Apache NuttX

RTOS temellerinden ESP32 üzerinde çalışan uygulamaya. İki gün, altı modül, altı uygulama.

Nazım KOÇ · 2 gün × 6 saat

Program

01	RTOS temelleri	2.0 saat
02	NuttX, build sistemi ve menuconfig	2.0 saat
03	NSH ve kendi uygulaman	2.0 saat
04	POSIX API: task, scheduler, senkronizasyon	2.5 saat
05	Dosya sistemi: littlefs ve procfs	1.5 saat
06	I2C ile IMU okuma — bitirme projesi	2.0 saat

Nasıl çalışacağız

Yarısı anlatım

Her modül kısa bir kavram bloğuyla açılır. Slaytlardaki her kod parçası lab föyünde tam haliyle var; yazmaya çalışmak yerine dinleyin.

Yarısı klavye

Altı lab, hepsi kendi kartınızda. Lab föyündeki adımları sırayla uygulayın; her labın sonunda beklenen çıktı yazılı.

Bitirme projesi

Son modülde I2C üzerinden bir IMU okuyup ölçümü kabuktan görüntüleyen bir uygulama yazacaksınız. Önceki beş labın toplamı.

Ön hazırlık

- Ubuntu 22.04+ veya WSL2 — Windows üzerinde doğrudan build desteklenmez
- ESP32 kartı (DevKitC, S3 veya C3 fark etmez), USB kablo, breadboard ve jumper
- Bir IMU modülü: MPU6050 veya benzeri, I2C çıkışlı
- Seri porta erişim: kullanıcı `dialout` grubunda olmalı
- Yaklaşık 5 GB boş disk ve toolchain indirmesi için internet

01

RTOS temelleri

Task, scheduler, öncelik. İki saat, kod yok.

Neden RTOS?

Tek döngünün sınırı

Bare-metal bir programda işler sırayla çalışır. Bir işin içindeki bekleme diğer bütün işleri geciktirir. İş sayısı arttıkça zamanlama elle yönetilemez hale gelir ve kod state machine yığınına dönüşür.

RTOS'un teklifi

Her işi kendi akışında, sanki tek iş varmış gibi yazarsınız. Hangisinin ne zaman çalışacağına scheduler karar verir. Bekleme, CPU'yu kilitlemek yerine sırayı bir başkasına devreder.

Aynı iş, iki yaklaşım

BARE-METAL

```
while (1)
  {
    led_toggle();
    delay_ms(1000);
    /* CPU 1 saniye kilitli */

    if (button_pressed())
      {
        beep();
        /* 1 sn'ye kadar gecikme */
      }
  }
```

RTOS

```
void *led_task(void *arg)
  {
    while (1)
      {
        led_toggle();
        usleep(1000000);
        /* yalnızca bu akış bekler */
      }
  }

void *button_task(void *arg)
  {
    while (1)
      {
        wait_for_button();
        beep();  /* anında */
      }
  }
```

Task nedir?

- Task, kendi stack'i ve kendi çalışma durumu olan bağımsız bir kod akışıdır
- Her task'ın bir önceliği ve bir durumu vardır: çalışıyor, hazır veya bekliyor
- Tek çekirdekte aynı anda yalnızca bir task koşar; geri kalanlar sırasını bekler
- Task değişimine context switch denir — kayıtlar saklanır, diğerinin kayıtları yüklenir
- NuttX'te thread'ler aynı adres alanını ve dosya tanıtıcılarını paylaşır, task'lar paylaşmaz

Task durumları

DURUM	ANLAMI	BURADAN NASIL ÇIKAR
`Running`	Şu anda CPU'da çalışıyor	Bekleme çağrısı yapar veya preempt edilir
`Ready`	Çalışmaya hazır, sırasını bekliyor	Scheduler seçtiğinde Running olur
`Waiting`	Bir olayı bekliyor: semaphore, mesaj, zaman	Olay gerçekleşince Ready olur
`Inactive`	Oluşturuldu ama henüz başlatılmadı	Aktive edildiğinde Ready olur

Scheduler ve öncelik

- Kural tek cümle: hazır olan en yüksek öncelikli task çalışır
- Preemption — yüksek öncelikli bir task hazır olur olmaz çalışan task durdurulur
- NuttX'te öncelik aralığı 1–255, varsayılan değer 100
- Aynı öncelikli task'lar için FIFO veya zaman dilimli round-robin seçilir
- Düşük öncelikli bir task hiç çalışamıyorsa bu bir tasarım hatasıdır, scheduler hatası değil

Gerçek zamanlı, hızlı demek değil

Ölçü ortalama gecikme değil, garanti edilen en kötü durum gecikmesidir. Bir işin her seferinde 2 ms içinde bitmesi, çoğu zaman 0,1 ms sürüp bazen 50 ms sürmesinden iyidir.

Lab 1 · Kağıt üzerinde tasarım

Bir sulama kontrolcüsünü task'lara bölün.

01 Cihaz her 2 saniyede nem okuyacak, ekranı 200 ms'de bir yenileyecek, vana butonuna 50 ms içinde tepki verecek

02 Kaç task olsun? Her birine bir isim ve bir görev cümlesi yazın

03 Öncelikleri sıralayın ve her sıralamanın gerekçesini tek cümleyle yazın

04 Hangi iki task aynı veriye dokunuyor? İşaretleyin — modül 04'te buraya döneceğiz

02

NuttX, build sistemi ve menuconfig

Kaynak ağacı, üç komutluk build,
konfigürasyon.

NuttX nedir?

- Apache lisanslı, gerçek zamanlı işletim sistemi — küçük mikrodenetleyiciden 64-bit işlemciye kadar
- Ayırt edici yanı POSIX ve ANSI C uyumu: Linux'ta bildiğiniz API'lerin aynısı
- Yapılandırmayla büyür ve küçülür: birkaç KB'lik çekirdekten dosya sistemli, ağ yığınlı sisteme kadar
- ESP32, ESP32-S3, ESP32-C3 ve C6 dahil yüzlerce kart destekli
- Beraberinde Linux tarzı bir kabuk, sürücü modeli ve `/dev` ağacı gelir

Aynı kod, iki hedef

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

static void *worker(void *arg)
{
  for (int i = 0; i < 3; i++)
    {
      printf("worker: %d\n", i);
      usleep(500000);
    }
  return NULL;
}
```

Bu dosya ana makinede `gcc` ile, ESP32'de NuttX build sistemiyle derlenir. Tek satır değişmez — RTOS'a özel bir API öğrenmeniz gerekmez.

Nerede duruyor?

ÖLÇÜT	NUTTX	FREERTOS / ESP-IDF	EMBEDDED LINUX
Uygulama API'si	POSIX, standart C	Kendi API'si	POSIX
Tipik RAM ihtiyacı	Onlarca KB	Onlarca KB	Onlarca MB
Gerçek zaman	Sert	Sert	Yumuşak
Kabuk ve dosya sistemi	Dahili	Sınırlı	Tam
Donanım taşınabilirliği	Yüksek	Çip ailesine bağlı	MMU gerektirir

Kaynak ağacı

```
nuttxspace/
├── nuttx/                      ← çekirdek
│   ├── arch/                   işlemci mimarileri
│   ├── boards/                 kart tanımları
│   ├── sched/                  scheduler
│   ├── fs/  drivers/  net/
│   └── include/                POSIX başlıkları
└── apps/                       ← nuttx-apps deposu
    ├── examples/               hazır örnekler
    ├── nshlib/                 kabuk
    └── system/
```

Kendi uygulamanız `apps/` altında kendi dizinine girer — modül 03'te yazacağız.

Kurulum

```
# bağımlılıklar
sudo apt install git make ninja-build gperf flex bison \
     libncurses-dev kconfig-frontends python3-pip

# kaynak kod: iki depo, kardeş dizin
mkdir nuttxspace && cd nuttxspace
git clone https://github.com/apache/nuttx.git nuttx
git clone https://github.com/apache/nuttx-apps.git apps

# ESP32 araçları
pip3 install esptool kconfiglib
```

Toolchain'i Espressif'in hazır paketinden kurup `PATH`'e ekleyin: Xtensa kartlar için `xtensa-esp32-elf`, RISC-V kartlar için `riscv-none-elf`.

İlk build üç komut

```
cd nuttx

# 1. kart ve konfigürasyon seç
./tools/configure.sh esp32-devkitc:nsh
#   ESP32-S3 için: esp32s3-devkit:nsh#   ESP32-C3 için: esp32c3-devkit:nsh# 2. derle
make -j$(nproc)

# 3. karta yaz
make flash ESPTOOL_PORT=/dev/ttyUSB0
```

Konfigürasyon adı bir hazır profildir. `nsh` profili kabuğu içeren en yaygın başlangıç noktasıdır.

menuconfig ile yapılandırma

```
make menuconfig
```

- `/` tuşu ile sembol arayın — aradığınız seçeneğin nerede olduğunu söyler
- Sonuç `.config` dosyasına yazılır; bu dosya derlemenin tek gerçeğidir
- `make savedefconfig` ile küçük ve paylaşılabilir bir profil çıkarılır
- Bir seçenek görünmüyorsa bağımlılığı kapalıdır — arama sonucu `depends on` satırını gösterir
- Konfigürasyon değişince `make` yeterlidir; kart değişirse `make distclean` gerekir

Lab 2 · İlk build ve flash

Kartınızda NSH promptunu görün.

01 Kaynak ağacını kurun ve kartınıza uygun hedefi `configure.sh` ile seçin

02 `make -j$(nproc)` ile derleyin, üretilen binary'nin boyutunu not edin

03 Karta yazın ve seri monitörü açın: `picocom -b 115200 /dev/ttyUSB0`

04 `nsh>` promptunda `help` yazın ve komut listesini görün

05 `make menuconfig` ile bir örneği açıp yeniden derleyin, boyut değişimini karşılaştırın

03

NSH ve kendi uygulaman

Kabuk, /dev ağacı, uygulama ekleme.

NSH nedir?

- NuttShell, sistemin içinde çalışan küçük bir kabuk — bash'in tanıdık komutlarını taşır
- Kendisi bir uygulamadır; çekirdeğin parçası değildir, kapatılabilir
- Yapılandırmada etkinleştirilen her uygulama NSH'de bir komut olarak görünür
- Seri port üzerinden konuşur; ağ etkinse telnet ile de erişilebilir
- Geliştirme sırasında en hızlı hata ayıklama aracınız burasıdır

Sık kullanılan komutlar

KOMUT	NE YAPAR	ÖRNEK
`help`	Etkin komutları listeler	`help`
`ls`	Dizin ve aygıtları gösterir	`ls /dev`
`free`	Kalan bellek	`free`
`ps`	Task listesi ve durumları	`ps`
`cat`	Dosya veya aygıt okur	`cat /proc/version`
`uname`	Sistem bilgisi	`uname -a`

Donanım da bir dosya

```
nsh> ls /dev
/dev:
 console
 i2c0            ← I2C veriyolu
 null
 ttyS0           ← seri port
 userleds        ← LED sürücüsü

nsh> cat /proc/version
```

Her aygıt `/dev` altında bir dosyadır. Erişim `open`, `read`, `write`, `ioctl` ile yapılır — donanıma özel bir API öğrenmeye gerek yoktur.

Uygulama iskeleti

apps/examples/selam/selam_main.c

```
#include <nuttx/config.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Selam, NuttX!\n");

  for (int i = 1; i < argc; i++)
    {
      printf("  argüman %d: %s\n", i, argv[i]);
    }

  return 0;
}
```

Uygulamayı build'e bağlamak

KCONFIG

```
config EXAMPLES_SELAM
  tristate "Selam örneği"
  default n
  ---help---
    Basit bir NSH komutu.

if EXAMPLES_SELAM

config EXAMPLES_SELAM_PROGNAME
  string "Komut adı"
  default "selam"

config EXAMPLES_SELAM_STACKSIZE
  int "Stack boyutu"
  default 2048

endif
```

MAKEFILE

```
include $(APPDIR)/Make.defs

PROGNAME  = $(CONFIG_EXAMPLES_SELAM_PROGNAME)
PRIORITY  = SCHED_PRIORITY_DEFAULT
STACKSIZE = $(CONFIG_EXAMPLES_SELAM_STACKSIZE)
MODULE    = $(CONFIG_EXAMPLES_SELAM)

MAINSRC = selam_main.c

include $(APPDIR)/Application.mk
```

MAKE.DEFS

```
ifneq ($(CONFIG_EXAMPLES_SELAM),)
CONFIGURED_APPS += $(APPDIR)/examples/selam
endif
```

Lab 3 · Kendi komutun

NSH'den çağrılan bir komut yazın.

01 `apps/examples/selam/` dizinini ve dört dosyasını oluşturun

02 `make menuconfig` → Application Configuration → Examples altında komutu etkinleştirin

03 Derleyip yazın, `nsh> selam dünya` komutunu çalıştırın

04 Komutu argümanlarını toplayacak şekilde değiştirin ve yeniden derleyin

05 `free` çıktısını komuttan önce ve sonra karşılaştırın

04

POSIX API: task, scheduler, senkronizasyon

pthread, semaphore, mutex, message queue.

İki bağımsız akış

```
static void *blinker(void *arg)
{
  const char *isim = (const char *)arg;

  while (1)
    {
      printf("%s çalıştı\n", isim);
      usleep(500000);
    }
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t1, t2;
  pthread_create(&t1, NULL, blinker, "hızlı");
```

Öncelik ve stack belirlemek

```
pthread_attr_t attr;
struct sched_param param;

pthread_attr_init(&attr);

param.sched_priority = 120;          /* 1–255, varsayılan 100 */
pthread_attr_setschedparam(&attr, &param);
pthread_attr_setstacksize(&attr, 4096);

pthread_create(&tid, &attr, sensor_task, NULL);
```

Stack boyutu yetersizse sistem uyarı vermeden çöker. Yeni bir thread'i 2048 bayt ile başlatıp `ps` çıktısındaki kullanım oranına bakarak ayarlayın.

Paylaşılan veri bozulur

```
static volatile int sayac = 0;

static void *artir(void *arg)
{
  for (int i = 0; i < 100000; i++)
    {
      sayac++;      /* oku, artır, yaz — üç adım */
    }
  return NULL;
}

/* iki thread çalıştırıldığında sonuç
   200000 değil, her seferinde farklı bir sayı */
```

Context switch üç adımın arasında gerçekleşebilir. İki akışın aynı veriye dokunduğu her yer korunmalıdır.

Mutex ile korumak

```
static pthread_mutex_t kilit = PTHREAD_MUTEX_INITIALIZER;
static int sayac = 0;

static void *artir(void *arg)
{
  for (int i = 0; i < 100000; i++)
    {
      pthread_mutex_lock(&kilit);
      sayac++;                          /* kritik bölüm */
      pthread_mutex_unlock(&kilit);
    }
  return NULL;
}
```

Kritik bölümü kısa tutun. İçinde `printf` veya bekleme çağrısı yapmayın — kilidi tutarken beklemek bütün sistemi yavaşlatır.

Semaphore ile sinyal vermek

```
static sem_t hazir;

sem_init(&hazir, 0, 0);          /* başlangıç sayacı 0 *//* üreten taraf: veri hazır olduğunda */
sem_post(&hazir);

/* tüketen taraf: sinyal gelene kadar bekler */
sem_wait(&hazir);
printf("veri geldi\n");
```

Mutex sahiplik kurar, semaphore sayar. Bir olayı haber vermek için semaphore, bir kaynağı korumak için mutex kullanın.

Message queue ile veri taşımak

ÜRETEN

```
struct mq_attr attr =
  {
    .mq_maxmsg  = 8,
    .mq_msgsize = sizeof(int),
  };

mqd_t mq = mq_open("/olcum",
    O_WRONLY | O_CREAT, 0666, &attr);

int deger = 42;
mq_send(mq, (char *)&deger,
        sizeof(deger), 0);
```

TÜKETEN

```
mqd_t mq = mq_open("/olcum",
                   O_RDONLY);

int deger;
unsigned int prio;

/* mesaj gelene kadar bekler */
mq_receive(mq, (char *)&deger,
           sizeof(deger), &prio);

printf("ölçüm: %d\n", deger);
```

Hangisi ne zaman

MEKANİZMA	NE İÇİN	TİPİK KULLANIM
`mutex`	Paylaşılan veriyi korumak	İki task aynı struct'a yazıyor
`semaphore`	Olay bildirmek, kaynak saymak	Interrupt task'ı uyandırıyor
`message queue`	Veri taşımak	Sensör → işleyici akışı
`signal`	Kesme benzeri bildirim	Zaman aşımı, iptal

Lab 4 · İki task, bir kuyruk

Üreten ve tüketen task'ları birbirine bağlayın.

01 Sayaç deneyini mutex'siz çalıştırın ve sonucu not edin, sonra mutex ekleyip tekrarlayın

02 Saniyede bir artan sayı üreten bir thread yazın ve kuyruğa gönderin

03 Kuyruktan okuyup ekrana yazan ikinci bir thread yazın

04 Tüketen thread'in önceliğini üretenin altına indirin, davranış değişimini gözleyin

05 Kuyruk boyutunu 1'e düşürüp üretimi hızlandırın: `mq_send` ne yapıyor?

05

Dosya sistemi: littlefs ve procfs

Kalıcı depolama ve sistemi içinden izlemek.

Hangi dosya sistemi

DOSYA SİSTEMİ	NEREDE DURUR	NE İÇİN
`littlefs`	Flash	Güç kesintisine dayanıklı kalıcı depolama
`tmpfs`	RAM	Geçici dosyalar, yeniden başlatmada silinir
`procfs`	Sanal	Sistem durumu: task, bellek, sürücü
`romfs`	Flash, salt okunur	Sabit yapılandırma ve betikler

littlefs ile kalıcı depolama

```
# kabuktan: bir kez biçimlendir, sonra mount et
nsh> mkfatfs /dev/esp32flash
nsh> mount -t littlefs /dev/esp32flash /data
nsh> echo "olcum" > /data/kayit.txt
nsh> cat /data/kayit.txt
```

```
/* koddan: standart C dosya işlemleri */
FILE *f = fopen("/data/kayit.txt", "a");
fprintf(f, "%d,%d\n", zaman, deger);
fclose(f);
```

procfs ile sistemi izlemek

```
nsh> ps
  PID  PRI POLICY   TYPE    NPX STATE    STACK  USED  COMMAND
    0    0 FIFO     Kthread   - Ready    003056 000704 Idle_Task
    1  100 RR       Task      - Running  002988 001824 nsh_main
    2  120 FIFO     pthread   - Waiting  004080 000912 sensor_task

nsh> cat /proc/2/status
nsh> free
```

STACK ve USED sütunları stack boyutunu ayarlamanın tek güvenilir yoludur. Kullanım %80'i geçiyorsa büyütün.

Lab 5 · Ölçümü diske yaz

Lab 4'ün kuyruğunu dosyaya bağlayın.

01 menuconfig'den littlefs ve procfs desteğini etkinleştirin

02 Flash bölümünü `/data` altına mount edin ve kabuktan bir dosya yazın

03 Tüketen thread'i, aldığı her değeri `/data/olcum.csv` dosyasına ekleyecek şekilde değiştirin

04 Kartı yeniden başlatın ve dosyanın yerinde olduğunu doğrulayın

05 `ps` çıktısında kendi thread'lerinizin stack kullanımını okuyun ve boyutu ayarlayın

06

I2C ile IMU okuma

Bitirme projesi: sensörden kabuğa.

I2C NuttX'te nasıl görünür

- Veriyolu `/dev/i2c0` olarak açılır; menuconfig'de I2C ve I2C karakter sürücüsü etkinleştirilir
- Sensörün hazır sürücüsü varsa doğrudan bir aygıt dosyası olarak görünür ve `read` ile okunur
- Sürücü yoksa `ioctl` ile ham register okuma yapılır — IMU için bu yolu kullanacağız
- Kablolama: SDA, SCL, 3V3 ve GND. Pin numaraları menuconfig'de kart bölümünde tanımlıdır
- `i2ctool` ile kod yazmadan veriyolunu taramak ilk adımdır

Önce veriyolunu tara

```
nsh> i2c dev 0x03 0x77
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
                            ^^ IMU burada

# WHO_AM_I register'ını oku (MPU6050: 0x75)
nsh> i2c get -a 0x68 -r 0x75
READ Bus: 0 Addr: 68 Value: 68
```

Adres listede yoksa problem koddan değil kablolamadan kaynaklanır. Bu iki komut, sensörün canlı olduğunun kanıtıdır.

Register okumak

```
int fd = open("/dev/i2c0", O_RDONLY);

uint8_t reg = 0x3b;                    /* ACCEL_XOUT_H */
uint8_t buf[6];

struct i2c_msg_s msg[2] =
  {
    { .frequency = 400000, .addr = 0x68, .flags = 0,
      .buffer = &reg, .length = 1 },
    { .frequency = 400000, .addr = 0x68, .flags = I2C_M_READ,
      .buffer = buf,  .length = 6 },
  };

struct i2c_transfer_s xfer = { .msgv = msg, .msgc = 2 };
ioctl(fd, I2CIOC_TRANSFER, (unsigned long)&xfer);

int16_t ax = (buf[0] << 8) | buf[1];   /* big-endian */
printf("ax = %d\n", ax);
```

Bitirme projesi

IMU okuyan, kaydeden ve kabuktan sorgulanan bir uygulama.

01	I2C'yi etkinleştirin, `i2ctool` ile IMU adresini doğrulayın
02	Sensörü uyandırın: `PWR_MGMT_1` register'ına 0 yazın
03	100 ms'de bir üç eksen ivme okuyan bir thread yazın, değerleri kuyruğa gönderin
04	Kuyruktan okuyan thread değerleri `/data/imu.csv` dosyasına yazsın
05	Son okumayı ekrana basan bir NSH komutu ekleyin: `nsh> imu`

Sık yapılan hatalar

BELİRTİ	SEBEBİ	ÇÖZÜM
Uygulama menuconfig'de yok	Make.defs eksik	CONFIGURED_APPS satırını ekleyin
Açıklanamayan çökme	Stack yetersiz	`ps`
Bir task hiç çalışmıyor	Öncelik sıralaması	Yüksek öncelikliye bekleme ekleyin
Sayaç yanlış artıyor	Korumasız paylaşılan veri	Mutex ile sarın
I2C aygıtı görünmüyor	Kablolama veya pin ayarı	`i2c dev`
Kart değişince build hatası	Eski .config	`make distclean`

Buradan sonra

- Resmî dokümantasyon: `nuttx.apache.org/docs` — kart tanımları ve sürücü listesi
- `apps/examples/` dizini: her alt dizin çalışan bir örnek
- Sürücü yazmak: `drivers/sensors/` içindeki mevcut bir sürücüyü şablon alın
- Topluluk: dev@nuttx.apache.org listesi ve Apache NuttX Discord kanalı
- Sonraki adım önerisi: kendi sensörünüz için bir sürücü yazıp `/dev` altına bağlayın

Teşekkürler

Lab föyü, kod örnekleri ve ön hazırlık dokümanı ayrı dosyalarda. Sorular için iletişimde kalın.

eposta@ornek.com