οφείλονται σε προβλήματα διαχείρισης μνήμης [2] [3] ενώ η σχετικά πρόσφατη δημοσίευση της αμερικάνικης κυβέρνησης προωθεί την χρήση των ασφαλών γλωσσών προγραμματισμού [1] για αυτόν ακριβώς τον λόγο. Η Rust, με το μοντέλο Aliasing XOR Mutability, είναι μια ξεκάθαρη λύση και αν και δεν είναι σε καμία περίπτωση τέλεια αναφέρετε πολύ συχνά ως μια μοντέρνα εναλλακτική της C/C++.

Ο σκοπός της εκάστοτε εργασίας είναι να εξετάσουμε το παραπάνω ζήτημα, να κατανοήσουμε και να συγκρίνουμε τρία διαφορετικά οικοσυστήματα (Arduino, ESP-IDF, Embassy) ένα από τα οποία βασίζετε στην Rust (Embassy). Θα χρησιμοποιήσουμε τον μικροελεγκτή ESP32C6-DEVKITC-1, τον οποίο από εδώ και πέρα θα αναφέρουμε ως ESP και θα συγκρίνουμε τα αποτελέσματα με βάση την υπολογιστική ισχύς (CPU Usage), αποθηκευτικό χώρο και μνήμη κατά την εκτέλεση που χρησιμοποιεί το κάθε ένα. Παράλληλα θα εξεταστεί το λεγόμενο "ease of development" αλλά όχι άμεσα, προτιμώντας να περιγράψουμε πλήρως την λειτουργικότητα κάθε οικοσυστήματος χωρίς προσωπικές απόψεις καθώς είναι η διευκόλυνση είναι προφανώς υποκειμενική.

Με σκοπό την πλήρη διαφάνεια των αποτελεσμάτων και γενικά της πληροφoρίας που βρίσκεται στο παρών κείμενο αξίζει να σημειωθεί ότι αναφέρονται πολλές πτυχές

της πληροφορικής εδώ. Από την πρακτική αρχιτεκτονική υπολογιστών μέχρι και την θεωρητική μηχανή Turing αναφέρονται όλα με την πρόθεση ότι ο αναγνώστης είναι υπεύθυνος να εκλάβει την χρησιμότητα, ανάγκη ή εγκυρότητα των παρακάτω αναφορών όπως εκείνος πιστεύει. Είναι εκτός από τα πλαίσια αυτής της εργασίας η πλήρης περιγραφή ορισμένων κλάδων και κατά συνέπεια πολλές λεπτομέρειες αφαιρούντε, προφανώς όμως παρέχετε η κατάλληλη βιβλιογραφία η οποία αναπαριστά αυτήν την γνώση πολύ καλύτερα από ότι θα μπορούσε το παρών γραπτό.

## 1.1 Συστήματα Αυτοματισμού

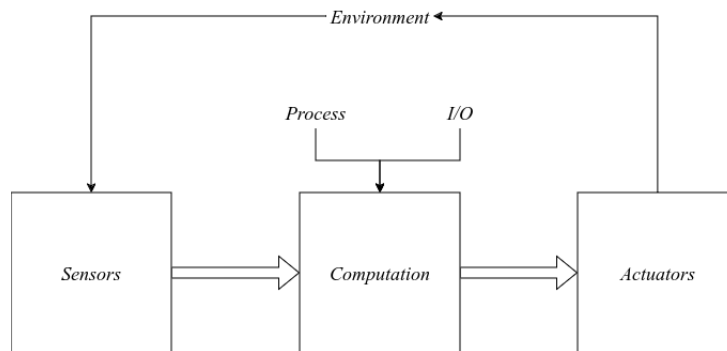


Figure 1: Αναπαράσταση βασικών στοιχείων συστημάτων αυτοματισμού.

Ένα σύστημα αυτοματισμού είναι ένα σύστημα που αποτελείτε από τρία βασικά στοιχεία:

1. Τους σένσορες ή αισθητήρες, που λαμβάνουν δεδομένα από το εξωτερικό τους περιβάλλον και παράγουν ένα όσο το δυνατόν πιο αξιόπιστο αναλογικό ή ψηφιακό σήμα.
2. Ένα υπολογιστικό σύστημα, για παράδειγμα έναν μικροελεγκτή το οποίο μπορεί να χρησιμοποιήσει το σήμα των αισθητήρων για υπολογισμούς, επεξεργασία ή I/O.
3. Το στοιχείο δράσης ή ενεργοποιητής, που με βάση των υπολογισμών του υπολογιστικού συστήματος ενεργοποιούνται ή παραμένουν αδρανής. Αυτό το στοιχείο αποτελεί την έξοδο του συστήματος αυτοματισμού και είναι αυτό το οποίο έχει συνήθως αντίκτυπο στο εξωτερικό του περιβάλλον.

Το υπολογιστικό σύστημα μπορεί να χρησιμοποιηθεί για να παίρνει αποφάσεις με βάση των τιμών που δέχεται από τους αισθητήρες και εκεί φαίνεται η πραγματική χρησιμότητα των αυτόματων συστημάτων. Πολλές φορές με βάση την λήψη αποφάσεων (υπολογισμών) το αποτέλεσμα του στοιχείου δράσης χρειάζεται να επιστρέψει στην είσοδο του συστήματος, για παράδειγμα εάν χρειάζεται να ρυθμιστεί ένας σένσορας

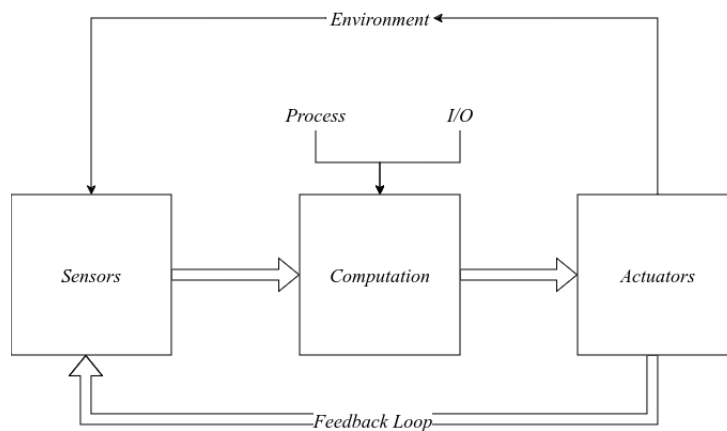


Figure 2: Σύστημα κλειστού βρόχου.

δυναμικά. Σε αυτήν την περίπτωση το σύστημα λέγεται σύστημα ανατροφοδότησης ή αλλιώς σύστημα κλειστού βρόχου (βλ. [Σχήμα 2](#)).

Το υπολογιστικό σύστημα στην δικιά μας περίπτωση είναι ο ESP. Έστω ότι έχουμε και έναν αισθητήρα απόστασης HC-SR04. Μπορούμε να θεωρήσουμε ότι ο ενεργοποιητής είναι η ενσωματωμένη κεραία του WiFi που κάθε λίγο χρονικό διάστημα επικοινωνεί ασύρματα με ένα Router για να στείλει δεδομένα στο Cloud.

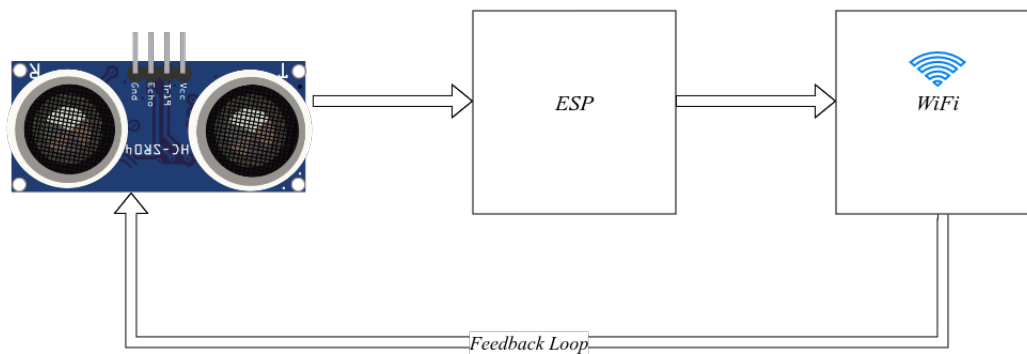


Figure 3: Αυτόματο σύστημα παρούσας περίπτωσης.

Οι σένσορες είναι τις περισσότερες φορές απλά ολοκληρωμένα κυκλώματα (integrated circuits) τα οποία κάνουν κάποιον είδους μετρήσεις στο εξωτερικό περιβάλλον. Ο αισθητήρας απόστασης αποτελεί καλό παράδειγμα καθώς η λειτουργία του είναι εύκολα κατανοητή. Μόλις το trigger pin λάβει σήμα HIGH (5V) για τουλάχιστον 10us ο σένσορας ενεργοποιείται στέλνοντας 8 κύματα υπερήχου συχνότητας 40kHz. Η αντανάκλαση του ήχου έχει ως αποτέλεσμα το echo pin να μεταβεί με την σειρά του σε κατάσταση HIGH. Εφόσον η ταχύτητα του ήχου είναι σταθερή και έχουμε

την ικανότητα να μετρήσουμε τον χρόνο για τον οποίο το echo pin διατηρείτε σε κατάσταση υψηλής τάσης μπορούμε να μετρήσουμε την απόσταση μέσω  $velocity = \frac{\Delta x}{\Delta t}$ . Για να τον συνδέσουμε στην ίδια διάταξη κυκλώματος με τον μικροελεγκτή πρέπει να συνδέσουμε τα pins VCC, Trig, Echo, Gnd στα αντίστοιχα GPIO pins όπως φαίνεται παρακάτω (βλ. Σχήμα 4).

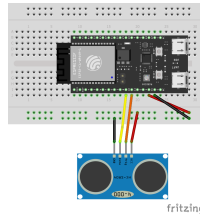


Figure 4: Παρούσα διάταξη.

## 1.2 Embedded Systems

Ο πιο σύγχρονος τρόπος προγραμματισμού παίρνει μέρος συνήθως σε αυτό που αποκαλούμε user-space ενός λειτουργικού συστήματος. Στο user-space το OS (Operating System) αναλαμβάνει ένα πολύ μεγάλο μέρος της πολυπλοκότητας του προγραμματισμού. Η δημιουργία εικονικής μνήμης, η δυνατότητα παράλληλου προγραμματισμού, η διαχείριση μνήμη σορού (heap memory) είναι όλα προβλήματα που αντιμετωπίζονται σχεδόν αυτόματα πλέον. Επίσης μέσω του λειτουργικού συστήματος αποκρύβονται οι λεπτομέρειες του hardware, τα πάντα σχεδόν προσφέρονται ως ψηφιακά API (Application Programming Interface), τα οποία χτίζονται το ένα πάνω στο άλλο. Με κάθε πρόσθεση επιπέδου αφαίρεσης χάνετε λίγο παραπάνω η ικανότητα πλήρους διαχείρισης του hardware με το θετικό ότι σε γενικές γραμμές κάνει την ζωή του προγραμματιστή πιο εύκολη. Είναι ξεκάθαρο λοιπόν ότι τα abstraction layers είναι χρήσιμα, αλλά είναι δίκιοπο μαχαίρι. Από την μια πλευρά η υλοποίηση των περισσότερων εφαρμογών γίνεται απίστευτα πιο εύκολη. Από την άλλη, τι γίνεται όταν πραγματικά χρειαζόμαστε να επικοινωνήσουμε με το hardware που έχουμε μπροστά μας; Πολλές φορές υπάρχουν συστήματα που δεν μπορούν ή δεν χρειάζεται να τρέξουν ένα ολόκληρο operating system. Πολλές φορές δεν έχουμε την επιλογή να τρέξουμε κώδικα πάνω σε ένα abstraction. Για παράδειγμα, συστήματα που βρίσκονται σε απλές συσκευές όπως θερμοστάτες και ρολόγια όχι μόνο δεν είναι απαραίτητο να τρέχουν OS αλλά πολλές φορές δεν είναι βέλτιστο. Αναφερόμαστε στα συστήματα που δεν χρησιμοποιούν παραδοσιακά OS ως embedded systems διότι τα βρίσκουμε συχνά ενσωματωμένα σε μεγαλύτερες συσκευές.

### 1.2.1 Βασικές αρχές υπολογιστικών συστημάτων

Συνήθως προγραμματίζουμε ένα embedded system στο κύριο μηχάνημα μας (host machine) το οποίο είναι λογικά διαφορετικής αρχιτεκτονικής. Αυτό σημαίνει ότι πρέπει να μετατρέψουμε τον πηγαίο κώδικα της high level γλώσσας που χρησιμοποιούμε στην αντίστοιχη γλώσσα που καταλαβαίνει ο εξωτερικός επεξεργαστής. Αυτή η

διαδικασία λέγεται cross-compiling. Μάλιστα είναι απαραίτητο να δώσουμε στο πρόγραμμα τις απαραίτητες τοποθεσίες στην μνήμη στην οποία θα εκτελεί εντολές ή θα αποθηκεύει στατικά δεδομένα, αυτή είναι δουλειά του linker. Αναλυτικότερα ο πηγαίος κώδικας, μαζί με ότι βιβλιοθήκες χρησιμοποιούνται περνάνε από τον compiler και πρώτα προεπεξεργάζονται και ενώνονται σε μια δομή. Μεταφράζονται αργότερα στην assembly που καταλαβαίνει το target μηχανήμα και ο assembler τα μετασχηματίζει σε ένα ή περισσότερα Relocatable Object αρχεία, δηλαδή συλλογές από bytes που αντιπροσωπεύουν τον κώδικα στην target γλώσσα. Έπειτα τα object files περνάνε από τον linker ο οποίος ανάλογα με τις ρυθμίσεις του οργανώνει το αρχείο και προσθέτει απαραίτητα δεδομένα για να δημιουργήσει ένα εκτελέσιμο για το target μηχανήμα πρόγραμμα, για παράδειγμα ένα ELF32 (βλ. Σχήμα 5).

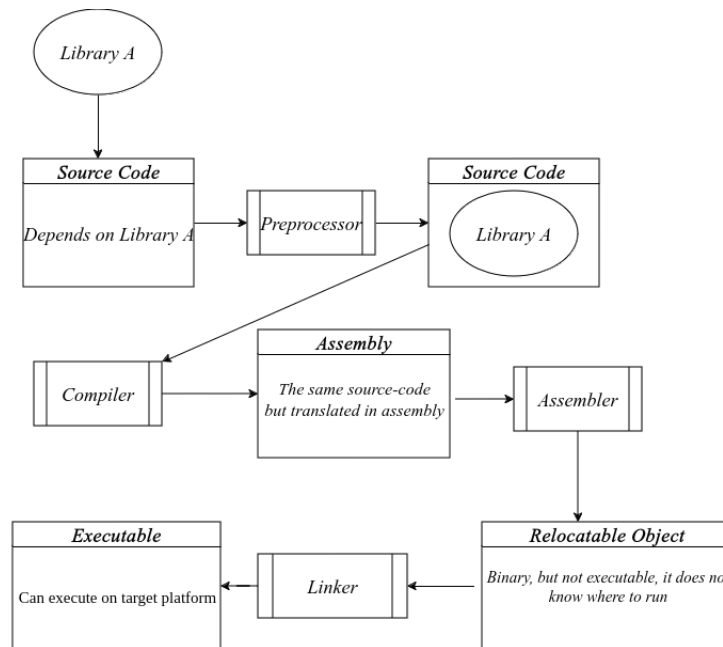


Figure 5: Στάδια μεταγλώττισης.

Η πραγματικότητα είναι ότι πλέον, ειδικά για μικρά projects, η διαδικασία των σταδίων μεταγλώττισης είναι αυτοματοποιημένη και γίνονται όλα μέσω του οικοσυστήματος όπως θα δούμε αργότερα. Όμως εξακολουθεί να είναι ιδιαίτερα χρήσιμη η γνώση τους και απασχολεί πολύ τον κόσμο της πληροφορικής ακόμα και σήμερα, για λόγους βελτιστοποίησης και debugging.

Εφόσον έχουμε δημιουργήσει ένα εκτελέσιμο αρχείο πρέπει με κάποιο τρόπο να το στείλουμε στην συσκευή. Συνήθως χρησιμοποιούμε κάποιο σειριακό πρωτόκολλο, τύπου USB το οποίο εσωτερικά συνδέεται με κάποιου είδους μη πτητικής μνήμης για να το αποθηκεύσει (βλ. Σχήμα 6). Αυτό έχει ως αποτέλεσμα το πρόγραμμα να διατηρείτε ανεξαρτήτως αν αργότερα κάνουμε επανεκκίνηση ή τερματίσουμε την συσκευή.



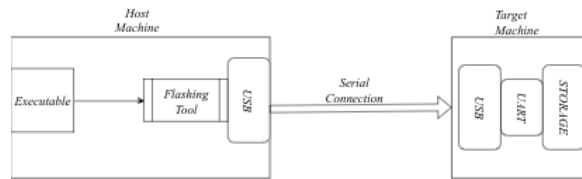


Figure 6: Development workflow

Όταν ενεργοποιήσουμε την μηχανή τα απαραίτητα στοιχεία από την μη πτητική μνήμη αντιγράφονται στην προσωρινή και ο καταχωρητής program counter του επεξεργαστή δείχνει στην διεύθυνση μνήμης στην οποία βρίσκεται η πρώτη εντολή του πρώτου προγράμματος που θα τρέξει. Συνήθως αυτό είναι κάποιου είδους boot-loader, αλλά στα ενσωματωμένα συστήματα δεν είναι πάντα αυτή η περίπτωση και τρέχει το firmware που έχουμε γράψει κατευθείαν.

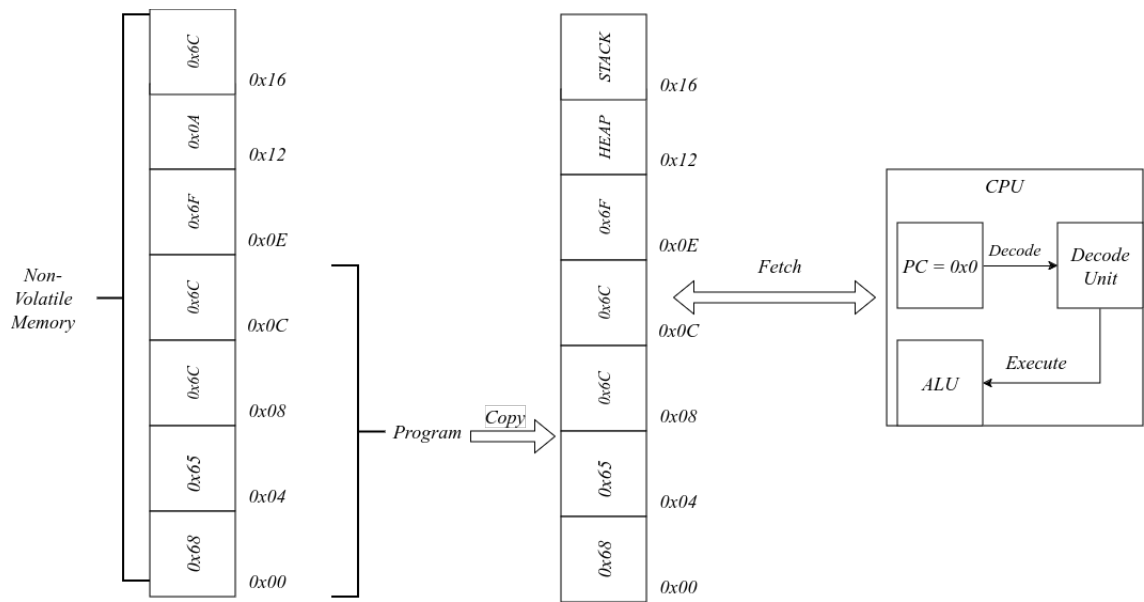


Figure 7: Κύκλος εντολών μηχανής

Αξίζει να σημειωθεί ότι δεν είναι πάντα ξεκάθαρο το πως η μη πτητική μνήμη αντιγράφεται στην προσωρινή εξ αρχώς, μπορεί οι μνήμες να έχουν ενσωματωμένο firmware και να επικοινωνούν με σύνδεση τύπου I2C ή UART ή πιο λογικό να είναι ο bootloader του επεξεργαστή σε μια ανεξάρτητη ROM (Read Only Memory) σταθερής διεύθυνσης και μεγέθους και να είναι αυτός υπεύθυνος για την επικοινωνία. Παράλληλα, υπάρχουν πολλοί καταχωρητές οι οποίοι διαχειρίζονται την κατάσταση (state) του επεξεργαστή και κατά συνέπεια ολόκληρου του συστήματος. Ο program counter αναφέρετε διότι διατηρείτε σε πρακτικά όλες τις αρχιτεκτονικές.

Είναι σημαντικό να γνωρίζουμε επίσης ότι μια εντολή δεν εκτελείτε κατευθείαν. Όπως αναπαριστά το παραπάνω διάγραμμα, πρώτα ο επεξεργαστής διαβάζει την εντολή από την μνήμη (fetch). Έπειτα την αποσπά για να την κατανοήσει (decode). Θυμόμαστε ότι έχουμε αποθηκεύσει το εκτελέσιμο αρχείο, δηλαδή αυτό που είναι κατανοητό στην μηχανή, σε δυαδική μορφή, όχι σε γραμματοσειρά που είναι κατανοητή στους ανθρώπους. Για παράδειγμα οι εντολές της αρχιτεκτονικής RISC-V 32bit έχουν σταθερό μέγεθος και αναπαριστούν μια αριθμητική τιμή  $x \in [0, 2^{31}] \subseteq \mathbb{N}_0$ . Ο αριθμός αυτός στην δυαδική του μορφή, μπορεί να διαχωριστεί σε  $n \in \mathbb{N}$  θραύσματα, συνήθως τα λέμε nibbles, μέσο bit operations και εξηγούν στον επεξεργαστή τι πρέπει να κάνει. Στην πολύ συχνή εντολή `add t0, t1, t2`, στην οποία προσθέτουμε την τιμή του καταχωρητή `t1` με την τιμή του καταχωρητή `t2` και την αποθηκεύουμε στον `t0`, ένα unit του επεξεργαστή κάνει τις παρακάτω πράξεις για να την αποκωδικοποιήσει.

$Bits_{0-31}$  is stored in memory and then enters a unit capable of decoding RISC-V instructions:  
 $Bits_{0-31} = \text{add } t0, t1, t2 = 0x007302b3_{(16)}$   
 $Bits_{0-6} = Bits_{0-31} \& 0x7F_{(16)} \rightarrow \text{opcode}$   
 $Bits_{7-11} = (Bits_{0-31} \gg 7_{(10)}) \& 0x1F_{(16)} \rightarrow \text{destination register}$   
 $Bits_{12-14} = (Bits_{0-31} \gg 12_{(10)}) \& 0x7_{(16)} \rightarrow \text{operation (addition)}$   
 $Bits_{15-19} = (Bits_{0-31} \gg 15_{(10)}) \& 0x1F_{(16)} \rightarrow \text{source register 1}$   
 $Bits_{20-24} = (Bits_{0-31} \gg 20_{(10)}) \& 0x1F_{(16)} \rightarrow \text{source register 2}$   
 $Bits_{25-31} = (Bits_{0-31} \gg 25_{(10)}) \& 0x7F_{(16)} \rightarrow \text{addition variant}$

Η λογική αποκωδικοποίησης είναι ίδια σε πρακτικά όλες τις αρχιτεκτονικές όμως το μέγεθος μιας εντολής δεν είναι σε όλες σταθερό. Τέλος, παίρνουν μέρος οι αριθμητικές πράξεις ή το I/O με βάση τα αποτελέσματα του decoding (execute).

Η καθυστέρηση που έρχεται ως επίπτωση των παραπάνω πράξεων μπορεί να φαίνεται στην αρχή σαν μεγάλη υπερφόρτωση του υπολογιστικού συστήματος και πράγματι για παλαιότερες αρχιτεκτονικές ήταν. Το λεγόμενο Instruction Cycle όμως (Fetch  $\rightarrow$  Decode  $\rightarrow$  Execute) που μόλις αναλύσαμε έχει οριστεί για έναν πολύ συγκεκριμένο λόγο. Πρόκειται για την τεχνική υλοποίησης "διοχέτευση" (pipelining). Ουσιαστικά ο επεξεργαστής εκμεταλλεύεται την καθυστέρηση που παίρνει μέρος υλοποιώντας μια δομή, για τους σκοπούς μας ένα pipeline που μπορεί να δεχθεί μέχρι  $\nu \in \mathbb{N}$  εντολές. Κάθε φορά που αλλάζει το state μιας εντολής προχωράει στο pipeline και προσθέτετε μια καινούργια εντολή στην αρχικοποιημένη της κατάσταση. Αυτό δίνει την ψευδαίσθηση παραλληλισμού στο επίπεδο εντολών του επεξεργαστή και είναι ένας από τους πολλούς λόγους που σύγχρονα προγράμματα μπορούν να τρέχουν τόσο γρήγορα χωρίς απαραίτητα να τρέχουν πάνω σε ένα runtime.

Η περίπτωση του ESP32C6 είναι ότι αν και είναι πανίσχυρος για μικροελεγκτής ανάλογα με το framework περιέχει μινιμαλιστικά, στην καλύτερη περίπτωση, abstractions.

## 1.2.2 ESP ως ενσωματωμένο σύστημα

## 1.2.3 Μεταγλώττιση και Τύποι

|                                  |                           |                |                |                |
|----------------------------------|---------------------------|----------------|----------------|----------------|
| $v = 4$                          | <i>Clock Cycle Number</i> |                |                |                |
| <i>Instruction Number</i>        | <i>1</i>                  | <i>2</i>       | <i>3</i>       | <i>4</i>       |
| <i>Instruction<sub>i</sub></i>   | <i>Fetch</i>              | <i>Decode</i>  | <i>Execute</i> | <i>Done</i>    |
| <i>Instruction<sub>i+1</sub></i> | <i>Waiting</i>            | <i>Fetch</i>   | <i>Decode</i>  | <i>Execute</i> |
| <i>Instruction<sub>i+2</sub></i> | <i>Waiting</i>            | <i>Waiting</i> | <i>Fetch</i>   | <i>Decode</i>  |
| <i>Instruction<sub>i+3</sub></i> | <i>Waiting</i>            | <i>Waiting</i> | <i>Waiting</i> | <i>Fetch</i>   |

Figure 8: Αναπαράσταση βασικών στοιχείων συστημάτων αυτοματισμού.

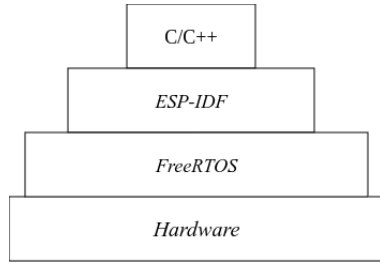


Figure 9: ESP-IDF Abstraction Layers

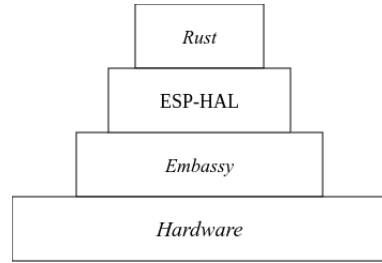


Figure 10: Embassy Abstraction Layers

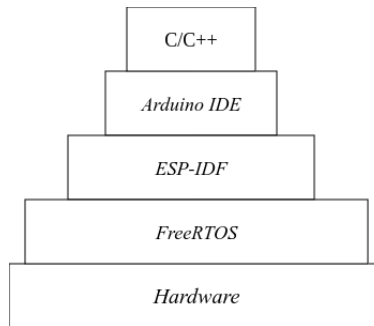


Figure 11: Arduino Abstraction Layers

## 2 Υλοποίηση C

Η υλοποίηση C στο περιβάλλον ESP-IDF είναι ιδιαίτερα εύκολη καθώς οι περισσότερες βιβλιοθήκες είναι ενσωματωμένες στο build system που προσφέρει η Espressif. Συγκεκριμένα οι βιβλιοθήκες που μας ενδιαφέρουν και είναι έτοιμες να γίνουν include μετά το setup είναι:

1. Η υλοποίηση FreeRTOS.
2. Η υλοποίηση του MQTTV5.
3. Η βιβλιοθήκη για WiFi.

Μας λείπει δηλαδή:

1. Μια βιβλιοθήκη για διαχείριση του HCSR04.
2. Μια βιβλιοθήκη για δημιουργία JSON.

Προτεινόμενος τρόπος να ξεκινήσουμε ένα project είναι να αντιγράψουμε ένα template από το repository του esp-idf.

```
1 git clone https://github.com/espressif/esp-idf
2 mkdir my-project
3 cp $ESP_IDF_PATH/examples/$MY_TEMPLATE my-project/
```

Έπειτα κάνουμε source τα python scripts που μας βοηθάνε να κάνουμε build, flash και monitor. Τα python scripts μόλις γίνουν source γίνονται ορατά στον χρήστη και συνήθως χρησιμοποιούμε το `idf.py` ως το κύριο πρόγραμμα για να τα τρέξουμε.

```
1 . $ESP_IDF_PATH/export.sh
```

Μπορούμε πλέον να δημιουργήσουμε τα απαραίτητα αρχεία στα οποία θα γραφτεί ο κώδικας. Η εικόνα 12 παρουσιάζει την δομή των αρχείων του project χωρίς τις βιβλιοθήκες.

Το `sdkconfig` είναι μια συλλογή από macros τα οποία περνάνε ως flags στον compiler. Η χρησιμότητα τους είναι κυρίως στην εξατομίκευση του προγράμματος και του μικροελεγκτή. Μέσο αυτών μπορούμε να επιλέξουμε την ταχύτητα του ρολογιού του επεξεργαστή, να ενεργοποιήσουμε ή να απενεργοποιήσουμε hooks ή να φτιάξουμε τα δικά μας `#defines` για διαφορετικές υλοποιήσεις του ίδιου project. Προφανώς θα μπορούσαμε να κάνουμε όλα τα παραπάνω προγραμματιστικά αλλά το `sdkconfig` προσφέρει ένα ωραίο interface μέσω του `idf.py menuconfig`. Χρησιμοποιώντας αυτό το interface εμείς θέτουμε `configUSE_IDLE_HOOK` ίσο με 1 για να μπορούμε να μετρήσουμε το idle time του επεξεργαστή.

Το `CMakeLists` είναι απλώς το build script. Ο τρόπος με τον οποίο το περιβάλλον ESP-IDF κατηγοριοποιεί ένα project είναι μέσω των components, ουσιαστικά εξωτερικές βιβλιοθήκες. Οτιδήποτε δεν είναι component θεωρείται κύριος πηγαίος κώδικας και βρίσκεται στον φάκελο με το όνομα του project. Η κατανόησή του παραπάνω είναι σημαντική διότι υπάρχουν δύο (τουλάχιστον) `CMakeLists`. Το build script που βρίσκεται

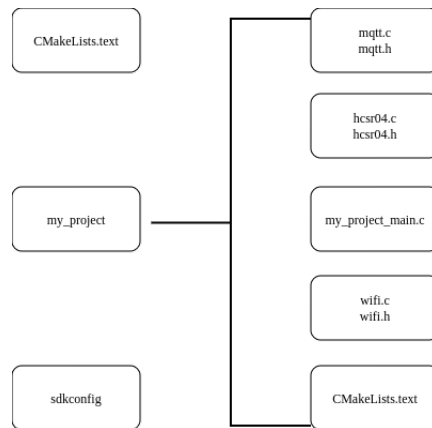


Figure 12: Project filestructure

στην ρίζα του project και το build script που βρίσκεται στον φάκελο που βρίσκεται ο πηγαίος κώδικας. Το εξωτερικό build script είναι υπεύθυνο για την ενσωμάτωση components και ορίζει το όνομα του project. Για παράδειγμα εφόσον θέλουμε τουλάχιστον δύο έξτρα βιβλιοθήκες:

```

1 cmake_minimum_required(VERSION 3.16)
2 set(EXTRA_COMPONENT_DIRS ./components/)
3 include($ENV{IDF_PATH}/tools/cmake/project.cmake)
4 project(my-project)

```

Δημιουργούμε το component directory και εκεί βάζουμε το source code των libraries. Στην δικιά μας περίπτωση χρησιμοποιούμε το esp-idf-lib το οποίο περιέχει ένα driver για τον HCSR04 και το CJSON για να δημιουργήσουμε JSON strings.

Το εσωτερικό CMakeLists περιέχει όλα τα αρχεία που έχουν πηγαίο κώδικα:

```

1 idf_component_register(SRCS "my_project_main.c"
2                        "hcsr04.c"
3                        "wifi.c"
4                        "mqtt.c"
5                        INCLUDE_DIRS "")

```

## 2.1 FreeRTOS

Στην εισαγωγή αναφέραμε τον κύκλο ζωής εντολών, τον τρόπο που συνήθως ο επεξεργαστής εκτελεί instructions. Η ιδέα αυτή της ψευδαίσθησης του παραλληλισμού είναι αρκετά σύνθητες σε εφαρμογές λειτουργικών συστημάτων. Το FreeRTOS, συγκεκριμένα η υλοποίηση του ESP-IDF, δεν διαφέρει καθόλου.

Πρόκειται για ένα όσο το δυνατόν πιο μινιμαλιστικό λειτουργικού σύστημα. Αποτελείτε απλώς από:

1. Έναν scheduler.
2. Tasks που τρέχουν με βάση τον scheduler.
3. Hooks που τρέχουν όταν παίρνουν μέρος events του λειτουργικού συστήματος.
4. Αντικείμενα που επιτρέπουν την επικοινωνία μεταξύ tasks.

Υποστηρίζονται πολλοί scheduling αλγόριθμοι αλλά αναφερόμαστε κυρίως στο default που χρησιμοποιεί το ESP-IDF. Ο default scheduler είναι κατά κύριο λόγο προεκτοπιστικός (preemptive) το οποίο σημαίνει ότι μια διεργασία μπορεί να διακοπεί οποιαδήποτε στιγμή από κάποια άλλη υψηλότερης ή ίσης προτεραιότητας. Κάθε task έχει σταθερή προτεραιότητα (fixed priority). Όταν δύο tasks έχουν ίση προτεραιότητα χρησιμοποιούνται time slices για να αλλάξει η κατάσταση τους.

Όπως φαίνεται στο σχήμα 13 τα tasks έχουν 4 καταστάσεις:

1. Ready, μπορούν να καλεστούν για να τρέξουν.
2. Running, τρέχουν αυτήν την χρονική στιγμή.
3. Blocked, δεν τρέχουν αυτήν την χρονική στιγμή.
4. Suspended, το task δεν είναι διαθέσιμο στον scheduler.

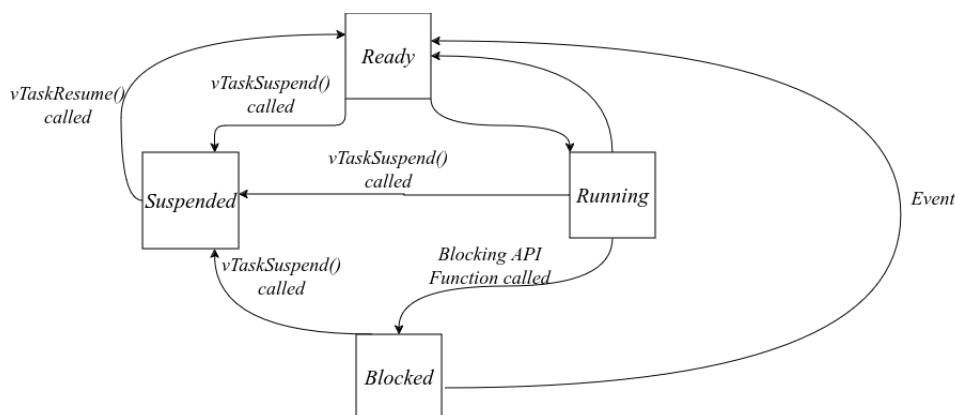


Figure 13: Καταστάσεις διεργασιών

Τα states Ready και Running αλλάζουν συνεχώς ανάλογα με τις προτεραιότητες και τα ενδιάμεσα blocks. Όσο αναφορά το Suspended state σπάνια το διαχειριζόμαστε άμεσα σαν χρήστες και κατά κύριο λόγο καλείτε όταν κάνουμε `vTaskDelete(NULL)` για να "σβήσουμε" το task που τρέχει αυτήν την στιγμή.

Για να δημιουργήσουμε ένα task που ο scheduler μπορεί να δει χρησιμοποιούμε:

```
1 BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,  
2     const char * const pcName,  
3     const configSTACK_DEPTH_TYPE uxStackDepth,  
4     void *pvParameters,  
5     UBaseType_t uxPriority,  
6     TaskHandle_t *pxCreatedTask  
7     );
```

Η συνάρτηση φαίνεται πολύ πιο περίπλοκη από ότι είναι την πραγματικότητα:

1. Το `TaskFunction_t` είναι απλώς ένας δείκτης σε μια συνάρτηση που επιστρέφει `void` και έχει `void*` ως παράμετρο. Αυτή η συνάρτηση είναι ο κώδικας του task που θέτει ο χρήστης.
2. Το `const char* const` είναι προφανές, δεν έχει ιδιαίτερη χρησιμότητα και υπάρχει κυρίως για λόγους debugging.
3. Το `const configSTACK_DEPTH_TYPE uxStackDepth` είναι το μέγεθος του stack σε words (όχι bytes). Κάθε task έχει ξεχωριστό stack και ο χρήστης είναι υπεύθυνος να προβλέψει το πιο είναι το ελάχιστο απαραίτητο μέγεθος για λόγους βελτιστοποίησης.
4. Το `void*` είναι οι παράμετροι που το task δέχεται. Δεν μπορούμε να προσθέσουμε stack-based values εδώ επειδή κάποια στιγμή θα βγούνε από το scope και θα καταλήξουμε σε ένα null pointer dereference error. Προφανώς αυτό ισχύει σε οποιαδήποτε τιμή που δείχνει σε μη προσβάσιμη μνήμη.
5. Το `UBaseType_t` είναι ένα `unsigned int` που αντιπροσωπεύει την προτεραιότητα.
6. Το `TaskHandle_t` είναι προαιρετικό αλλά επίσης χρήσιμο για debugging, είναι ένα typedef ενός δείκτη οπότε αν δεν θέλουμε να το ορίσουμε το θέτουμε ως NULL.

### 3 Υλοποίηση Rust

Για την υλοποίηση σε Rust είναι αρχικά σημαντικό να ξεκαθαρίσουμε ορισμένα πράγματα. Ο μεταγλωττιστής της Rust παρέχει στην πραγματικότητα δύο ξεχωριστές γλώσσες, την ασφαλής (safe) Rust και την μη ασφαλής (unsafe) Rust. Η safe Rust όπως υπονοεί το όνομα είναι το μέρος της γλώσσας που χρησιμοποιεί στατική ανάλυση για να αποτρέψει σύνηθες σφάλματα που αφορούν την διαχείριση μνήμης. Η unsafe Rust ενεργοποιείται με την λέξη κλειδί `unsafe` και επιτρέπει ενέργειες που ενδεχομένως μπορούν να διακόψουν (halt) την εκτέλεση του προγράμματος, όπως να κάνουμε dereference έναν σκέτο (raw) δείκτη. Η σύνηθες υλοποίηση της λογικής της Rust που εξηγείτε στα παρακάτω κεφάλαια διατηρείται και στις δύο εκδοχές της γλώσσας και όπως θα δούμε μπορούν προφανώς να συνυπάρχουν.

Σε γενικές γραμμές όμως η safe Rust είναι ένα υποσύνολο της unsafe Rust. Μάλιστα διαισθητικά τουλάχιστον μπορούμε να πούμε πως η safe Rust είναι υποσύνολο της C. Δηλαδή έστω ότι όλα τα πιθανά προγράμματα εκφράσιμα από μια παραδοσιασκή μηχανή Turing βρίσκονται στο σύνολο  $U$ , όλα τα πιθανά προγράμματα εκφράσιμα από το specification της C  $C = U$  και όλα τα πιθανά προγράμματα της safe Rust  $SR \subseteq C$ . Συνεχίζοντας ισχύει ότι για την unsafe rust  $UR = C = U, SR \subseteq UR$ .

Έχοντας πει αυτά γίνεται ξεκάθαρη η πολυπλοκότητα του μεταγλωττιστή. Σε σύγκριση με την C η οποία μπορεί να μεταφραστεί σε γλώσσα μηχανής με μόνο ένα ενδιάμεσο στάδιο η Rust χρειάζεται το λιγότερο 3 στην μόνη υλοποίηση που υπάρχει. Σε αυτά τα τέσσερα στάδια, δεν μετριοούνται καν οι ενδιάμεσες αναπαραστάσεις του LLVM. Στο σχήμα 14 η κάθε ενδιάμεση αναπαράσταση μεταλλάζει τον πηγαίο κώδικα σε όλο και μια πιο απλή γλώσσα που στο τέλος η μηχανή μπορεί να κατανοήσει.

1. HIR: Είναι ο πηγαίος κώδικας αλλά desugared, δηλαδή οντότητες όπως είναι ο χρόνος ζωής μεταβλητών (lifetimes) ή οι τύποι των μεταβλητών εκφράζονται πλέον ρητά. Ένα feature σημαντικού ενδιαφέροντος για εμάς που γίνεται desugared σε αυτό το στάδιο είναι το `async`.
2. THIR: Σε αυτήν την αναπαράσταση η απλοποίηση του κώδικα γίνεται πιο εμφανής. Σύνηθες τύποι μετατρέπονται σε πιο απλούς (primitives), τα generics μεταφράζονται στις στατικές τους μορφές και οι μέθοδοι μετατρέπονται σε συναρτήσεις.
3. MIR: Ο λόγος ύπαρξης του THIR είναι για να γίνεται πιο απλή η μετάφραση σε αυτό το στάδιο. Είναι ουσιαστικά ένα Control-Flow Graph (CFG) που αναπαριστά την ροή εκτέλεσης του προγράμματος. Εδώ γίνεται η στατική ανάλυση απαραίτητη για τον Borrow Checker, τον εγκέφαλο του στατικού ελεγκτή της Rust που είναι υπεύθυνος για το specification που πρέπει να ακολουθεί το πρόγραμμά μας.

Η τελική μετατροπή είναι του LLVM το οποίο είναι ένα third-party framework που χρησιμοποιείτε ως παγκόσμια γλώσσα μηχανής. Είναι μια υπερβολικά απλή και μινιμαλιστική γλώσσα προγραμματισμού βασισμένη σε SSA (Static Single Assignment) η οποία μεταφράζεται στις πιο γνωστές αρχιτεκτονικές. Η SSA είναι ιδανική για να αναπαραστήσει την Rust σε χαμηλότερο επίπεδο κυρίως για την ιδιότητα που



έχει όσο αναφορά το code-locality. Εδώ είναι καλή στιγμή να αναφερθούμε στο specification της Rust, τους κανόνες δηλαδή που ένα πρόγραμμα πρέπει να ακολουθεί για να γίνει compile.

1. Το πρώτο βασικό χαρακτηριστικό της γλώσσας είναι ότι τα σύμβολα είναι σταθερές by default, για να ορίσουμε μεταβλητές πρέπει να χρησιμοποιήσουμε την λέξη κλειδί `mut`.

```
1 let x = 5;
2 x = 6; //Error 'x' is not mutable
3
4 let mut y = 5;
5 y = 6; // Valid, since its 'mut'
6
```

2. Κάθε τιμή στη Rust έχει ένα και μόνο ένα owner. Όταν ο owner της τιμής βγαίνει εκτός scope η τιμή αποδεσμεύεται αυτόματα από τη μνήμη.

```
1 fn main() {
2     let s = String::from("hello");
3     //Create inner scope
4     {
5         let t = s; //'t' now owns 's'
6     }
7     //Inner scope ends here
8     println!("{}", s); // Error: 's' has been
9     ↪ consumed/moved
10 }
```

3. Κάθε τιμή μπορεί να έχει άπειρες αμετάβλητες αναφορές `&T` ή μόνο μια μεταβλητή αναφορά `&mut T`.

```
1 let mut x = 10;
2 let y = &x; // Immutable reference
3 let z = &x; // Valid, can have infinite
4 println!("{}", y, z);
5
6 let w = &mut x; // Error: Cant '&mut T' while '&T'
7     ↪ exist
```

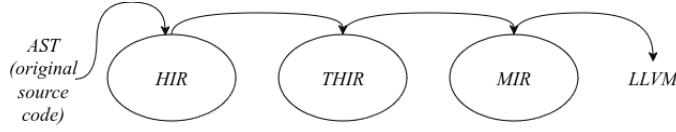


Figure 14: Rusts Intermediate Representations.

### 3.1 Λογική και Design Patterns

#### 3.1.1 Γραμμική Λογική

Η λογική είναι από τις σημαντικότερες αρχές της Rust καθώς αποτυπώνει με αυστηρό μαθηματικό τρόπο τους υποκείμενους μηχανισμούς του μεταγλωττιστή. Μάλιστα έχουμε ήδη αναφέρει ότι όλες οι γλώσσες συμπίπτουν στη επιστήμη της λογικής με τον έναν ή τον άλλον τρόπο, σε καμία περίπτωση όμως δεν είναι απαραίτητη για την κατανόηση τους. Γιατί λοιπόν είναι εξαίρεση η Rust? Η λέξη κλειδί είναι η αυστηρότητα της γλώσσας και η απαίτηση του μεταγλωττιστή ένα πρόγραμμα να είναι σωστό με βάσει προορισμένους κανόνες. Το σύστημα λογικής που χρησιμοποιείτε για να εκφράσει την εγκυρότητα ενός προγράμματος είναι ένα υποσύνολο της γραμμικής λογικής (Linear Logic).

Στις συνηθισμένες λογικές/γλώσσες προγραμματισμού χρησιμοποιούμε  $\phi \vdash \psi$  εννοώντας ότι εφόσον ξέρουμε την εγκυρότητα του  $\phi$  μπορούμε να συμπεράνουμε την εγκυρότητα του  $\psi$ . Η σχέση αυτή διατηρείται για πάντα, ανεξαρτήτως της κατάστασης, μόλις αποδείξουμε ότι  $\psi$  είναι αληθές επειδή το  $\phi$  είναι αληθές δεν μπορούμε ξαφνικά να αλλάξουμε γνώμη,  $\phi \rightarrow true$  για πάντα αλλιώς αλλάζει και το αποτέλεσμα του  $\psi$ . Αυτό έχει ως αποτέλεσμα ορισμένα συμπεράσματα να είναι πολύ πιθανά, για παράδειγμα έστω  $A \rightarrow B, A \rightarrow \Gamma$ , τότε ένα ενδεχόμενο συμπέρασμα μπορεί να είναι  $A \vdash B \wedge \Gamma$ . Το παραπάνω παράδειγμα αν και συμβατό με την πραγματικότητα δεν μοντελοποιεί πλήρως ένα σωστό πρόγραμμα. Αν το  $A$  ήταν πρόσβαση σε έναν πόρο όπως ένα αρχείο δεν μπορούμε να το επαναχρησιμοποιήσουμε και για το  $B$  και για το  $\Gamma$ , τι γίνεται αν ένα από τα δύο απενεργοποιήσει τον πόρο ή στο παράδειγμα μας κλείσει το αρχείο? Χρειαζόμαστε λοιπόν έναν τρόπο να ελέγχουμε για αυτές τις περιπτώσεις.

Η γραμμική λογική προσθέτει την έννοια της κατανάλωσης. Με  $A \multimap B$  εννοούμε ότι για ένα  $A$  παράγετε ένα  $B$ . Προσθέτοντας επίσης έναν εναλλακτικό τελεστή για το "γραμμικό λογικό και"  $\otimes$  ισχύει ότι:

$$A \multimap B, A \multimap \Gamma, A \not\vdash B \otimes \Gamma$$

Το  $A$  καταναλώνετε από την πιο αριστερή οντότητα, κατά σύμβαση, το  $B$ . Με τον ίδιο συνειρμό ισχύει  $A \multimap B, A \multimap \Gamma, A, A \vdash B \otimes \Gamma$  καθώς τώρα έχουμε δύο οντότητες του  $A$  η κάθε μια θα καταναλωθεί από το  $B$  και  $\Gamma$  αντίστοιχα. Το γεγονός ότι ο μεταγλωττιστής καταλαβαίνει την γραμμική λογική δεν σημαίνει ότι την αντικαθιστά οποιαδήποτε άλλη πλήρως.

Πολλές φορές είναι απαραίτητο να παραχθούν  $v \in \mathbb{N}$  αναφορές (references) αμετάβλητης μορφής, δηλαδή ισχύει:

$$\alpha : T \rightarrow \beta : \&T \quad (1)$$

Όμως δεν μπορούμε να έχουμε παραπάνω από μια αναφορά αν είναι μεταβλητή, σε αυτήν την περίπτωση καταναλώνετε:

$$\alpha : T \multimap \beta : \&mutT \quad (2)$$

Το ίδιο ισχύει για οποιαδήποτε μη-αντιγράψιμη τιμή:

$$\alpha : T \multimap \beta : T \quad (3)$$

Ορισμένοι τύποι όμως είναι υπερβολικά πολύ εύκολο να αντιγραφθούν χρησιμοποιώντας κάποιου είδους memcry, για παράδειγμα u8. Αυτοί οι τύποι είναι αντιγράψιμοι.

$$\alpha : T \text{ is Copy} \rightarrow \beta : T \quad (4)$$

Η έννοια της κατανάλωσης της γραμμικής λογικής είναι ίδια ακριβώς με την έννοια της ιδιοκτησίας (ownership) της Rust. Παρακάτω είναι μερικά ακόμη παραδείγματα.

```

1 struct T;
2
3 fn reference_infinite_times(a: T) -> T {
4     let b = &a;
5     let c = &a;
6     let d = &a;
7     // let e = &mut a; Error, cant consume variable
8     // that is referenced in its lifetime
9     return a;
10 }
11
12 fn mutate_reference(mut a: &mut T) -> &mut T {
13     let b = a;
14     // return a; Error, use of consumed variable 'a'
15     return b;
16 }
17
18 fn consume_and_return(a: T) -> T {
19     let b = a;
20     // let c = a; Error, use of consumed variable 'a'
21     return b;
22 }
23
24 fn can_copy(mut a: u8) -> u8 {
25     let b = a;
26     let c = a;
27     let d = &mut a;
28     let e = &a;

```

```

29     return a;
30 }
31
32
33 fn main() {
34     let v0: T = T;
35
36     let v1 = reference_infinite_times(v0);
37     //infinite_references(var0); Error,
38     //use of consumed variable 'v0'
39
40     let mut v2 = consume_and_return(v1);
41     //infinite_references(var1); Error,
42     //use of consumed variable 'v1'
43
44     let v3 = mutate_reference(&mut v2);
45     //v2 is consumed, mutated and returned as v3
46
47     let v = can_copy(42);
48 }

```

### 3.1.2 Builder Pattern

Ίσως το πιο διαδεδομένο pattern που η λογική της Rust βοηθάει να μοντελοποιήσει είναι το Builder Pattern (πρότυπο κατασκευαστή). Επιτρέπει την κατασκευή σύνθετων αντικειμένων βήμα προς βήμα χρησιμοποιώντας την γραμμική λογική για να μπορέσουμε να ρυθμίσουμε το αντικείμενο ανάλογα με τις ανάγκες μας.

Όταν ένα struct διαθέτει πολλές προαιρετικές παραμέτρους, η χρήση μιας απλής συνάρτησης αρχικοποίησης τύπου new μπορεί να οδηγήσει σε δύσκολη συντήρηση και δυσανάγνωστο κώδικα. Χαρακτηριστικό παράδειγμα είναι η δημιουργία μιας δομής Car με πολλά χαρακτηριστικά:

```

1 struct Car {
2     brand: String,
3     model: String,
4     year: u16,
5     color: String,
6     automatic: bool,
7 }

```

Ένας κλασικός τρόπος για την αρχικοποίηση ενός αντικειμένου είναι η χρήση μιας συνάρτησης:

```

1 impl Car {
2     fn new(brand: &str, model: &str, year: u16, color:
    ↪ &str, automatic: bool) -> Self {

```

```

3      Car {
4          brand: brand.to_string(),
5          model: model.to_string(),
6          year,
7          color: color.to_string(),
8          automatic,
9      }
10 }
11 }
12
13 let car = Car::new("Toyota", "Corolla", 2022, "Blue",
    ↪ true);

```

Αυτός ο τρόπος δημιουργίας έχει τρία βασικά προβλήματα:

- Η σειρά των ορισμάτων είναι αυστηρή, το οποίο για μεγάλες βιβλιοθήκες ή σύνθετα drivers μπορεί να προκαλέσει σύγχυση στον χρήστη.
- Η ανάγνωση του κώδικα είναι δυσανάγνωστη, ιδιαίτερα όταν υπάρχουν πολλές παράμετροι.
- Το αντικείμενο δεν είναι ρυθμιζόμενο ενώ θα μπορούσε να είναι. Έστω δηλαδή ότι όλα τα αυτοκίνητα είναι manual, του 2020 και άσπρα εκτός αν ρυθμιστούν αλλιώς.

Το Builder Pattern αντιμετωπίζει αυτά τα προβλήματα, επιτρέποντας τη σταδιακή διαμόρφωση ενός αντικείμενου μέσω αλυσίδωσης μεθόδων (method chaining). Αυτές οι μέθοδοι ρύθμισης καταναλώνουν τον Builder και επιστρέφουν τον μεταλλαγμένο Builder με τα καινούργια πλέον πεδία επιτρέποντας να συνεχίσει η αλυσίδα ή να καλεστεί η τελική μέθοδος build που θα καταναλώσει τον Builder για μια τελευταία φορά και θα δημιουργήσει ένα αντικείμενο Car.

Παρακάτω δίνεται μια υλοποίηση του προτύπου για το struct Car:

```

1 struct Car {
2     brand: String,
3     model: String,
4     year: u16,
5     color: String,
6     automatic: bool,
7 }
8
9 struct CarBuilder {
10     brand: String,
11     model: String,
12     year: Option<u16>,
13     color: Option<String>,
14     automatic: Option<bool>,

```

```

15 }
16
17 impl CarBuilder {
18     fn new(brand: &str, model: &str) -> Self {
19         Self {
20             brand: brand.to_string(),
21             model: model.to_string(),
22             year: None,
23             color: None,
24             automatic: None,
25         }
26     }
27
28     fn year(mut self, year: u16) -> Self {
29         self.year = Some(year);
30         self
31     }
32
33     fn color(mut self, color: &str) -> Self {
34         self.color = Some(color.to_string());
35         self
36     }
37
38     fn automatic(mut self, automatic: bool) -> Self {
39         self.automatic = Some(automatic);
40         self
41     }
42
43     fn build(self) -> Car {
44         Car {
45             brand: self.brand,
46             model: self.model,
47             year: self.year.unwrap_or(2020), //
↳ Default value
48             color: self.color.unwrap_or_else(||
↳ "White".to_string()), // Default value
49             automatic:
↳ self.automatic.unwrap_or(false), // Default value
50         }
51     }
52 }
53
54 fn main() {
55     let car = CarBuilder::new("Toyota", "Corolla")
56         .year(2023)
57         .color("Red")

```

```

58         .automatic(true)
59         .build();
60
61     println!(
62         "Car: {} {} {} ({} ) | {} Automatic: {}",
63         car.brand, car.model, car.year, car.color,
64         ↪ car.automatic
65     );
66 }

```

Προφανώς είναι αρκετά παραπάνω boilerplate αλλά η χρήση για τον end-user της βιβλιοθήκης είναι υπερβολικά πιο απλοϊκή.

### 3.1.3 Asynchronous Rust

Το ασύγχρονο (async) μοντέλο της Rust είναι ένα ιδιαίτερα χρήσιμο εργαλείο, αλλά για να κατανοηθεί πλήρως είναι απαραίτητη μια εισαγωγή στις έννοιες των generics.

Τα generics στη Rust επιτρέπουν τη συγγραφή γενικευμένου κώδικα που μπορεί να λειτουργεί με διάφορους τύπους δεδομένων χωρίς ανάγκη για επανάληψη. Χρησιμοποιώντας generics, μπορούμε να ορίζουμε συναρτήσεις, δομές (structs), enumerations (enums) και traits, αυξάνοντας την επαναχρησιμοποίηση του κώδικα και την ασφάλεια τύπων.

```

1 fn get_first_element<T>(slice: &[T]) -> Option<T> {
2     slice.get(0)
3 }
4
5 fn main() {
6     let numbers = vec![1, 2, 3];
7     let first_number = get_first_element(&numbers);
8     println!("First number: {:?}", first_number);
9
10    let words = vec!["hello", "world"];
11    let first_word = get_first_element(&words);
12    println!("First word: {:?}", first_word);
13 }

```

Τα generics της Rust αποτελούν αφαίρεση χωρίς κόστος εκτέλεσης, καθώς ο μεταγλωττιστής παράγει εξειδικευμένο κώδικα ανάλογα με τους τύπους που χρησιμοποιούνται κατά την μεταγλώττιση στο ενδιάμεσο στάδιο THIR. Δηλαδή για το παραπάνω παράδειγμα η συνάρτηση `get_first_element` παράγει στην πραγματικότητα δύο συναρτήσεις στον τελικό κώδικα μια που έχει παράμετρο και `output usize` και μια `&str`.

Μπορούμε να περιορίσουμε το σύνολο των πιθανών τιμών ανάλογα με τα traits (ουσιαστικά interfaces της Java) που υλοποιεί το generic. Για παράδειγμα, αν θέλουμε μια συνάρτηση να μπορεί να χρησιμοποιεί μόνο τύπους δεδομένων που υλοποιούν το trait Display:

```
1 use std::fmt::Display;
```

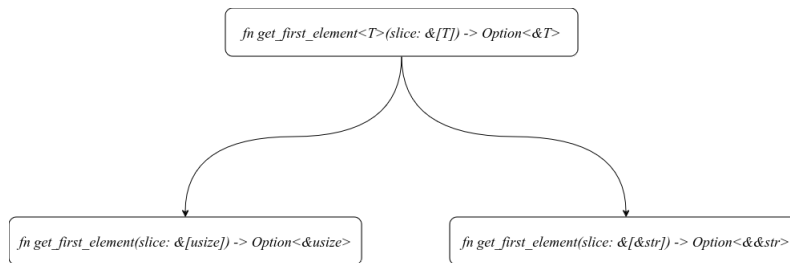


Figure 15: Παράδειγμα μεταγλώττισης generics.

```

2
3 fn print_element<T: std::fmt::Display>(element: T) {
4     println!("Element:␣{}", element);
5 }
6
7 fn main() {
8     print_element(42);           // i32 implements Display
9     print_element("hello");     // &str implements
10    ↪ Display
11     print_element(vec![1, 2, 3]); // Error: Vec does
12    ↪ not implement Display
13 }

```

Το μοντέλο async στην Rust βασίζεται στα Futures τα οποία με την σειρά τους στηρίζονται πολύ στα generics. Τα Futures είναι traits και αντιπροσωπεύουν τιμές που δεν είναι άμεσα διαθέσιμες αλλά θα γίνουν ενδεχόμενος κάποια στιγμή στο μέλλον. Ο μεταγλωτιστής υλοποιεί τα Futures ως state machines τα οποία δεν τρέχουν κατά την διάρκεια της εκτέλεσης αλλά καταλήγουν ενσωματωμένα στο τελικό binary ως loops και jumps.

Ένα απλό παράδειγμα χρήσης async είναι το κλασικό HTTP fetch:

```

1 async fn fetch_data(url: &str) -> Result<String,
2    ↪ reqwest::Error> {
3     let response = reqwest::get(url).await?;
4     let body = response.text().await?;
5     Ok(body)
6 }
7
8 #[tokio::main]
9 async fn main() {
10     match fetch_data("http://example.com").await {
11         Ok(body) => println!("Response:␣{}", body),
12         Err(e) => println!("Error:␣{:?}", e),
13     }
14 }

```



```
13 }
```

Το γεγονός ότι τα `async` σύμβολα είναι `compile time` οντότητες αφαιρεί ένα τεράστιο `overhead` που έρχεται με την υλοποίηση των παραδοσιακών runtimes.

Για να κατανοήσουμε πλήρως αυτόν μηχανισμό πρέπει να εξετάσουμε πώς αποσαφηνίζεται (`desugar`) ένα `async block`. Στο ενδιάμεσο στάδιο HIR οτιδήποτε σύμβολο είναι `async` μεταφράζεται στην πρωτόγονη μορφή του, δηλαδή ένα `Future` με μια `generic` παράμετρο.

Το `Future trait` φαίνεται κάπως έτσι:

```
1 pub trait Future<Output> {  
2     // Required method  
3     fn poll(self: Pin<&mut Self>, cx: &mut  
4         ↪ Context<'_>) -> Poll<Output>;  
5 }
```

Κατά συνέπεια η ταυτότητα της συνάρτησης `fetch` αποσαφηνίζεται από

```
1 async fn fetch_data(url: &str) -> Result<String,  
2     ↪ request::Error>
```

σε

```
1 fn fetch_data<F:  
2     ↪ Future<Result<String, request::Error>>>(url:  
3     ↪ &str) -> F
```

Παράλληλα το ενσωματωμένο μέρος της συνάρτησης αντικαθιστά τα `await` με την μέθοδο `poll`

```
1 fn fetch_data<F:  
2     ↪ Future<Result<String, request::Error>>>(url:  
3     ↪ &str) -> F  
4     let response = request::get(url).poll(someContext);  
5     let body = response.text().poll(someContext);  
6     Ok(body)
```

Προφανώς η υλοποίηση του `Future trait` και κατά συνέπεια της μεθόδου `poll` αλλάζει σε κάθε `async` οικοσύστημα. Η γενική ιδέα όμως είναι ότι η `poll` επιστρέφει την κατάσταση της συνάρτησης που καλείτε, δηλαδή:

1. Αν περιμένει την σειρά της για το επόμενο κάλεσμα (`Waiting`).
2. Αν βρίσκεται στην διαδικασία του καλέσματος (`Pending`).
3. Αν είναι έτοιμη και έχει τελειώσει το κάλεσμα (`Done`).

Τα `enums` της Rust μοντελοποιούν τις καταστάσεις σχεδόν πάντα, το παρακάτω παράδειγμα δεν θα κάνει `compile` για πολλούς λόγους και μπορεί να θεωρηθεί ψευδοκώδικας.

```

1  enum FetchDataState
2  {
3      Waiting,
4      Pending,
5      Done(Result<&str, request::Error>),
6  }
7
8  enum RespDataState
9  {
10     Waiting,
11     Pending,
12     Done(Result<&str, request::Error>),
13 }
14
15 struct get
16 {
17     state: FetchDataState,
18     ...,
19 }
20
21 struct resp
22 {
23     state: RespDataState,
24     ...,
25 }
26
27 impl<Output=Result<&str, request::Error>>
28     ↪ Future<Output> for get
29 {
30     fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
31     ↪ -> Poll<Output>
32     {
33         loop
34         {
35             match &mut this.state
36             {
37                 FetchDataState::Waiting => Poll::Pending,
38                 FetchDataState::Pending => Poll::Pending,
39                 FetchDataState::Done(res) =>
40                 ↪ Poll::Ready(res),
41             }
42         }
43     }
44 }

```

```

42
43 impl<Output=Result<&str,request::Error>>
44     ↪ Future<Output> for resp
45 {
46     fn poll(self: Pin<&mut Self>,cx: &mut Context<'_>)
47     ↪ -> Poll<Output>
48     {
49         loop
50         {
51             match &mut this.state
52             {
53                 RespDataState::Waiting => Poll::Pending,
54                 RespDataState::Pending => Poll::Pending,
55                 RespDataState::Done(res) => Poll::Ready(res),
56             }
57         }
58     }
59 }

```

Μια καλή ερώτηση σε αυτό το σημείο είναι πιο το νόημα? Στο παραπάνω παράδειγμα είναι ξεκάθαρο ότι δεν γίνεται τίποτα παράλληλα απλώς ελέγχουμε αν η συνάρτηση έχει τελειώσει και πράγματι αυτή είναι η περίπτωση.

Τα Futures δεν είναι τίποτα χωρίς έναν εκτελεστή (Executor) ο οποίος μπορεί να αλλάζει την συνάρτηση που καλείτε ανάλογα με το Context της. Ένας εκτελεστής αποτελείται από έναν Spawner που προσθέτει ή αφαιρεί συναρτήσεις που επιστρέφουν Futures και έναν Waker που "ξυπνάει" τις συναρτήσεις από τον Spawner για να ελέγξει την κατάσταση τους.

Στην παρακάτω υλοποίηση ψευδοκώδικα ο εκτελεστής είναι απλώς μια δυναμική λίστα και ο Spawner μια συνάρτηση που προσθέτει Tasks.

```

1 struct Executor {
2     queue: VecDeque<Task>,
3 }
4
5 impl Executor {
6     fn new() -> Self {
7         Executor {
8             queue: VecDeque::new(),
9         }
10    }
11
12    fn spawn(&mut self, task: Task) {
13        self.queue.push_back(task);
14    }
15
16    fn run(&mut self) {

```

```

17         while let Some(mut task) =
18             ↪ self.queue.pop_front() {
19                 let waker = task.get_waker();
20                 let mut context =
21                     ↪ Context::from_waker(&waker);
22                 match task.future.as_mut().poll(&mut
23                     ↪ context) {
24                     Poll::Pending =>
25                     ↪ self.queue.push_back(task),
26                     Poll::Ready(_) => {},
27                 }
28             }
29         }
30     }
31
32 struct Task {
33     future: Pin<Box<dyn Future<Output = ()>>>,
34 }
35
36 impl Task {
37     fn get_waker(&self) -> Waker {
38         //Wake the task
39     }
40 }
41
42 fn main() {
43     let mut executor = Executor { queue:
44         ↪ VecDeque::new() };
45
46     ↪ executor.queue.push_back(Task::new(fetch_data("http://example.com")));
47
48     executor.run();
49 }

```

Για λόγους αισθητικής τις περισσότερες φορές χρησιμοποιούνται macros τα οποία κρύβουν την πραγματική λειτουργία του executor. Τα macros στην Rust είναι πάρα πολύ έξυπνα και μπορούν να διαβάσουν πρακτικά όλο το συντακτικό της γλώσσας. Κατά συνέπεια μια συνάρτηση για παράδειγμα με τον εκτελεστή της embassy φαίνεται έτσι:

```

1 #[esp_hal_embassy::main]
2 async fn main(spawner: Spawner)
3 {
4     //do stuff
5     //spawn tasks

```

```

6  spawner.spawn(task1()).ok();
7  spawner.spawn(task2()).ok();
8  }

```

Προφανώς εδώ δεν φαίνεται να ορίζουμε κανέναν εκτελεστή και δεν υπάρχει καμία λούπα που να τρέχει τα tasks. Για τις περισσότερες εφαρμογές αυτό είναι ίσως πιο επιθυμητό στην δικιά μας περίπτωση όμως θέλουμε τα τρέχουμε χρονομετρητές πριν και μετά από κάθε poll που εκτελείται. Ήμαστε αναγκασμένοι λοιπόν να αποσαφηνίσουμε τα macros όσο το δυνατόν γίνεται:

```

1  #[main]
2  fn main() -> ! {
3      let executor = EXECUTOR.init(Executor::new(3 as
4      ↪ *mut ())),
5      let spawner = executor.spawner();
6      spawner.spawn(task1()).ok();
7      spawner.spawn(task2()).ok();
8      // let mut sleep_tick_count = 0;
9      let mut acc = 0.0;
10     let start_time = Instant::now();
11     loop {
12         let start_pol_time = Instant::now();
13         unsafe { executor.poll() };
14         let end_pol_time = Instant::now();
15         let elapsed_pol =
16         ↪ end_pol_time.duration_since(start_pol_time) as
17         ↪ f64;
18         acc += elapsed_pol_time;
19         let total_elapsed =
20         ↪ start_time.elapsed().as_ticks() as f64;
21         let cpu_usage = acc * 100_f64 / total_elapsed
22         //do stuff with cpu usage
23     }
24 }

```

Στον παραπάνω κώδικα η loop είναι η αντίστοιχη συνάρτηση run του ψευδοκώδικα η μόνη διαφορά είναι ότι δεν κάνουμε χειροκίνητα την επιλογή των task που θα τρέξει μέσω του Context. Στην πραγματικότητα το "Context" μπορεί να σημαίνει πολλά πράγματα, είναι απλώς ένας δείκτης που αντιπροσωπεύει διαφορετικά δεδομένα ανάλογα με την τιμή του. Εδώ επιλέχθηκε η τιμή 3 διαβάζοντας το source-code του εκτελεστή και συμπεραίνοντας ότι αυτή είναι η εντολή που αρχικοποιεί τον executor. Όταν καλούμε την poll προφανώς καλούνται εσωτερικά όλες οι συναρτήσεις που έχουν γίνει spawn.

### 3.1.4 HAL Patterns

Προφανώς κάθε MCU είναι διαφορετικό και διαθέτει διαφορετικά περιφερειακά, ποικίλες δυνατότητες, λιγότερα ή περισσότερα GPIO Pins etc. Το embedded-hal είναι μια συλλογή traits που αποσκοπούν στην μοντελοποίηση ενός abstract MCU. Υπάρχουν traits για πρωτόκολλα επικοινωνίας όπως UART και I2C αλλά και traits για λειτουργία των GPIO, PWM και οτιδήποτε άλλο είναι πιθανό να χρειαζόμαστε. Όπως έχουμε δει όμως τα traits δεν είναι τίποτα παραπάνω από πρότυπα, χρειαζόμαστε λοιπόν μια υλοποίηση των traits για τον δικό μας μικροελεγκτή. Θα μπορούσαμε για την συγκεκριμένη εργασία να κάνουμε τις δικές μας υλοποιήσεις, και θα γίνει αυτό σε έναν βαθμό αλλά υπάρχει ήδη το crate esp-hal το οποίο όχι μόνο προσφέρει υλοποιήσεις για αυτά τα traits αλλά παρέχει ένα struct, το Peripherals το οποίο μοντελοποιεί τα περιφερειακά του ESP.

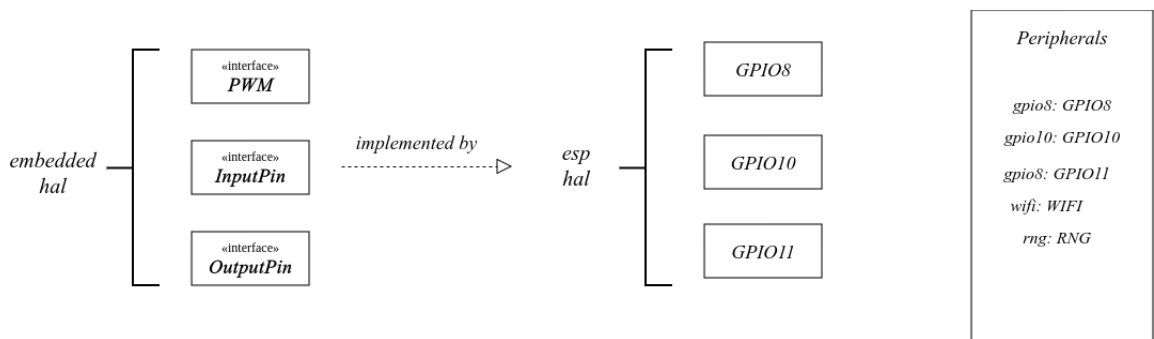


Figure 16: HAL Pattern για τον ESP.

Το Peripherals struct υλοποιείται διαφορετικά ανάλογα με τον μικροελεγκτή. Δηλαδή κάνουμε compile το esp-hal με ένα flag που αντιπροσωπεύει το όνομα του MCU που χρησιμοποιούμε. Στο cargo, το αντίστοιχο CMake της Rust, τα flags λέγονται features και γράφονται σε ένα αρχείο Cargo.toml που βρίσκεται στο root του project:

```

1 [dependencies]
2 ...other deps
3 esp-hal = { version = "0.23.1", features = ["esp32c6",
  ↳ "unstable"] }
```

Η προσέγγιση του Peripheral ακολουθεί τις αρχές της Rust που αναλύσαμε παραπάνω δηλαδή στο ότι κάθε περιφερειακό (USB Port,GPIO-Pins,SPI,UART) αποτελεί έναν μοναδικό και καταναλώσιμο πόρο. Κατά συνέπεια, μόλις το περιφερειακό ληφθεί από τη δομή που το διαχειρίζεται (συνήθως μια δομή τύπου Peripherals), θεωρείται ότι καταναλώθηκε και δεν μπορεί να χρησιμοποιηθεί ξανά χωρίς ρητή επιστροφή του ελέγχου από το πρόγραμμα. Για παράδειγμα, ένα GPIO pin (ψηφιακή θύρα εισόδου/εξόδου) μπορεί να χρησιμοποιηθεί είτε για είσοδο είτε για έξοδο, αλλά ποτέ ταυτόχρονα, καθώς η πρώτη χρήση καταναλώνει τον αντίστοιχο πόρο.

### 3.2 Γράφοντας το Project

Θα δούμε πως η υλοποίηση του Project με βάση το specification που έχουμε θέσει είναι ιδιαίτερα απλή. Παρόλα αυτά δεν προσφέρονται βιβλιοθήκες για τον HC-SR04 όπως στην περίπτωση του ESP-IDF. Παράλληλα το API της βιβλιοθήκης του MQTT είναι πολύ δύσκολο διαχειρίσιμο οπότε θα χρειαστεί κάποιου είδους βελτίωση χωρίς απαραίτητα να γράψουμε το πρωτόκολλο από την αρχή (το οποίο όμως φαίνεται να είναι η ιδανική λύση). Με βάση την γνώση που έχουμε δώσει στην εισαγωγή αυτού του κεφαλαίου όμως γίνεται ξεκάθαρο πως αυτό δεν αποτελεί πρόβλημα καθώς η εκφραση της Rust κάνει το γράψιμο κώδικα να φαίνεται σαν πολύ οργανική διαδικασία.

Μια επιλογή για να ξεκινήσουμε είναι να δημιουργήσουμε ένα Cargo.toml και να θέσουμε τις βιβλιοθήκες που θέλουμε να μεταγλωττίσουμε μαζί με το Project. Όμως υπάρχει μια καλύτερη λύση, το `esp-generate --chip esp32c6 my_project` δημιουργεί τα απαραίτητα αρχεία με τις βιβλιοθήκες που είναι απαραίτητες ανάλογα με τις ρυθμίσεις μας. Προφανώς αν θέλουμε προσθέτουμε κι άλλες μπορούμε είτε από ένα τερματικό `cargo add library` είτε με το συντακτικό TOML που έχουμε δείξει παραπάνω. Το τελικό directory έχει της εξής δομή:

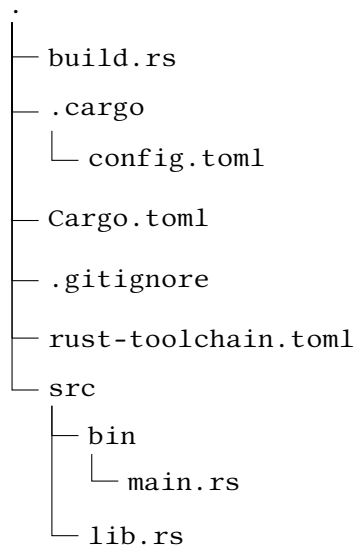


Figure 17: Project folder structure.

Το project θα αποτελείτε από δύο δικές μας βιβλιοθήκες που θα είναι υπεύθυνες για:

1. Την λειτουργία του αισθητήρα απόστασης.
2. Την λειτουργία του πρωτόκολλου MQTT.

### 3.2.1 HC-SR04 Driver

Ο σένσορας για να λειτουργήσει χρειάζεται εκτός από το VCC και το ground pin ένα input και ένα output. Οπότε γράφουμε σε ένα αρχείο `hcsr04.rs`:

```
1 pub struct HCSR04<'h, Clock: Timer> {  
2     trigger: Output<'h>,  
3     echo: Input<'h>,  
4     clock: Clock,  
5 }
```

Ήδη αρκετά κομμάτια στον κώδικα έχουν ιδιαίτερο ενδιαφέρον.

Αρχικά παρατηρούμε ότι δεν χρησιμοποιούμε συγκεκριμένο τύπο για GPIO pins. Χρησιμοποιώντας `Input` και `Output` όχι μόνο μοντελοποιούμε καλύτερα τον αισθητήρα αλλά αποφεύγουμε μελλοντικά runtime errors αν αφήναμε τον χρήστη να προσθέσει ότι pins ήθελε με τις δικές του ρυθμίσεις. Με αυτόν τον τρόπο εκφράζουμε "ο αισθητήρας καταναλώνει ένα output και ένα input pin".

Με την ίδια λογική ορίζουμε μια generic παράμετρο `Clock` η οποία περιορίζεται στους τύπους που υλοποιούν το `Timer` trait. Αφήνουμε λοιπόν στον χρήστη να επιλέξει το ρολόι που θα χρησιμοποιήσει εφόσον οι περισσότεροι μικροελεγκτές έχουν πολλά και ανάλογα με τους σκοπούς τα αποτελέσματα μπορούν να διαφέρουν. Για τον ESP, αν μας ενδιαφέρει πάρα πολύ η εγκυρότητα των τιμών του χρονομετρητή τότε συνήθως θέλουμε το system timer αλλιώς μπορούμε να επιλέξουμε ένα από τα `Timer Groups` που προσφέρει ο ESP ή ένα software timer της embassy το οποίο προφανώς βασίζεται σε ένα από τα παραπάνω.

Το συντακτικό `<'h>` έχει να κάνει με τον χρόνο ζωής (lifetime) των πεδίων στο εσωτερικό του struct. Η πλήρης ανάλυση των lifetimes είναι εκτός των πλαισίων της εργασίας αλλά ουσιαστικά αυτό που εκφράζουν είναι για πόσο χρόνο κατά την εκτέλεση του προγράμματος ζει ένα σύμβολο. Στην συγκεκριμένη περίπτωση όλα ζουν για το ίδιο χρονικό διάστημα, μέχρι ο `HCSR04` να βγει από το scope.

Όσο αναφορά τις μεθόδους του struct τις γράφουμε μέσα σε ένα `impl` block που διαθέτει τις κατάλληλες μεταβλητές. Τα `impl` blocks εκφράζουν το πιο μέθοδοι ορίζονται για έναν τύπο `T`.

```
1 impl<'h, Clock: Timer> HCSR04<'h, Clock>  
2 {  
3     //Methods go here  
4 }
```

Οι μεταβλητές αυτές είναι όσο πιο γενικές δυνατόν γίνεται. Θα μπορούσαμε κάλλιστα να ορίσουμε ένα πιο περιορισμένο implementation. Για παράδειγμα να ορίσουμε μεθόδους μόνο για τα `HCSR04` που έχουν για χρονομετρητή ένα `Alarm`:

```
1 impl<'h> HCSR04<'h, Alarm>  
2 {  
3     //Methods go here  
4 }
```



Χρειαζόμαστε προφανώς μια συνάρτηση για να δημιουργήσουμε το αντικείμενο που αντιπροσωπεύει τον αισθητήρα.

```
1 //inside the impl block
2 pub fn new<TRIG: PeripheralOutput + OutputPin, ECHO:
   ↳ PeripheralInput + InputPin>(
3   trigger: impl Peripheral<P = TRIG> + 'h,
4   echo: impl Peripheral<P = ECHO> + 'h,
5   timer: Clock,
6 ) -> Self {
7   Self {
8     trigger: Output::new(trigger, Level::Low),
9     echo: Input::new(echo, Pull::None),
10    clock: timer,
11  }
12 }
```

Η συνάρτηση έχει δύο παραμέτρους που είναι generic και πρέπει να κάνουν implement το Peripheral trait (διαφορετικό από το Peripheral struct).

1. Η trigger είναι μια παράμετρος που κάνει implement το Peripheral και είναι PeripheralOutput + OutputPin. Δηλαδή αναγκάζουμε την συνάρτηση να δεχτεί GPIO Pins που λειτουργούν σαν outputs. Έστω ότι για κάποιο λόγο προσπαθήσουμε να καλέσουμε την συνάρτηση με trigger = VCC Pin το πρόγραμμα δεν θα κάνει compile επειδή το VCC Pin δεν κάνει implement το OutputPin.
2. Η echo είναι μια παράμετρος που κάνει implement το Peripheral και είναι PeripheralInput + InputPin. Δηλαδή αναγκάζουμε την συνάρτηση να δεχτεί GPIO Pins που λειτουργούν σαν inputs. Έστω ότι για κάποιο λόγο προσπαθήσουμε να καλέσουμε την συνάρτηση με echo = VCC Pin το πρόγραμμα δεν θα κάνει compile επειδή το VCC Pin δεν κάνει implement το InputPin.

Η πιο ενδιαφέρουσα μέθοδος είναι αυτή που υπολογίζει την απόσταση:

```
1 //inside the impl block
2 pub fn measure_distance(&mut self) -> f64 {
3   let delay = Delay::new();
4
5   self.clock.start();
6   //Send ultrasonic wave
7   self.trigger.set_high();
8   delay.delay_micros(10);
9   self.trigger.set_low();
10  //Ultra sonic wave is sent
11
12  //Wait until echo bounces back
13  while self.echo.is_low() {
```

```

14 //DO NOT DEOPTIMIZE THE LOOP AWAY
15     unsafe { asm!("nop") };
16 }
17
18 //Start timing start = CURRENT_TIME
19 let start = self.clock.now();
20
21 //Wait until echo pin returns to low
22 while self.echo.is_high() {
23     //DO NOT DEOPTIMIZE THE LOOP AWAY
24     unsafe { asm!("nop") };
25 }
26
27 //End timing end = CURRENT_TIME > start
28 let end = self.clock.now();
29
30 let pulse_duration = (end - start).to_micros();
31
32 //return the distance
33 pulse_duration as f64 / 58_f64
34 }

```

Η συνάρτηση είναι blocking και υλοποιείται πλήρως συγχρονισμένα. Το πρώτο βήμα είναι να δημιουργήσουμε μια καθυστέρηση με υψηλή ακρίβεια μέσω της εντολής:

```

1 let delay = Delay::new();

```

Έπειτα ξεκινάει το χρονόμετρο ώστε να μπορούμε να μετρήσουμε τη διάρκεια του παλμού echo:

```

1 self.clock.start();

```

Στέλνουμε ένα κύμα υπερήχων στον αισθητήρα σηκώνοντας το trigger pin σε κατάσταση HIGH για 10μs:

```

1 self.trigger.set_high();
2 delay.delay_micros(10);
3 self.trigger.set_low();

```

Μετά, περιμένουμε να επιστρέψει η αντανάκλαση του κύματος. Όσο το echo pin παραμένει LOW (δηλαδή δεν έχει φτάσει ακόμα η αντανάκλαση), παραμένουμε σε μια ενεργή αναμονή (busy wait). Το `unsafe { asm!("nop") }` διασφαλίζει ότι η αναμονή δεν θα βελτιστοποιηθεί εκτός από τον compiler.

```

1 while self.echo.is_low() {
2     unsafe { asm!("nop") };
3 }

```

Όταν ανιχνευτεί η μετάβαση σε HIGH, ξεκινάει η χρονομέτρηση του παλμού:

```
1 let start = self.clock.now();
```

Ακολουθεί μια δεύτερη αναμονή έως ότου το pin echo επιστρέψει σε κατάσταση LOW, δηλαδή όταν τελειώσει ο παλμός:

```
1 while self.echo.is_high() {  
2     unsafe { asm!("nop") };  
3 }  
4 let end = self.clock.now();
```

Η διαφορά των δύο χρονικών στιγμών δίνει την διάρκεια του παλμού σε μικροδευτερόλεπτα:

```
1 let pulse_duration = (end - start).to_micros();
```

Τέλος, επιστρέφουμε την υπολογισμένη απόσταση χρησιμοποιώντας τον τύπο:

$$\text{απόσταση (σε εκατοστά)} = \frac{\text{χρόνος (μικροδευτερόλεπτα)}}{58}$$

```
1 pulse_duration as f64 / 58_f64
```

Ο συγκεκριμένος τύπος βασίζεται στην ταχύτητα του ήχου στον αέρα και λαμβάνει υπόψη το γεγονός ότι ο ήχος διανύει την απόσταση δύο φορές (προς και πίσω). Ο αριθμός 58 προκύπτει από την αναγωγή της εξίσωσης  $\Delta x = \frac{v \cdot t}{2}$  σε μορφή που επιστρέφει απόσταση σε εκατοστά.

Αξιοσημείωτο είναι επίσης το γεγονός ότι δεν χρησιμοποιείται καμία `await` ή ασύγχρονη λογική. Αυτό γίνεται για να εξασφαλιστεί ακρίβεια και συνέπεια στην μέτρηση χρόνου, καθώς οποιαδήποτε διακοπή θα αλλοίωνε το αποτέλεσμα. Η προσέγγιση αυτή είναι ιδανική για το portability του κώδικα καθώς μπορεί να τρέξει σε περιβάλλοντα τα οποία δεν έχουν executors και κατα συνέπεια async engines. Το να χρησιμοποιήσουμε μια μη-async συνάρτηση σε έναν executor είναι πολύ απλό οπότε συνήθως αυτός είναι ο τρόπος που προτείνεται το design.

### 3.2.2 MQTT Driver

Για την χρήση MQTT χρησιμοποιούμε την βιβλιοθήκη `rust-mqtt` και την `serde` για να κάνουμε `serialize` τα structs σε `json`.

Η βιβλιοθήκη `rust-mqtt` έχει αρκετά προβλήματα και αυτός είναι ο λόγος που αποφασίστηκε να δημιουργήσουμε ένα `wrapper` για το πείραμα.

Μας κάνει `export` ένα `mqtt client struct`:

```
1 pub struct MqttClient<'a, T, const MAX_PROPERTIES:  
    ↳ usize, R: RngCore>  
2 where  
3     T: Read + Write,  
4 {
```

```

5 raw: RawMqttClient<'a, T, MAX_PROPERTIES, R>,
6 }

```

Το οποίο φαίνεται προφανώς να είναι ένα wrapper για το πραγματικό client:

```

1 pub struct RawMqttClient<'a, T, const MAX_PROPERTIES:
   ↳ usize, R: RngCore>
2 where
3 T: Read + Write,
4 {
5 connection: Option<NetworkConnection<T>>,
6 buffer: &'a mut [u8],
7 buffer_len: usize,
8 recv_buffer: &'a mut [u8],
9 recv_buffer_len: usize,
10 config: ClientConfig<'a, MAX_PROPERTIES, R>,
11 }

```

Μπορεί κατά πρώτη όψη να μην είναι προφανές το πρόβλημα, αλλά ελέγχοντας πλήρως τον κώδικα δεν γίνεται καθόλου διατήρηση (ούτε καν αρχικοποίηση) των εσωτερικών buffer του `RawMqttClient` σε κανένα από τα παραπάνω structs. Ο χρήστης είναι υπεύθυνος να δημιουργήσει τα buffers, το socket που ακούει και στέλνει και να τα περάσει στο client. Αν χρησιμοποιούσαμε την βιβλιοθήκη κατευθείαν ο κώδικας θα ήταν δυσανάγνωστος στην καλύτερη περίπτωση οπότε προσθέτουμε ένα παραπάνω επίπεδο αφαίρεσης:

```

1 pub struct MqttDriver<'a> {
2 client: MqttClient<'a, &'a mut TcpSocket<'static>, 5,
   ↳ CountingRng>,
3 }
4
5 impl<'a> MqttDriver<'a>
6 {
7 }

```

Ο σκοπός είναι να κρύψουμε την διαχείριση του TCP socket από τον χρήστη οπότε αναγκαστικά δημιουργούμε ένα global socket μόλις ο χρήστης κάνει import την βιβλιοθήκη. Παράλληλα αρχικοποιούμε τα buffers για το TCP socket και το MQTT client.

```

1 //Global variables
2 //MQTT BUFFERS
3 static mut RECV_BUFFER: [u8; 4096] = [0; 4096];
4 static mut WRITE_BUFFER: [u8; 4096] = [0; 4096];
5 //TCP SOCKET BUFFERS
6 static mut RX_BUFFER: [u8; 4096] = [0; 4096];
7 static mut TX_BUFFER: [u8; 4096] = [0; 4096];

```

```

8 //SOCKET STATICCELL
9 static mut SOCKET: StaticCell<TcpSocket> =
    ↪ StaticCell::new();

```

Δεν μπορούμε να δημιουργήσουμε κατευθείαν ένα `TcpSocket` καθώς η μέθοδος `TcpSocket::new` έχει το signature `new(Stack, &mut [u8], &mut [u8])` όπου `stack` είναι το network interface που δημιουργείτε αργότερα κατά την εκτέλεση του προγράμματος. Παράλληλα δεν μπορούμε να κάνουμε `allocate` μνήμη στο `Heap` επειδή δεν ξέρουμε αν θα υπάρχει απαραίτητα. Το `StaticCell` είναι ένας έξυπνος δείκτης που δείχνει στην στατική μνήμη του προγράμματος. Γνωρίζουμε το (μέγιστο) μέγεθος του `TcpSocket` οπότε μπορούμε να δεσμεύσουμε μνήμη ίση με αυτό το μέγεθος και να δείξουμε σε αυτήν χωρίς να την αρχικοποιήσουμε. Αργότερα όταν παραχθεί ένα `Stack` καλούμε μια μέθοδο `init` για να θέσουμε τον δείκτη ίσο με το αποτέλεσμα του `TcpSocket::new`.

Ο χρήστης επίσης θα πρέπει να προσθέσει πολλές δικές του ρυθμίσεις οι οποίες μερικές μπορούν και μερικές δεν μπορούν να αποφευχθούν:

1. Για πόσο η σύνδεση να είναι Keep-Alive (Optional).
2. Το μέγιστο μέγεθος των πακέτων (πρέπει να είναι μικρότερο από το μέγεθος των buffers προφανώς) (Optional).
3. Την IP του εξωτερικού broker (Necessary).
4. Το ID του ως client (Necessary).
5. Το Username του ως client (Necessary).
6. Το Password του ως client (Necessary).

Αυτό φαίνεται πολύ καλό use-case για το builder pattern:

```

1 //inside MqttDriver impl block
2 pub fn new() -> MqttDriverBuilder<'a> {
3     MqttDriverBuilder {
4         id: "id",
5         username: "username",
6         pwd: "password",
7         keep_alive: None,
8         packet_size: None,
9         broker: (IpAddress::v4(0, 0, 0, 0), 1883),
10    }
11 }

```

Όπου το `MqttDriverBuilder` είναι:

```

1 pub struct MqttDriverBuilder<'a> {
2     id: &'a str,

```

```

3     username: &'a str,
4     pwd: &'a str,
5     keep_alive: Option<u16>,
6     packet_size: Option<u32>,
7     broker: (IpAddress, u16),
8 }
9
10 impl<'a> MqttDriverBuilder<'a> {
11     pub fn with_id(mut self, id: &'a str) -> Self {
12         self.id = id;
13         self
14     }
15
16     pub fn with_username(mut self, username: &'a str)
17     ↪ -> Self {
18         self.username = username;
19         self
20     }
21
22     pub fn with_password(mut self, pwd: &'a str) ->
23     ↪ Self {
24         self.pwd = pwd;
25         self
26     }
27
28     pub fn with_keep_alive(mut self, t: u16) -> Self {
29         self.keep_alive = Some(t);
30         self
31     }
32
33     pub fn with_packet_size(mut self, s: u32) -> Self {
34         self.packet_size = Some(s);
35         self
36     }
37
38     pub fn with_broker_ip(mut self, ip: IpAddress,
39     ↪ port: u16) -> Self {
40         self.broker = (ip, port);
41         self
42     }
43
44     pub async fn build(self, stack: Stack<'static>) ->
45     ↪ Result<MqttDriver<'a>, ConnectError> {
46         let mut conf: ClientConfig<'_, 5, CountingRng>
47         ↪ =
48             ClientConfig::new(MqttVersion::MQTTv5,

```

```

44     ↪ CountingRng(20000));
45     conf.add_client_id(self.id);
46     conf.add_username(self.username);
47     conf.add_password(self.pwd);
48     conf.max_packet_size =
49     ↪ self.packet_size.unwrap_or(1024);
50     conf.keep_alive =
51     ↪ self.keep_alive.unwrap_or(1000);
52
53     let socket = unsafe {
54     ↪ SOCKET.init(TcpSocket::new(stack, &mut
55     ↪ RX_BUFFER, &mut TX_BUFFER)) };
56
57     let status = socket.connect(self.broker).await;
58
59     match status {
60     ↪ Ok(_) => Ok(MqttDriver {
61     ↪     client: MqttClient::new(
62     ↪         socket,
63     ↪         unsafe { &mut WRITE_BUFFER },
64     ↪         4096,
65     ↪         unsafe { &mut RECV_BUFFER },
66     ↪         4096,
67     ↪         conf,
68     ↪     ),
69     ↪     }),
70     ↪ Err(err) => Err(err),
71     }
72 }

```

Η μέθοδος `build` είναι η τελευταία μέθοδος της αλυσίδας που πρέπει να καλεστεί. Καταναλώνει τον builder, αρχικοποιεί τον δείκτη στατικής μνήμης, δηλαδή το `TcpSocket`, και προσπαθεί να δημιουργήσει μια TCP σύνδεση με τον broker. Αν αυτή η σύνδεση είναι επιτυχής επιστρέφεται το `MqttDriver` αν όχι επιστρέφεται ένα σφάλμα και ο χρήστης είναι υπεύθυνος για το πως θα διαχειριστεί.

Τώρα απομένουν οι βασικές ενέργειες που πρέπει να κάνει το `MqttDriver`

1. Να μπορεί να καθιερώσει μέσω της TCP σύνδεσης μια MQTT σύνδεση.
2. Να μπορεί να δημοσιεύσει μηνύματα σε topics στον broker.

Αυτές οι ενέργειες μπορούν να αντιπροσωπευτούν σε δύο ξεχωριστές μεθόδους

```

1 //in MqttDriver impl block
2 pub async fn connect(&mut self) -> Result<(),
3     ↪ ReasonCode> {

```

```

3     self.client.connect_to_broker().await
4 }
5
6 pub async fn publish_sensor(&mut self, message:
7     ↳ SensorMessage) -> Result<(), ReasonCode> {
8     let mut json = [0u8; 248];
9     let len = serde_json_core::to_slice(&message, &mut
10     ↳ json).unwrap();
11     let a = core::str::from_utf8(&json[..len]).unwrap();
12     self.client
13         .send_message(
14             "sensors/distance",
15             a.as_bytes(),
16             QualityOfService::QoS0,
17             false,
18         )
19         .await
20 }
21
22 pub async fn publish_device(&mut self, message:
23     ↳ DeviceMessage) -> Result<(), ReasonCode> {
24     let mut json = [0u8; 248];
25     let len = serde_json_core::to_slice(&message, &mut
26     ↳ json).unwrap();
27     let a = core::str::from_utf8(&json[..len]).unwrap();
28     self.client
29         .send_message(
30             "device/performance",
31             a.as_bytes(),
32             QualityOfService::QoS0,
33             false,
34         )
35         .await
36 }

```

Τα `SensorMessage` και `DeviceMessage` είναι απλώς structs που μέσω macros κάνουν implement δύο traits του `Serde`:

1. Το `Serialize` που επιτρέπει την μετατροπή του struct message σε ένα String της μορφής JSON.
2. Το `Deserialize` που επιτρέπει την μετατροπή ενός String της μορφής JSON σε ένα struct message.

και φαίνονται έτσι:

```

1 #[derive(Serialize, Deserialize)]

```



```

2 pub struct SensorMessage {
3     distance: f64,
4 }
5
6 impl SensorMessage {
7     pub fn new(distance: f64) -> Self {
8         Self { distance }
9     }
10 }
11
12 #[derive(Serialize, Deserialize)]
13 pub struct DeviceMessage {
14     cpu_usage: f64,
15     current_mem_usage: usize,
16 }
17
18 impl DeviceMessage {
19     pub fn new(cpu_usage: f64, current_mem_usage: usize)
20         ↪ -> Self {
21         Self {
22             cpu_usage,
23             current_mem_usage,
24         }
25     }
26 }

```

Τα traits μας δίνουν πρόσβαση στην συνάρτηση `serde_json_core::to_slice(T, &mut [u8])` τα οποία δημιουργούν το JSON σε bytes με βάση την τιμή του Message. Παρατηρούμε ότι κατά την δημοσίευση των μηνυμάτων δεν στέλνουμε ολόκληρο το buffer &json αλλά το `&json[. . len]` δηλαδή το buffer μέχρι το τελευταίο byte που αντιπροσωπεύει ένα char. Εάν δεν γίνει αυτή η εξακρίβωση το telegraf θα δυσκολευτεί να κάνει parse το json και πιθανότατα θα καταλήξει σε σφάλμα.

### 3.2.3 Main Program

Πλέον έχουμε τις κατάλληλες βιβλιοθήκες για να δημιουργήσουμε το τελικό project. Για το κύριο πρόγραμμα μπορούμε να ξεκινήσουμε τις προφανείς global μεταβλητές που χρειάζονται

```

1 //Resources
2 static EXECUTOR: StaticCell<Executor> =
3     ↪ StaticCell::new();
4 static WIFI_CONTROLLER: StaticCell<EspWifiController>
5     ↪ = StaticCell::new();
6 static STA_STACK_RESOURCES:
7     ↪ StaticCell<StackResources<3>> =

```

```

    ↪ StaticCell::new();
5 //Signals
6 static RUN_MQTT: Signal<CriticalSectionRawMutex, bool>
    ↪ = Signal::new();
7 static DISTANCE_SIGNAL:
    ↪ Signal<CriticalSectionRawMutex, f64> =
    ↪ Signal::new();
8 static CPU_SIGNAL: Signal<CriticalSectionRawMutex,
    ↪ f64> = Signal::new();
9 static MEM_SIGNAL: Signal<CriticalSectionRawMutex,
    ↪ usize> = Signal::new();
10
11 //WiFi Credentials
12 const SSID: &str = "some_ssid";
13 const PWD: &str = "some_pwd";

```

Τα StaticCells είναι απλώς δείκτες όπως έχουμε εξηγήσει για έναν Executor, Wifi-Controller και ένα Stack για ένα WiFi station, θα αρχικοποιηθούν αργότερα. Τα Signals είναι global μεταβλητές που σταματάνε την ροή του task αλλά δεν είναι blocking στον executor επιτρέποντας σε άλλα tasks να τρέξουνε, εσωτερικά λειτουργούν σαν mutexes.

Χρειαζόμαστε επίσης κάποιου είδους allocator για ένα μικρό heap που είναι απαραίτητο για το WiFi και μόνο:

```

1 extern crate alloc;
2 const HEAP_SIZE: usize = 32 * 1024;
3 static mut HEAP: MaybeUninit<[u8; HEAP_SIZE]> =
    ↪ MaybeUninit::uninit();

```

Πρέπει να κάνουμε annotate την κύρια συνάρτηση ως main για να ξέρει ο linker ότι αυτό είναι το entry point του προγράμματος.

```

1 #[main]
2 fn main() -> ! {
3     //program code here
4 }

```

Έχουμε ήδη αναφέρει το πως θέλουμε να λειτουργεί ο εκτελεστής έτσι ώστε να μπορούμε να μετρήσουμε το CPU usage. Πριν αρχίσει το πρόγραμμα να τρέχει όμως πρέπει να κάνουμε set up των drivers και να αρχικοποιήσουμε σημαντικές μεταβλητές όπως το heap:

```

1 //Inside main
2 //Run in max cpu freq
3 let config =
    ↪ esp_hal::Config::default().with_cpu_clock(CpuClock::max());
4 //get peripherals

```

```

5 let peripherals = esp_hal::init(config);
6 //enable logging
7 esp_println::logger::init_logger_from_env();
8 //Initialize the heap
9 unsafe {
10
11     ↪ esp_alloc::HEAP.add_region(esp_alloc::HeapRegion::new(
12         HEAP.as_mut_ptr() as *mut u8,
13         HEAP_SIZE,
14         esp_alloc::MemoryCapability::Internal.into(),
15     ));
16 }
17 //Embassy Peripherals
18 let timg1 = TimerGroup::new(peripherals.TIMG1);
19 let timg0 = TimerGroup::new(peripherals.TIMG0);
20 let mut rng = Rng::new(peripherals.RNG);
21 let wifi = peripherals.WIFI;
22
23 //HCSR04 Peripherals
24 let timsys = SystemTimer::new(peripherals.SYSTIMER);
25 let trigger = peripherals.GPIO11;
26 let echo = peripherals.GPIO10;

```

Η αρχικοποίηση του WiFi φαίνεται λίγο δυσνόητη αλλά στην πραγματικότητα είναι απλώς υπερβολικά πολύ verbose

1. Πρώτα αρχικοποιούμε τον WifiController με ένα Timer, ένα Rng και ένα Radio Clock.
2. Έπειτα θέτουμε το mode του controller ως station, αυτό μας επιστρέφει ένα wifi client driver τύπου WifiDevice και τον καινούργιο controller με το mode που θέσαμε.
3. Μετά χρειαζόμαστε ένα configuration για να πούμε στο client το πως θα συνδεθεί σε ένα Router (Username, Password, Encryption Protocol) και το πως θα πάρει IP. Εμείς θα χρησιμοποιήσουμε το default DHCP Client configuration.
4. Τέλος δημιουργούμε το TCP/IP stack με ένα random seed, τα resources που έχουμε στην στατική μνήμη και τα configurations που δημιουργήσαμε.

```

1 let wifi_controller =
2
3     ↪ &*WIFI_CONTROLLER.init(esp_wifi::init(timg0.timer0,
4     ↪ rng, peripherals.RADIO_CLK).unwrap());

```

```

4  let sta_resources =
    ↪ STA_STACK_RESOURCES.init(StackResources::new());
5  let (wifi_sta, mut controller) =
6      esp_wifi::wifi::new_with_mode(wifi_controller,
    ↪ wifi, WifiStaDevice).unwrap();
7
8  let sta_config =
    ↪ embassy_net::Config::dhcpv4(Default::default());
9  let rng_seed = (rng.random() as u64) << 32 |
    ↪ rng.random() as u64;
10 let conf =
    ↪ Configuration::Client(ClientConfiguration {
11     ssid: String::<32>::from_str(SSID).unwrap(),
12     password: String::<64>::from_str(PWD).unwrap(),
13     auth_method: AuthMethod::WPA2Personal,
14     ..Default::default()
15 });
16 controller.set_configuration(&conf).unwrap();
17 let (sta_stack, sta_runner) =
    ↪ embassy_net::new(wifi_sta, sta_config,
    ↪ sta_resources, rng_seed);

```

Στο τελευταίο βήμα μας επιστρέφετε ένας τύπος `Runner` και το `Stack`. Πλέον ήμασταν έτοιμοι να αρχικοποιήσουμε το `embassy` και τον `HCSR04` και τον `executor`:

```

1  //Embassy Initialization
2  esp_hal_embassy::init(timg1.timer0);
3  //HCSR04
4  let hcsr04 = HCSR04::new(trigger, echo, timsys.alarm0);
5  //MQTT Driver
6
7  //Executor
8  let executor = EXECUTOR.init(Executor::new(3 as *mut
    ↪ ())),);
9  let spawner = executor.spawner();
10 spawner.must_spawn(sta_connection(sta_runner));
11 spawner.must_spawn(wifi_connection(controller));
12 spawner.must_spawn(mqtt(sta_stack));
13 spawner.must_spawn(measure_distance(hcsr04));
14 //Spawn tasks
15 let mut acc = 0.0;
16 let start_time = Instant::now();
17 loop {
18     let start_pol = Instant::now();
19     unsafe { executor.poll() };
20     let end_pol = Instant::now();

```

```

21 let elapsed_pol =
    ↪ end_pol.duration_since(start_pol).as_ticks() as
    ↪ f64;
22 acc += elapsed_pol;
23 let total_elapsed =
    ↪ start_time.elapsed().as_ticks() as f64;
24 CPU_SIGNAL.signal(acc * 100_f64 / total_elapsed);
25 let used_mem = esp_alloc::HEAP.used();
26 MEM_SIGNAL.signal(used_mem);
27 }

```

Προφανώς δεν έχουμε γράψει ακόμα τα tasks που κάνουν spawn:

1. Το `sta_connection` απλώς θα καλεί την μέθοδο του runner `runner.run()`.
2. Το `wifi_connection` θα περιμένει μέχρι το station είναι έτοιμο. Μόλις είναι θα ξυπνήσει το mqtt.
3. Το mqtt κοιμάται μέχρι να ξυπνήσει από το `wifi_connection` μέσω ενός `RUN_MQTT` signal. Μόλις ξυπνήσει ελέγχει αν έχει λάβει ipv4 και μπορεί να επικοινωνήσει με τον έξω κόσμο. Μόλις λάβει ηλεκτρονική διεύθυνση συνδέεται στον broker και περιμένει για τα signals CPU και DISTANCE τα οποία μόλις λάβει τα στέλνει στον broker.
4. Το `measure_distance` μετράει την απόσταση που υπολογίζει ο αισθητήρας HCSR04 μέσω του driver και στέλνει μέσω του signal την τιμή αυτής της απόστασης στο mqtt.

Ένα task είναι απλώς μια συνάρτηση που είναι annotated με ένα macro `#[task]`:

```

1  #[task]
2  async fn sta_connection(mut runner: Runner<'static,
    ↪ WifiDevice<'static, WifiStaDevice>>) {
3  runner.run().await;
4  }
5
6  #[task]
7  async fn wifi_connection(mut controller:
    ↪ WifiController<'static>) {
8
9  controller.start_async().await.unwrap();
10 match controller.connect_async().await {
11     Ok(_) => (),
12     Err(err) => info!("Error_{}_connecting_{{:#?}}", err),
13 }
14
15 loop {
16     match esp_wifi::wifi::sta_state() {

```

```

17         WifiState::StaConnected => {
18             //Signal status that station has connected.
19             RUN_MQTT.signal(true);
20             break;
21         }
22         e => info!("Other_{:#?}", e),
23     }
24     TaskTimer::after_millis(500).await;
25 }
26 }
27
28 #[task]
29 async fn measure_distance(mut sensor: HCSR04<'static,
    ↳ Alarm>) {
30     loop {
31         DISTANCE_SIGNAL.signal(sensor.measure_distance());
32         TaskTimer::after_millis(250).await;
33     }
34 }
35
36 #[task]
37 async fn mqtt(sta_stack: Stack<'static>) {
38     //Wait for signal that WiFi_Sta has connected to a
    ↳ network
39     let _run = RUN_MQTT.wait().await;
40
41     //Wait to get IP.
42     loop {
43         if let Some(config) = sta_stack.config_v4() {
44             break;
45         }
46         TaskTimer::after_millis(500).await;
47     }
48     //TCP Connect to Broker
49
50     let mqtt_driver = MqttDriver::new()
51         .with_broker_ip(IpAddress::v4(77, 237, 242, 215),
    ↳ 1883)
52         .with_id("esp32c6Client")
53         .with_password("chris")
54         .with_username("chris")
55         .with_packet_size(1024)
56         .with_keep_alive(300)
57         .build(sta_stack)
58         .await;
59

```

```

60 match mqtt_driver {
61     Ok(mut mqtt) => {
62         //MQTT connect to broker
63         let status = mqtt.connect().await;
64         match status {
65             Ok(_) => loop {
66                 let current_distance =
↪ DISTANCE_SIGNAL.wait().await;
67                 let msg =
↪ SensorMessage::new(current_distance);
68
69                 if let Err(err) =
↪ mqtt.publish_sensor(msg).await {
70                     info!("Received_reason_code_
↪ {:#?}", err);
71                 }
72
73                 let cpu_usage =
↪ CPU_SIGNAL.wait().await;
74                 let mem_usage =
↪ MEM_SIGNAL.wait().await;
75                 let msg =
↪ DeviceMessage::new(cpu_usage, mem_usage);
76                 if let Err(err) =
↪ mqtt.publish_device(msg).await {
77                     info!("Received_reason_code_
↪ {:#?}", err);
78                 }
79             },
80             Err(err) => info!("Mqtt_Error:_{:#?}",
↪ err),
81         }
82     }
83     Err(err) => info!("Tcp_Error:_{:#?}", err),
84 }
85 }

```

## References

- [1] Ann Marie Corvin, *World in Disruption: Trust in Rust*. [Online]. Available: <https://techinformed.com/world-in-disruption-trust-in-rust/> (visited on 02/15/2025).
- [2] C. Catalin, *Microsoft: 70 percent of all security bugs are memory safety issues*. [Online]. Available: <https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/> (visited on 02/15/2025).
- [3] G. Thomas, *A proactive approach to more secure code*. (visited on 02/15/2025).