Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχ. και Μηχανικών Υπολογιστών

Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων VLSI

Σχεδιασμός Ενσωματωμένων Συστημάτων

Project 2020-2021 - Παρουσίαση

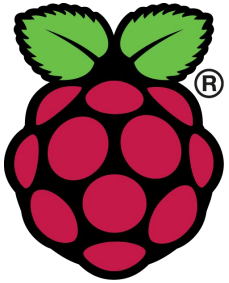
Custom ARM OS from scratch

armOS


Κουτρούμπας Αθανάσιος

Project Advisors:
Δημοσθένης Μασούρος
Μανώλης Κατσαραγάκης

Αρχική Ιδέα



Υλοποίηση ενός βασικού πυρήνα για ARM επεξεργαστή.

- Διεπαφή ανάμεσα σε User Applications και Hardware.
- Διαχείριση διεργασιών και μνήμης.
- Διαχείριση των hardware συσκευών μέσω drivers.

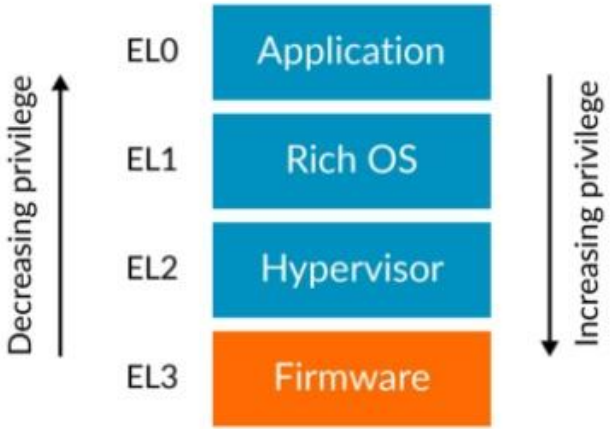
Επιλογή του Raspberry Pi ως development board:

- Φθηνές - Διαθέσιμες συσκευές (Pi Zero W ~ 10€).
- Πολύ μεγάλο community για bare-metal development.
- Καλό Documentation για τα περιφερειακά και λειτουργία του (όχι και τόσο τελικά...)

Υποστήριξη πυρήνα (kernel) για 2 διαφορετικές αρχιτεκτονικές και Pi boards:

Raspberry Pi Board	Αρχιτεκτονική	Chip	Επεξεργαστής	ARM Family
Pi Zero W	32-bit - Aarch32	BCM2835	ARM1176jzf	ARMv6
Pi 4	64-bit - Aarch64	BCM2711	ARM Cortex-A72	ARMv8-A

ARMv6-ARMv7 vs. ARMv8



Αρκετά διαφορετικές αρχιτεκτονικές:

- Instruction Set:
  - 32-bit vs. 64-bit Architecture and Assembly Language
- Processor State:
  - Current Program Status Register - `CPSR`
  - Πολλοί System Registers που καθορίζουν το state - `PSTATE`
- Exception Levels:
  - Privilege Levels - `PL0, PL1, PL2`
  - Exception Levels - `EL0, EL1, EL2, EL3`

Exception Levels καθορίζουν:

- Δικαιώματα πρόσβασης στην μνήμη.
- Δικαιώματα πρόσβασης σε καταχωρητές συστήματος.
- Δίνουν την δυνατότητα για απομόνωση των εφαρμογών, για μεγαλύτερη ασφάλεια.

Custom Kernel - Bare Metal Applications

Ο κώδικας που γράφουμε δεν θα τρέξει κάτω από κάποιο Λειτουργικό Σύστημα (μάλλον αυτός θα είναι το ΛΣ), υπάρχουν κάποιες προϋποθέσεις που πρέπει να ξέρουμε:

- Δεν υπάρχει **virtual memory** ούτε κάποια προστασία μνήμης -> Όλη η φυσική μνήμη είναι διαθέσιμη.
  - Συνέπεια: Δεν υπάρχουν segmentation faults (ούτε kernel oops), αν κάτι πάει στραβά, δύσκολο να το ξέρουμε.

```
thanos@thanos-X240 ~/Desktop ./bad_program
[1]    519788 segmentation fault (core dumped)  ./bad_program
```

- Δεν υπάρχει στοίβα (**stack**), και καμία άλλη δομή δεδομένων έτοιμη.
- Δεν μπορούμε να χρησιμοποιήσουμε την **βιβλιοθήκη της C**.
  - Όποια συνάρτηση θέλουμε να χρησιμοποιήσουμε, πρέπει να την υλοποιήσουμε.

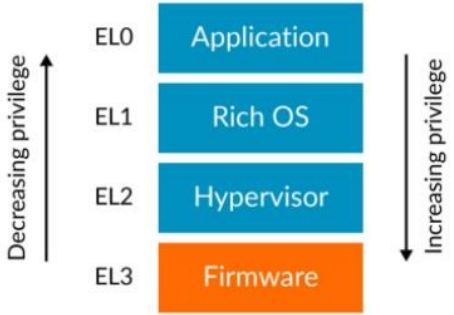
Booting

Διαδικασία Booting για το Raspberry Pi:

- Η GPU τρέχει πρώτη και αρχικοποιεί το hardware.
- Ψάχνει να βρει ένα αρχείο στο `/boot` directory της κάρτας SD, που αρχίζει με `kernel` και τελειώνει σε `.img` (Ο custom kernel compiled και σε binary μορφή).
- Φορτώνει τον πυρήνα στην κατάλληλη διεύθυνση και ξεκινάει η CPU να εκτελεί εντολές από εκείνη την διεύθυνση.
    - Για **32-bit** kernels, kernel φορτώνεται στην διεύθυνση: **`0x8000`**
    - For **64-bit** kernels, kernel φορτώνεται στην διεύθυνση: **`0x80000`**

```
thanos@thanos-X240 ~/Desktop ls boot
bcm2708-rpi-b.dtb       bcm2709-rpi-2-b.dtb       bcm2711-rpi-4-b.dtb  fixup4.dat     fixup_x.dat        start4db.elf  start_x.elf
bcm2708-rpi-b-plus.dtb  bcm2710-rpi-2-b.dtb       bcm2711-rpi-cm4.dtb  fixup4db.dat   issue.txt          start4.elf
bcm2708-rpi-b-rev1.dtb  bcm2710-rpi-3-b.dtb       bootcode.bin         fixup4x.dat    kernel.img         start4x.elf
bcm2708-rpi-cm.dtb      bcm2710-rpi-3-b-plus.dtb  cmdline.txt          fixup_cd.dat   LICENCE.broadcom   start_cd.elf
bcm2708-rpi-zero.dtb    bcm2710-rpi-cm3.dtb       COPYING.linux        fixup.dat      overlays           start_db.elf
bcm2708-rpi-zero-w.dtb  bcm2711-rpi-400.dtb       fixup4cd.dat         fixup_db.dat   start4cd.elf       start.elf
```

1ο Στάδιο kernel - Setup C environment

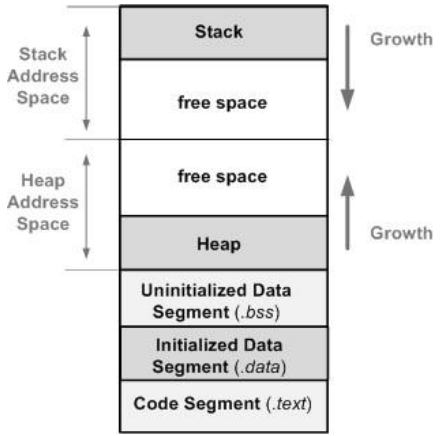


Για την εκτέλεση του κώδικα με κατάλληλα δικαιώματα για το Pi 4, χρειάζεται:

- Να ορίσουμε μόνο έναν CPU Core (**Core 0**) να εκτελεί εντολές.
- Να εισέλθουμε σε **Exception Level 1 - Rich OS**.

Για να έχουμε ένα βασικό περιβάλλον για να γράψουμε κώδικα C χρειάζεται:

- Να ορίσουμε την στοίβα (**stack**).
- Να μηδενίσουμε το Block Starting Symbol (**BSS**) segment.
  - Χώρος global μεταβλητών που δεν έχουν αρχικοποιηθεί.
- Να μεταφέρουμε την εκτέλεση σε C κώδικα:
  - Συνάρτηση `kernel_main()`

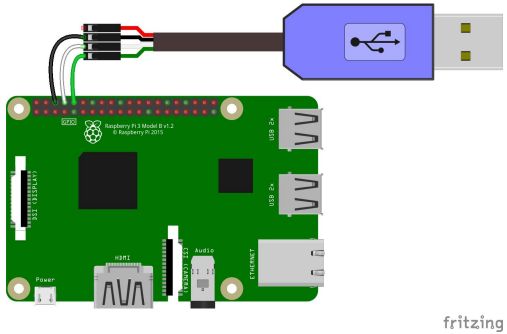


Code Reference: `/arch/armv6/boot.S, /arch/armv8-a/boot.S`

2ο Στάδιο kernel - Σειριακή Επικοινωνία UART

Αφού έχουμε μπει σε C κώδικα πλέον, για να μπορέσουμε να έχουμε αλληλεπίδραση με τον πυρήνα, χρειαζόμαστε ένα μέσο επικοινωνίας:

- **UART**: Ασύγχρονη σειριακή επικοινωνία χρησιμοποιώντας δύο συνδέσεις:
  - Ένα άκρο **Rx**, το οποίο λαμβάνει σειριακά δεδομένα.
  - Ένα άκρο **Tx**, το οποίο μεταδίδει σειριακά δεδομένα.



Έπειτα μπορούμε να διαβάσουμε και να γράψουμε ένα χαρακτήρα στην σειριακή κονσόλα, χρησιμοποιώντας τις `uart_getc(), uart_putc()`.

- **`uart_getc()`**: Περιμένει μέχρι να είναι έτοιμη η UART να λάβει, και διαβάζει έναν χαρακτήρα.
- **`uart_putc()`**: Περιμένει μέχρι να είναι έτοιμη η UART να στείλει, και στέλνει έναν χαρακτήρα.

Πλέον μπορούμε να εκτυπώσουμε έναν string στην σειριακή κονσόλα, και να δούμε την έξοδο του:

```
Hello kernel world!
```

Code Reference: `/include/peripherals/uart.h, include/kernel/uart.h, /src/kernel/uart.c`

Interrupts - Vector Table (ARMv6 vs. ARMv8)

Και οι δύο οικογένειες ARM επεξεργαστών ορίζουν κατηγορίες διακοπών, εστιάζουμε στον ARMv8 και τις 4 κατηγορίες του:

- **Synchronous Interrupts**
  - Διακοπές που προκαλούνται από εντολές που εκτελούνται.
  - Όπως software interrupts, όταν πρέπει να εκτελέσει κώδικας με υψηλότερα δικαιώματα.
- **IRQ**
  - Όταν προκληθεί εξωτερική διακοπή, χαμηλής προτεραιότητας.
- **FIQ**
  - Όταν προκληθεί εξωτερική διακοπή, υψηλής προτεραιότητας.
- **System Error**
  - Όταν προκληθεί σφάλμα από εξωτερικές διακοπές.

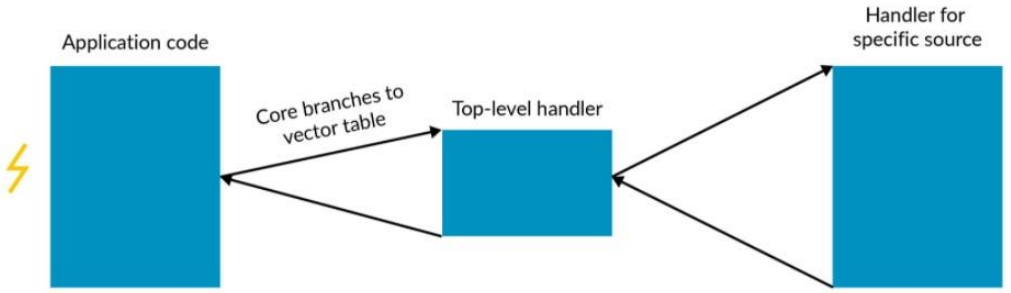
Ο τρόπος που εξυπηρετούνται αυτές οι διακοπές είναι μέσω του Vector Table.

Vector Table:

Κομμάτι κώδικα στην αρχή της μνήμης, όπου περιλαμβάνει εντολές branch στο κατάλληλο handler για την διακοπή που προκλήθηκε.

3ο Στάδιο kernel - Handling Interrupts

- Θέλουμε να διαχειριστούμε προς το παρόν μόνο τα IRQ interrupts.
  - ARMv6: IRQ
  - ARMv8: EL1h IRQ
- Ορίζουμε το Vector Table για κάθε αρχιτεκτονική, όπου σε κάθε διεύθυνση καλούμε τον κατάλληλο handler:
  - Handler για IRQ διακοπές από τον C κώδικα.
  - Error Handler για όλες τις άλλες διακοπές.



Interrupt Handler:

- Στην είσοδο ενός handler, σώζουμε όλους τους καταχωρητές
- Στην έξοδο του handler τους επαναφέρουμε.

Code Reference: `/arch/armv6/entry.S`, `/arch/armv6/irq.S`, `/arch/armv8-a/entry.S`, `/arch/armv8-a/irq.S`, `/include/kernel/irq.c`

4ο Στάδιο kernel - Timer

Θέλουμε να χρησιμοποιήσουμε τους Timers των Pi με διακοπές, για να μπορούμε να περιμένουμε ασύγχρονα, εκτελώντας ταυτόχρονα άλλο κομμάτι κώδικα.

Η λειτουργία του timer είναι η εξής:

- Υπάρχει ένας free-running counter που μετράει συνεχώς.
- Θέτουμε μία τιμή (ο χρόνος που θέλουμε να περιμένουμε) σε κατάλληλο καταχωρητή και όταν φτάσει στην τιμή αυτή ο free-running counter, προκαλεί μια IRQ διακοπή.
- Στον handler για τα Timer interrupts:
    - Δηλώνουμε ότι χειριστήκαμε την διακοπή στον κατάλληλο καταχωρητή.
    - Ανανεώνουμε την τιμή του free-running counter του.
    - Ορίζουμε την τιμή που θέλουμε μέχρι την επόμενη διακοπή.

Code Reference: `/include/peripherals/timer.h, include/kernel/timer.h, /src/kernel/timer.c`

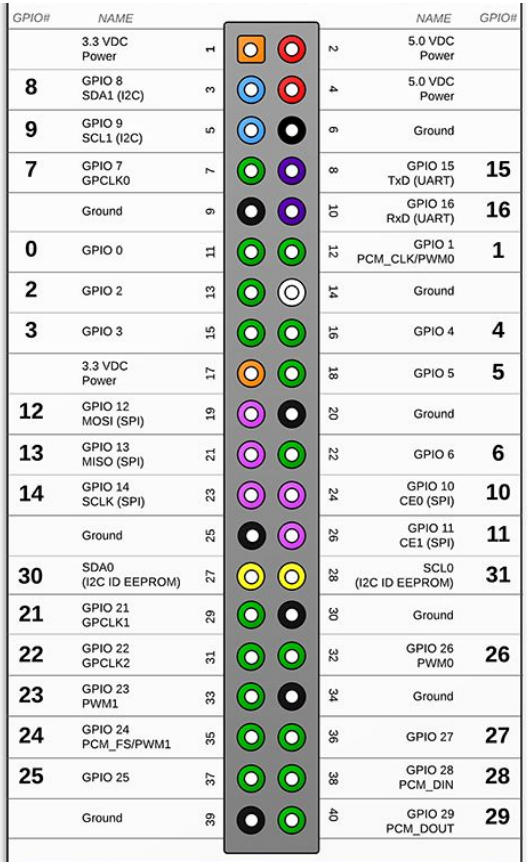
5ο Στάδιο kernel - GPIO Pins / LED

Διαχείριση General Purpose IO (GPIO) Pins των Boards:

- Για θέσιμο High ή Low των Pins.
- Για καθορισμό της λειτουργίας των Pins
  - Input
  - Output
  - Alternative Functions

Υλοποίηση συναρτήσεων για χειρισμό:

- LED σε GPIO pins.
- Onboard LED πάνω στο Pi.



GPIO#	NAME			NAME	GPIO#
	3.3 VDC Power	1	2	5.0 VDC Power	
8	GPIO 8 SDA1 (I2C)	3	4	5.0 VDC Power	
9	GPIO 9 SCL1 (I2C)	5	6	Ground	
7	GPIO 7 GPCLK0	7	8	GPIO 15 TxD (UART)	15
	Ground	9	10	GPIO 16 RxD (UART)	16
0	GPIO 0	11	12	GPIO 1 PCM_CLK/PWM0	1
2	GPIO 2	13	14	Ground	
3	GPIO 3	15	16	GPIO 4	4
	3.3 VDC Power	17	18	GPIO 5	5
12	GPIO 12 MOSI (SPI)	19	20	Ground	
13	GPIO 13 MISO (SPI)	21	22	GPIO 6	6
14	GPIO 14 SCLK (SPI)	23	24	GPIO 10 CE0 (SPI)	10
	Ground	25	26	GPIO 11 CE1 (SPI)	11
30	SDA0 (I2C ID EEPROM)	27	28	SCL0 (I2C ID EEPROM)	31
21	GPIO 21 GPCLK1	29	30	Ground	
22	GPIO 22 GPCLK2	31	32	GPIO 26 PWM0	26
23	GPIO 23 PWM1	33	34	Ground	
24	GPIO 24 PCM_FS/PWM1	35	36	GPIO 27	27
25	GPIO 25	37	38	GPIO 28 PCM_DIN	28
	Ground	39	40	GPIO 29 PCM_DOUT	29

Code Reference: `/include/peripherals/gpio.h`, `include/kernel/gpio.h`, `/src/kernel/gpio.c`, `include/kernel/led.h`, `/src/kernel/led.c`

6ο Στάδιο kernel - Scheduler

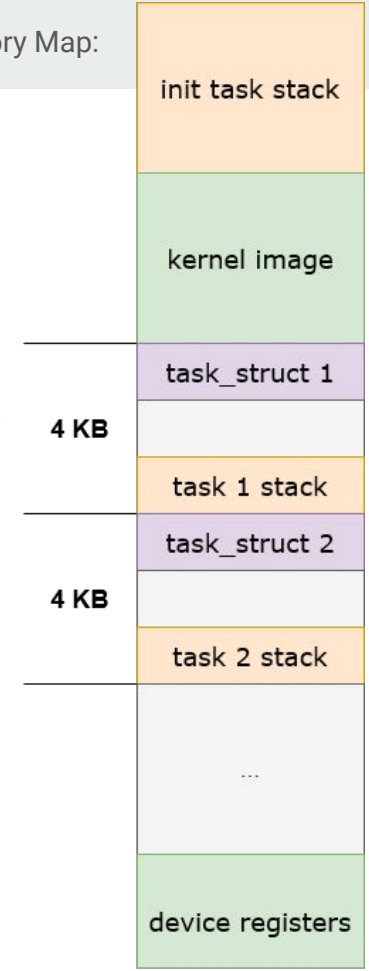
Θέλουμε να μπορούμε να εκτελούμε διαφορετικές διεργασίες concurrently, χωρίς να επηρεάζεται η μία από την άλλη.

- Ορίζουμε ένα **`task_struct`**, που έχει όλες τις πληροφορίες που χρειάζεται για μια διεργασία.

task_struct

cpu_context	state	priority	counter	preemptable

- Ορίζουμε συνάρτηση **`copy_process()`**, η οποία δημιουργεί μια διεργασία με όρισμα την συνάρτηση που θα εκτελέσει.
- Δημιουργούμε συναρτήσεις για **allocation** και **απελευθέρωση** μνήμης (**4KB page**) για μία διεργασία.



Code Reference: `/arch/armv6/scheduler.S`, `/arch/armv8-a/scheduler.S`, `/src/kernel/scheduler.c`, `src/kernel/fork.c`, `/src/kernel/mm.c`, `/src/kernel/mm.c`, `/src/kernel/process.c`

6ο Στάδιο kernel - Scheduler

- Δημιουργούμε συνάρτηση **`schedule()`**, η οποία είναι υπεύθυνη για την χρονοδρομολόγηση των διεργασιών, η οποία καλείται:
  - Είτε από την αρχική διεργασία (**init task**)
  - Είτε από τον **interrupt handler του timer**, έπειτα από κάποιο χρόνο που του ορίζουμε.

Το context switch, περιλαμβάνει την εξής διαδικασία:

- Σώσιμο καταχωρητών `current` διεργασίας και επαναφορά καταχωρητών `next` διεργασίας.
- Ο program counter δείχνει πλέον εκεί που είχε μείνει η διεργασία όταν σταμάτησε.

Αλγόριθμος Scheduler:

```
Loop Forever:
    max_counter = -1
    next = 0

    for all tasks:
        if (task->state == TASK_RUNNING && task->counter > max_counter):
            // Update maximum counter, and pid for next task
            max_counter = task->counter
            next = task
    // If a task to be scheduled is found, break and switch to task
    if (max_counter > 0):
        break

    // If no such task is found
    for all tasks:
        Increase task->counter

// Finally we switch to the selected task
switch_to(task[next]);
```

Code Reference: `/arch/armv6/scheduler.S`, `/arch/armv8-a/scheduler.S`, `/src/kernel/scheduler.c`, `src/kernel/fork.c`, `/src/kernel/mm.c`, `/src/kernel/mm.c`, `/src/kernel/process.c`

7ο Στάδιο kernel - Console

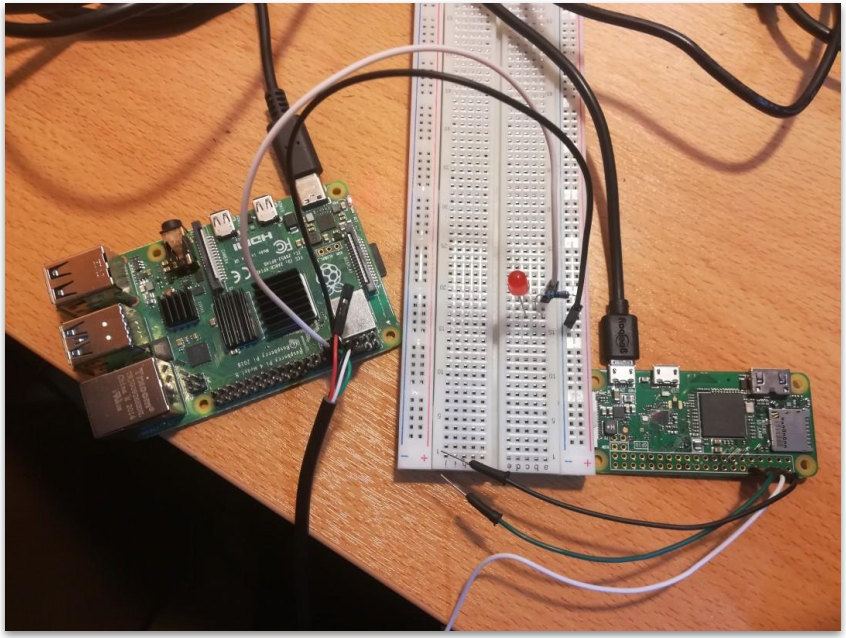
Τέλος, έχουμε υλοποίηση ενός βασικού console για:

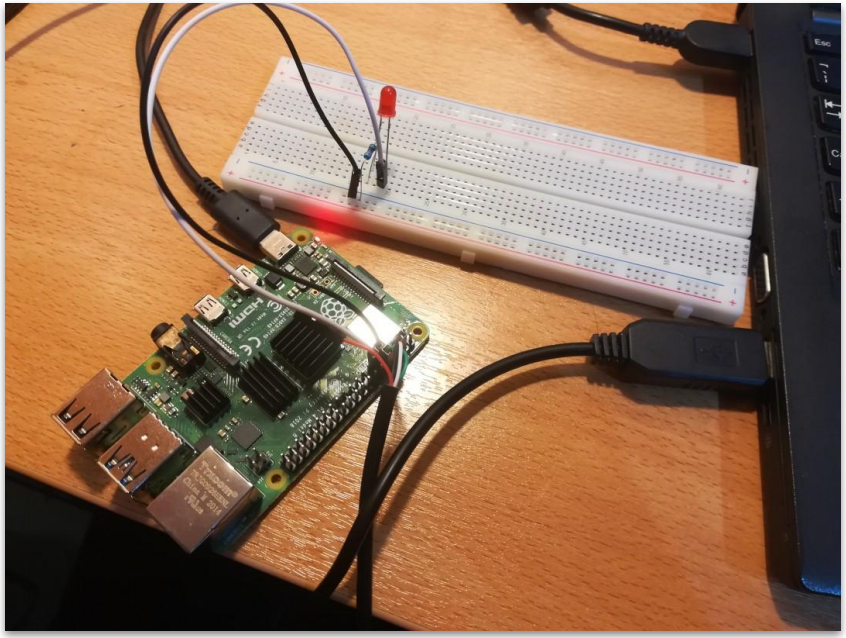
- Διάβασμα εισόδου από χρήστη.
- Εκτέλεση υπαρχόντων εντολών.
- Εμφάνιση αποτελεσμάτων εξόδου.

```
This is a minimal console, type 'help' to see the available commands. (Maximum Input Length: 80)
root@pi-4# help
Available commands:
    help:
        Prints available commands to the console.
    help_led:
        Prints available LED commands to the console.
    create_procs:
        Creates proc_num kernel processes.
    run_procs:
        Runs the created kernel processes concurrently.
    kill_procs:
        Kills all created kernel processes.
    halt:
        Halts the system.
root@pi-4#
```

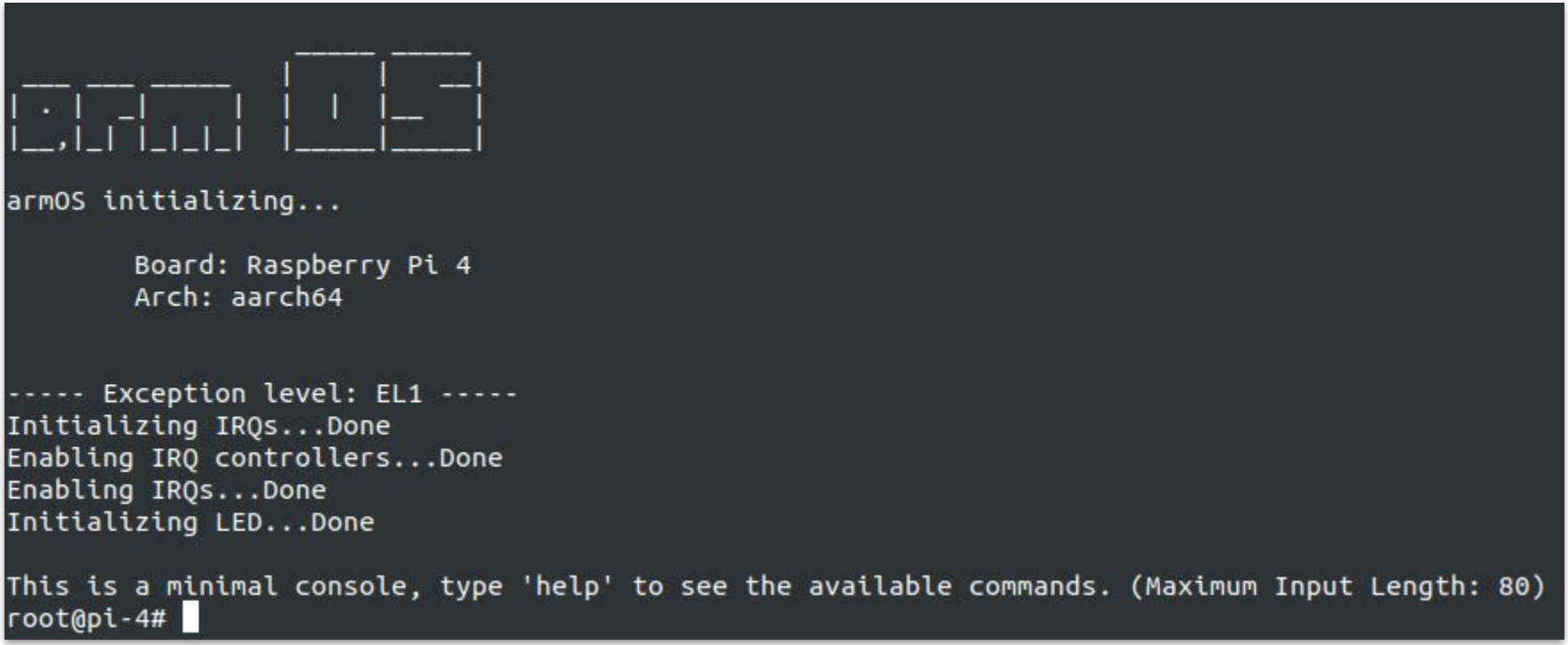
Code Reference: `src/kernel/console.c, /include/kernel/console.c`

Kernel in action



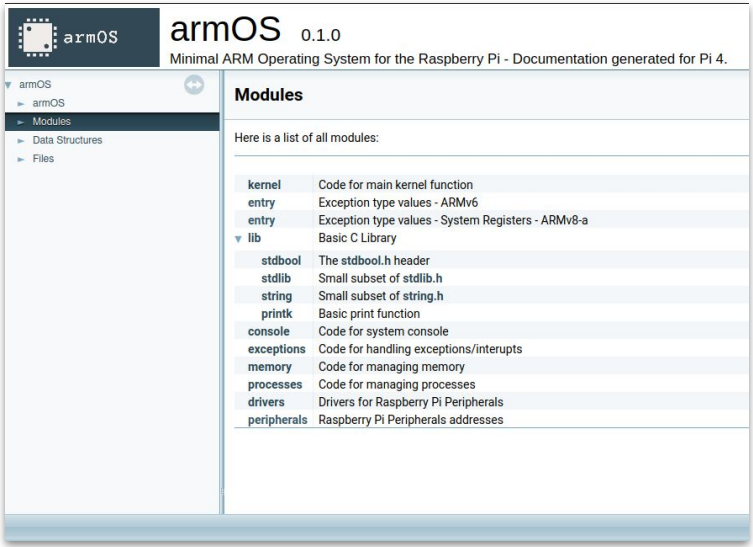


Kernel in action

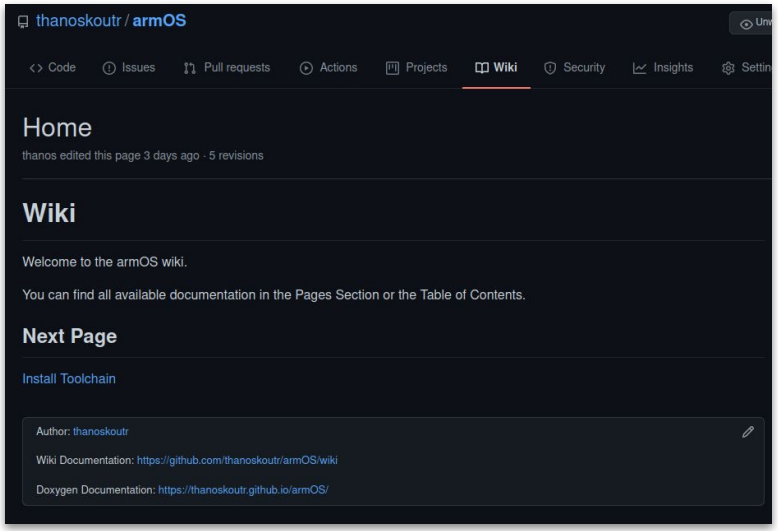


Documentation

Υπάρχει αναλυτικό documentation του project σε δύο format.



Στο GitHub Pages του GitHub repository του project.



Στο Wiki section του GitHub repository του project.

Links

GitHub Repository: https://github.com/thanoskoutr/armOS

Wiki Documentation: https://github.com/thanoskoutr/armOS/wiki

Doxygen Documentation: https://thanoskoutr.github.io/armOS/

Resources

https://github.com/thanoskoutr/armOS#resources