

REDES INALÁMBRICAS DE SENSORES

MÁSTER EN INGENIERÍA ELECTRÓNICA, ROBÓTICA Y AUTOMÁTICA

PRÁCTICA 1. INTRODUCCIÓN A LA PROGRAMACIÓN EN EL SISTEMA OPERATIVO CONTIKI

Eduardo Hidalgo Fort

ehidalgo@us.es

José María Hinojo Montero

jhinojo@us.es

Dpto. Ingeniería
Electrónica



ÍNDICE

1. Generalidades y conceptos de ContikiOS
2. Primera aplicación (Hello World)
3. Aspectos básicos de programación
 1. Temporizadores
 2. Procesos y protohilos
4. Interfaces de comunicación: adquisición de sensores

I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

- SO **multiplataforma** de código abierto (licencia BSD)

<https://github.com/contiki-os/contiki/releases>

- Escrito en lenguaje de programación C
- El desarrollo fue iniciado por Adam Dunkels en 2003
- Actualmente se desarrolla en un entorno colaborativo:
 - Cualquier usuario puede sincronizar su copia de trabajo con el repositorio
- Enfocado a redes inalámbricas sobre SoC con restricciones:
 - Bajo consumo energético
 - Capacidad de almacenamiento reducida
 - Baja latencia
- Está compuesto de un *Núcleo de Sistema* y de *Aplicaciones*



Contiki

The Open Source OS for the Internet of Things

I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

- El SO incluye los siguientes “paquetes”:
 - Núcleo multitarea
 - Protohilos
 - Stack de comunicaciones TCP/IP, con soporte IPv6
 - Interfaz de Usuario (GUI)
 - Navegador web
 - Cliente Telnet
- Portado a distintos entornos:
 - Microcontroladores: Atmel, NXP, Nordic, TI, Microchip, ST, ...
 - Ordenadores: Apple II, Intel, Atari, ...
 - Consolas: Atari, Nintendo, NEC, ...
- En la actualidad:
 - CONTIKI
 - CONTIKI-NG



Contiki

The Open Source OS for the Internet of Things



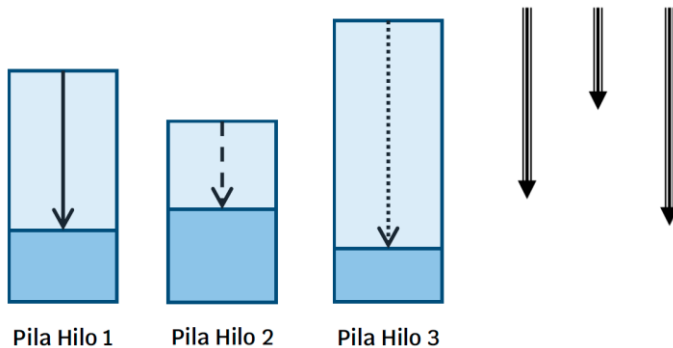
CONTIKI
NEXT GENERATION

<https://github.com/contiki-ng/contiki-ng>

I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

• Programación en Hilos

- Ejecución secuencial en cada hilo
- Mayor ocupación de memoria



• Programación en Eventos

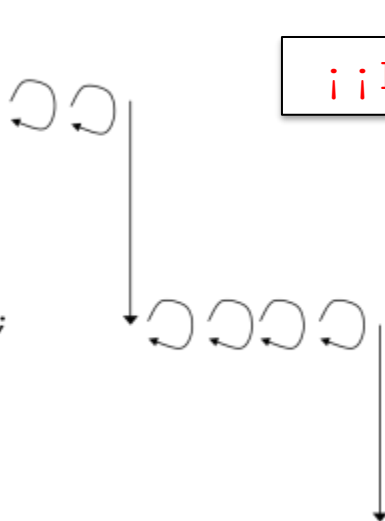
- La ejecución salta en cada evento
- Todos los manejadores usan la misma pila de memoria



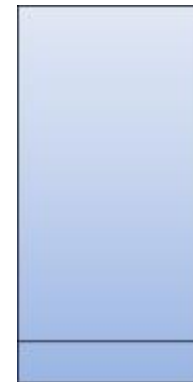
I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

- Contiki introduce una nueva abstracción: Programación basada en PROTOHILOS.
- Solución de compromiso entre hilos y eventos:
 - Pila única de memoria -> Menores requerimientos de almacenamiento
 - Flujo de control secuencial en cada protohilo -> Sin máquina de estados explícita

```
int a_protothread(struct pt *pt) {  
    PT_BEGIN(pt);  
    /* ... */  
    PT_WAIT_UNTIL(pt, condition1);  
    /* ... */  
    if(something) {  
        /* ... */  
        PT_WAIT_UNTIL(pt, condition2);  
        /* ... */  
    }  
    PT_END(pt);  
}
```



;;Pila de memoria única!!



I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

- El repositorio se encuentra en ~/contiki-ng
- Contiene 5 directorios
 - **arch:** Archivos dependientes de los dispositivos
 - **CPU:** Drivers de los diferentes microcontroladores soportados
 - **Dev:** Controladores para periféricos (sensores, leds, etc.)
 - **Platform:** Drivers de placas con soporte de Contiki (nRF52840, Z1, etc.)
 - **examples:** Ejemplos de uso de diferentes características del SO
 - **os:** Ficheros propios del SO Contiki
 - **Sys:** Gestión de procesos, temporizadores, semáforos, etc.
 - **Net:** funciones de red (rutas, seguridad, etc.)
 - **test:** Códigos de realización de pruebas
 - **tools:** Herramientas disponibles



I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

- **Instalación de máquina virtual**
 - Instalar VirtualBox (<https://www.virtualbox.org/wiki/Downloads>)
 - Descargar e instalar la máquina virtual (https://gitlab.com/laboratorio-ris/contiki-ng/-/blob/master/doc/es/prepare-vm_virtualbox.md)
 - Creación de espacio de trabajo y clonación en máquina virtual (<https://gitlab.com/laboratorio-ris/contiki-ng>)
 - El proceso de creación y configuración se detalla en https://gitlab.com/laboratorio-ris/contiki-ng/-/blob/master/doc/es/prepare-vm_virtualbox.md
 - Realizar test inicial (<https://gitlab.com/laboratorio-ris/contiki-ng>)

I. GENERALIDADES Y CONCEPTOS DE CONTIKI OS

- Ejercicio 1: Búsqueda de ficheros de la plataforma nRF52840 (1 punto)

Localice y escriba la ruta de los drivers de cada uno de los chips que componen la placa que se utiliza en las prácticas. Incluya tanto los ficheros fuente como los ficheros de cabecera.

- Microcontrolador
- Transceiver radio
- Almacenamiento externo
- Almacenamiento interno

II. PRIMERA APLICACIÓN (HELLO WORLD)

Primera Aplicación: Hello World

- Dirigirse al directorio del ejemplo: `~/contiki-ng/examples/hello-world`
- La aplicación contiene:
 - Archivo de información/ayuda para el ejemplo: `README.md`
 - Fichero fuente en lenguaje C: `hello-world.c`
 - Información para la compilación: `Makefile`

II. PRIMERA APLICACIÓN (HELLO WORLD)

Hello-world.c

```
#include "contiki.h"
#include <stdio.h> /* For printf() */
/* ..... */
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/* ..... */
PROCESS_THREAD(hello_world_process, ev, data)
{
    static struct etimer timer;

    PROCESS_BEGIN();
    /* Setup a periodic timer that expires after 10 seconds. */
    etimer_set(&timer, CLOCK_SECOND * 10);

    while(1) {
        printf("Hello, world\n");

        /* Wait for the periodic timer to expire and then restart the timer. */
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer)); etimer_reset(&timer);
    }
    PROCESS_END();
}
```

II. PRIMERA APLICACIÓN (HELLO WORLD)

- Para compilar cualquier programa se necesitan 3 ficheros en la carpeta de la app:
 - **app.c (obligatorio)**: Fichero fuente con la aplicación desarrollada
 - **app.h (opcional)**: Fichero de cabecera con definiciones de vbles, ctes y funciones
 - **Makefile (obligatorio)**: Fichero de compilación de app
 - Permite compilar la aplicación desde el código fuente especificando una serie de reglas y rutas a los ficheros que necesita la aplicación
- Contenido mínimo del fichero Makefile:

```
CONTIKI = ../..
```

Especifica la localización del directorio raíz donde se localiza Contiki-NG

```
all: app-name
```

Indica qué aplicación ha de ser compilada

```
include $(CONTIKI)/Makefile.include
```

Incluye el Makefile.include

II. PRIMERA APLICACIÓN (HELLO WORLD)

Compilación y ejecución en modo nativo

Acceder al directorio de trabajo

```
cd ~/contiki-ng/examples/hello-world/
```

Realizar la compilación

```
make
```

Verificar los ficheros que se han generado

```
ls -la
```

Ejecutar la aplicación en la propia máquina

```
./hello-world.native
```

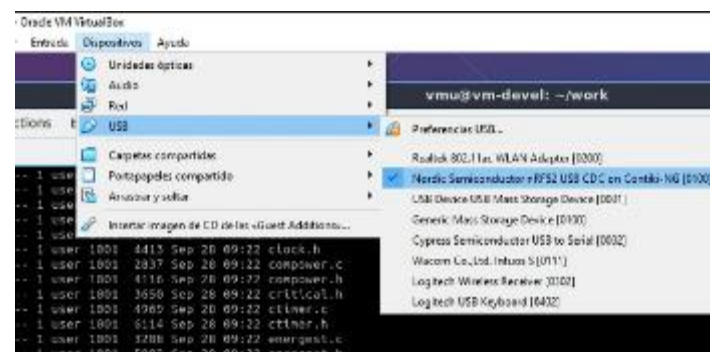
II. PRIMERA APLICACIÓN (HELLO WORLD)

Compilación y carga en nRF52840

Acceder al directorio de trabajo

```
cd ~/contiki-ng/examples/hello-world/
make TARGET=nrf52840 savetarget
make
```

Conectar la placa y activar el modo DFU



Cargar el archivo generado en el nodo físico y visualizar la salida estándar (USB)

```
make hello-world.dfu-upload
picocom -fh -b 115200 --imap lfcrLf /dev/ttyACM0
```

II. PRIMERA APLICACIÓN (HELLO WORLD)

- **Ejercicio 2: Hello World (1 punto)**

Realice el Test inicial: Hello World del repositorio de instalación (<https://gitlab.com/laboratorio-ris/contiki-ng>). Adjunte las capturas necesarias para mostrar la compilación, la carga del programa y la ejecución del mismo.

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN TEMPORIZADORES

- Existen varios tipos de temporizadores:
 - **Pasivo**
 - `struct timer`: a simple timer, without built-in notification (caller must check if expired).
 - **Activos**
 - `struct ctimer`: schedules calls to a callback function.
 - `struct etimer`: schedules events to Contiki-NG processes.
 - **Tiempo real**
 - `struct rtimer`: real-time task scheduling, with execution from ISR at exact time. Safe from interrupt.

Archivos fuente y de cabecera: `$HOME/contiki-ng/os/sys`

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN TEMPORIZADORES

Function	Purpose
<code>void etimer_set(struct etimer *t, clock_time_t interval)</code>	Start the timer.
<code>void etimer_reset(struct etimer*t)</code>	Restart the timer from the previous expire time.
<code>void etimer_restart(struct etimer*t)</code>	Restart the timer from current time.
<code>void etimer_stop(struct etimer*t)</code>	Stop the timer.
<code>int etimer_expired(struct etimer *t)</code>	Check if the timer has expired.
<code>int etimer_pending()</code>	Check if there are any non-expired event timers.
<code>clock_time_t etimer_next_expiration_time()</code>	Get the next event timer expiration time.
<code>void etimer_request_poll()</code>	Inform the etimer library that the system clock has changed.

Archivos fuente y de cabecera

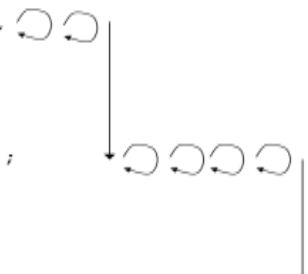
`$HOME/contiki-ng/os/sys/etimer.c` `$HOME/contiki-ng/os/sys/etimer.h`

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN

PROCESOS Y PROTOHILOS

- Un protohilo se define como una función de código

```
int a_protothread(struct pt *pt) {
    PT_BEGIN(pt);
    /* ... */
    PT_WAIT_UNTIL(pt, condition1);
    /* ... */
    if(something) {
        /* ... */
        PT_WAIT_UNTIL(pt, condition2);
        /* ... */
    }
    PT_END(pt);
}
```



- Un proceso en Contiki OS es un protohilo. Se crean a partir de macros.

```
~/contiki-ng/os/sys/process.c
~/contiki-ng/os/sys/process.h
```

PROCESS(name, strname) → Declaración de un proceso

PROCESS_THREAD(name, ev, data) → Definición de un proceso

¡¡Puede haber más de uno en la misma aplicación!!

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN

PROCESOS Y PROTOHILOS

```
#include "contiki.h"
```

```
#include <stdio.h> /* For printf() */
```

```
/*-----  
PROCESS(hello_world_process, "Hello world process");  
AUTOSTART_PROCESSES(&hello_world_process);  
/*-----
```

Declaración del proceso en la cabecera del programa

Listado de procesos que se cargan al arranque

```
PROCESS_THREAD(hello_world_process, ev, data)  
{
```

```
PROCESS_BEGIN();
```

Macro de inicialización (obligatoria)

```
while(1) {
```

```
PROCESS_WAIT_EVENT();
```



Código específico del proceso
Macro de espera de un evento

```
PROCESS_END();
```

Macro de finalización (obligatoria)

```
}
```

- **ev:** variable que recoge los eventos lanzados por otros procesos
- **data:** variable puntero en la que otros procesos pueden pasar datos al generar un evento

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN

PROCESOS Y PROTOHILOS

- **Gestión de eventos:**

- Normalmente se procesan con dos macros

```
PROCESS_WAIT_EVENT_UNTIL (etimer_expired(&et))  
PROCESS_WAIT_EVENT_UNTIL (ev == reading_event)
```

- **Generación de eventos:**

- **POST:**

- Asíncrono -> ***int process_post(process_ptr, evento, ptr);***
 - Se generará el evento en cuanto lo decida el kernel de Contiki
- Síncrono -> ***void process_post_synch(process_ptr, evento, ptr);***
 - Se genera el evento en este momento
 - No se debe utilizar en una rutina de interrupción

- **POLL:**

- ***Void process_poll(process_ptr);*** -> genera evento *PROCESS_EVENT_POLL*
- Se puede utilizar en una rutina de interrupción

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN

PROCESOS Y PROTOHILOS

- Ejemplo de uso. La función *example_function* envía distintos tipos de eventos hacia el proceso anterior

```
static char msg[] = "Data";

static void
example_function(void)
{
    /* Start "Example process", and send it a NULL
       pointer. */

    process_start(&example_process, NULL);

    /* Send the PROCESS_EVENT_MSG event synchronously to
       "Example process", with a pointer to the message in the
       array 'msg'. */
    process_post_synch(&example_process,
                      PROCESS_EVENT_CONTINUE, msg); ————— Evento POST síncrono

    /* Send the PROCESS_EVENT_MSG event asynchronously to
       "Example process", with a pointer to the message in the
       array 'msg'. */
    process_post(&example_process,
                PROCESS_EVENT_CONTINUE, msg); ————— Evento POST asíncrono

    /* Poll "Example process". */
    process_poll(&example_process); ————— Evento POLL
}
```

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN

PROCESOS Y PROTOHILOS

- Ejercicio 3: Hello World modificado (2 puntos)

Realice una aplicación “Hello World” modificada de forma que ahora existan dos procesos:

- El original `hello_world_process`, que sólo se limitará a esperar eventos para imprimir un mensaje con un contador que se incrementará en cada evento recibido. Este contador llegará hasta 20, y posteriormente se reiniciará.
- Un nuevo proceso que se llamará `periodic_process`. Se ejecutará desde el principio junto con el anterior. Generará un evento cada 5 segundos y será enviado hacia el proceso `hello_world_process`.

```
Hello World! (number 0)
Hello World! (number 1)
Hello World! (number 2)
...
Hello World! (number 19)
Hello World! (number 20)
Hello World! (number 0)
Hello World! (number 1)
...
```

III. ASPECTOS BÁSICOS DE PROGRAMACIÓN

PROCESOS Y PROTOHILOS

- Ejercicio 4: Parpadeo de leds (3 puntos)

Realice la aplicación “Blink” que conmute los dos leds de la placa cada 2 y 3 segundos, respectivamente. Para ello, se hará uso de 3 procesos/protohilos:

- ***parpadeo_1_process***: Se encarga de conmutar el estado del Led 1 cada 2 segundos. Este proceso permanecerá esperando un evento del proceso *timer_process* antes de realizar la primera conmutación.
- ***parpadeo_2_process***: Se encarga de conmutar el estado del Led 2 cada 3 segundos. Este proceso permanecerá esperando un evento del proceso *timer_process* antes de realizar la primera conmutación.
- ***timer_process***: Proceso que se ejecutará al inicio de la aplicación y tras esperar 5 segundos enviará un evento *process_poll* a *Parpadeo_1_process* y *Parpadeo_2_process*.

IV. INTERFACES DE COMUNICACIÓN: ADQUISICIÓN DE SENSORES

- Para realizar la adquisición de sensores en Contiki hacen falta 3 cosas

- Ficheros genéricos de control de sensores

~/contiki-ng/os/lib/sensors.c

~/contiki-ng/os/lib/sensors.h

```
struct sensors_sensor {
    char *      type;
    int         (* value)      (int type);
    int         (* configure) (int type, int value);
    int         (* status)     (int type);
};

while(1) {
    PROCESS_WAIT_EVENT();

    do {
        events = 0;
        for(i = 0; i < num_sensors; ++i) {
            if(sensors_flags[i] & FLAG_CHANGED) {
                if(process_post(PROCESS_BROADCAST, sensors_event, (void *)sensors[i]) == PROCESS_ERR_OK) {
                    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event);
                }
                sensors_flags[i] &= ~FLAG_CHANGED;
                events++;
            }
        }
    } while(events);
}
```

Cada sensor almacena su información en esta estructura

Lectura de todos los eventos de sensores

IV. INTERFACES DE COMUNICACIÓN: ADQUISICIÓN DE SENSORES

- Ficheros para gestionar (driver) cada sensor en particular. Normalmente se encuentran en el directorio de cada plataforma

`~/contiki-ng/arch/platform/nrf52840/common/`

- Declaración de la variable del sensor de temperatura

```
/*-----*/  
/**  
 * \brief Returns device temperature  
 * \param type ignored  
 * \return Device temperature in degrees Celsius  
 */  
static int  
value(int type)  
{  
    int32_t volatile temp;  
  
    NRF_TEMP->TASKS_START = 1;  
    /* nRF52832 datasheet: one temperature measurement takes typically 36 us */  
    RTIMER_BUSYWAIT_UNTIL(NRF_TEMP->EVENTS_DATARDY, RTIMER_SECOND * 72 / 1000000);  
    NRF_TEMP->EVENTS_DATARDY = 0;  
    temp = nrf_temp_read();  
    NRF_TEMP->TASKS_STOP = 1;  
  
    return temp;  
}
```

- Función de activación: `SENSORS_ACTIVATE(temperature_sensor);`
- Función de lectura: `int32_t temperature_sensor.value(0);`
- Función de desactivación: `SENSORS_DEACTIVATE(temperature_sensor);`

IV. INTERFACES DE COMUNICACIÓN: ADQUISICIÓN DE SENSORES

- Ejercicio 5: Medida de la temperatura interna (3 puntos)

Realice una aplicación denominada “Temperature” que tenga la siguiente funcionalidad:

- Lectura de la temperatura medida por el sensor interno del NRF52840 cada 3 segundos
- Imprima por pantalla el valor leído en grados centígrados.

NOTA: Utilice procesos independientes para la lectura del sensor y para generar la temporización de 3 segundos.