

# Nano-Kernel Bare-Metal para RP2040

## Informe Final

---

Matías Cajal

Universidad Nacional de Río Negro

Laboratorio de Sistemas Embebidos

20 de junio de 2025

## Índice

---

1. Hardware utilizado .....	3
2. Software utilizado .....	3
3. Esquema de conexiones .....	4
II. Desarrollo .....	4
1. Instalación de software .....	4
2. Lenguaje de programación .....	5
3. Estructura de archivos .....	5
4. Bootloader y arranque del sistema .....	6
a. Boot2 (Segunda etapa de bootloader): .....	6
b. Vector Table: .....	6
5. Hardware Abstraction Layer (HAL) .....	7
a. Clocks: .....	7
b. Reset: .....	7
c. XOSC (External Oscillator): .....	7
d. PLL (Phase-Locked Loop): .....	7
e. IO Bank y Pad Bank: .....	7
f. SIO (Single-cycle I/O): .....	7
g. UART: .....	7
6. Scheduler .....	7
7. Cambio de contexto .....	8
8. Shell interactiva .....	8
Help .....	8
Peek .....	9
Poke .....	9
GPIO Set .....	9
Show Tasks .....	9
9. Gestión de memoria .....	9
10. Visualización y monitoreo de resultados .....	9
11. Código fuente .....	10

**Estado actual del desarrollo:**

El proyecto no fue finalizado debido a problemas encontrados durante el desarrollo.

**1. Hardware utilizado**

El proyecto contempla el desarrollo de un nano-kernel bare-metal para el microcontrolador RP2040, montado en una placa Raspberry Pi Pico. El RP2040 es un microcontrolador dual-core ARM Cortex-M0+ desarrollado por Raspberry Pi Foundation.

**Características del RP2040:**

- **CPU:** Dual ARM Cortex-M0+ hasta 133MHz
- **Memoria:** 264KB SRAM on-chip
- **Flash:** Hasta 16MB external QSPI flash
- **GPIO:** 30 pines GPIO multifunción
- **Periféricos:** 2× UART, 2× SPI, 2× I<sup>2</sup>C, 16× PWM
- **Timer:** 1× timer de 64-bit

La placa Raspberry Pi Pico proporciona una plataforma de desarrollo compacta con microcontrolador RP2040, 2MB de memoria flash QSPI, LED onboard conectado al GPIO 25, pines de desarrollo accesibles y conector micro-USB para programación y alimentación.

**Componentes adicionales utilizados:**

- Adaptador USB-Serie TTL (CP2102 3.3V) para comunicación UART
- Cables de conexión (jumper wires)
- Protoboard para conexiones adicionales (opcional)

**2. Software utilizado**

Para realizar el proyecto se utilizan diferentes herramientas de software gratuitas:

**Toolchain de desarrollo:**

- **GNU ARM Embedded Toolchain** (arm-none-eabi-gcc): Compilador cruzado para arquitectura ARM
- **GNU Assembler** (arm-none-eabi-as): Ensamblador para código ARM Thumb
- **GNU Linker** (arm-none-eabi-ld): Enlazador para generar ejecutables

**Herramientas de programación:**

- **elf2uf2-rs:** Para generar un archivo que permita flashear la Raspberry Pi Pico.

**Entorno de desarrollo:**

- **Visual Studio Code:** Editor de código con extensiones para C/C++

**Software de terminal:**

- **minicom:** Emulador de terminal serie para Linux

**3. Esquema de conexiones**

Las conexiones del proyecto son mínimas, utilizando principalmente la comunicación UART para la shell interactiva y el LED onboard para indicación visual.

**Tabla de conexiones principales:**

Componente	Pin RP2040	Función
LED onboard	GPIO 25	Indicador visual
UART0 TX	GPIO 0	Transmisión serie
UART0 RX	GPIO 1	Recepción serie
USB-Serie	-	Comunicación con PC

**II. Desarrollo****1. Instalación de software**

La instalación del entorno de desarrollo se realiza en un sistema Linux Ubuntu 24.04 LTS.

**Instalación del toolchain GNU ARM Embedded:** Se instala el toolchain ARM mediante el gestor de paquetes de Ubuntu:

```
sudo apt update
sudo apt install gcc-arm-none-eabi binutils-arm-none-eabi
```

**Instalación de Visual Studio Code:** Se instala VSCode mediante snap:

```
sudo snap install code --classic
```

**Verificación de la instalación:** Se verifica la instalación exitosa mediante:

```
arm-none-eabi-gcc --version
arm-none-eabi-as --version
```

## 2. Lenguaje de programación

El proyecto se desarrolla con lenguaje de programación C con un paradigma de programación imperativo. La elección se basa en la necesidad de control directo sobre el hardware sin abstracciones.

### C (freestanding):

- Lógica principal del nano-kernel
- Drivers de periféricos
- Gestión de tareas y scheduler
- Sin biblioteca estándar C (libc)

### Assembler ARM Thumb:

- Código de arranque (startup.S)
- Rutinas de cambio de contexto
- Manejo de interrupciones críticas
- Operaciones de bajo nivel

La elección de C freestanding permite control total sobre el hardware sin dependencias externas, mientras que Assembler proporciona el control preciso necesario para operaciones críticas del sistema.

## 3. Estructura de archivos

La estructura del proyecto se organiza de manera modular para facilitar el desarrollo y mantenimiento del nano-kernel bare-metal:

```
pico-kernel/  
├─ CMakeLists.txt      # Sistema de compilación CMake  
├─ memory.x            # Definición del layout de memoria  
├─ README.md           # Documentación del proyecto  
└─ src/  
    ├─ boot2/  
    │   ├─ boot2.S      # Segunda etapa de bootloader  
    │   └─ ram_cpy.S    # Rutinas de copia a RAM  
    ├─ hal/              # Hardware Abstraction Layer  
    └─ clocks/           # Gestión de relojes del sistema
```

```
|   └─ common/           # Funciones comunes del HAL
|   └─ io_bank/          # Configuración GPIO IO Bank
|   └─ pad_bank/         # Configuración de pads
|   └─ pll/              # Phase-Locked Loop
|   └─ reset/            # Gestión de reset de periféricos
|   └─ sio/              # Single-cycle I/O
|   └─ uart/             # Driver UART
|   └─ xosc/             # Oscillador externo
└─ main.c                # Función principal del programa
└─ rp2040.h              # Definiciones de hardware RP2040
└─ vector-table/
    └─ vector_table.S    # Tabla de vectores de interrupción
```

Esta organización modular permite separación clara de responsabilidades entre bootloader, HAL y aplicación.

## 4. Bootloader y arranque del sistema

### a. Boot2 (Segunda etapa de bootloader):

El RP2040 utiliza un bootloader en dos etapas. La primera etapa (Boot ROM) está integrada en el chip, y la segunda etapa (Boot2) debe implementarse en los primeros 256 bytes del flash.

**Boot2.S** contiene la configuración inicial del flash QSPI, inicialización de memoria y el salto a la aplicación principal.

**ram\_cpy.S** implementa las rutinas de copia fundamentales para inicializar las secciones .data desde flash hacia RAM, optimizadas para ARM Cortex-M0+.

### b. Vector Table:

La tabla de vectores de interrupción se implementa como un archivo separado para mayor claridad y mantenibilidad.

**vector\_table.S** contiene la tabla completa de vectores de interrupción del RP2040, incluyendo stack pointer inicial y Reset handler, excepciones del procesador (NMI, HardFault, SVC, PendSV, SysTick) e interrupciones específicas del RP2040 (Timer, PWM, USB, PIO, DMA, IO Banks, SPI, UART, ADC, I2C, RTC).

## 5. Hardware Abstraction Layer (HAL)

El HAL proporciona una capa de abstracción entre el hardware específico del RP2040 y las funciones de alto nivel del nano-kernel. Cada módulo del HAL gestiona un periférico específico.

### a. Clocks:

El módulo de clocks gestiona la configuración de todos los relojes del sistema, incluyendo PLLs y divisores de frecuencia.

### Características:

- Configuración de PLLs (Phase-Locked Loops)
- Gestión de divisores de clock
- Selección de fuentes de reloj
- Configuración de frecuencias para periféricos

### b. Reset:

El módulo de reset controla el reset individual de periféricos, permitiendo reinicializar componentes específicos sin afectar el resto del sistema.

### c. XOSC (External Oscillator):

Gestiona el oscilador externo de 12MHz que proporciona la referencia de tiempo base para el sistema.

### d. PLL (Phase-Locked Loop):

Los PLLs generan las frecuencias de operación del sistema a partir del oscilador de referencia.

### e. IO Bank y Pad Bank:

Configuran las funciones y características eléctricas de los pines GPIO.

**IO Bank:** Configuración de las funciones alternativas de cada pin GPIO.

**Pad Bank:** Control de resistencias pull-up/pull-down, drive strength y slew rate.

### f. SIO (Single-cycle I/O):

Proporciona acceso directo y rápido a los pines GPIO para operaciones que requieren latencia mínima.

### g. UART:

Driver para comunicación serie con soporte para interrupciones y FIFO.

## 6. Scheduler

El scheduler implementa un algoritmo de planificación pre-emptive Round-Robin.

**Estructura TCB (Task Control Block):** Cada tarea del sistema se representa mediante una estructura que contiene:

- Puntero al stack de la tarea
- Base y tamaño del stack asignado
- Estado actual de la tarea (READY, RUNNING, BLOCKED, TERMINATED)
- Prioridad para futuras mejoras
- Nombre identificativo de la tarea
- Contadores de tiempo de ejecución

**Algoritmo del scheduler:** El scheduler utiliza un algoritmo Round-Robin que selecciona cíclicamente la siguiente tarea en estado READY para ejecutar, garantizando una distribución equitativa del tiempo de CPU entre todas las tareas activas.

## 7. Cambio de contexto

El cambio de contexto se implementa en Assembler para máxima eficiencia y control preciso sobre los registros del procesador.

**Rutina de cambio de contexto:** La implementación en Assembler ARM Thumb realiza las siguientes operaciones:

- Guarda el contexto actual (registros R4-R11, LR)
- Cambia el stack pointer hacia el de la nueva tarea
- Restaura el contexto de la nueva tarea
- Retorna a la ejecución de la nueva tarea

Esta rutina es crítica para el funcionamiento del scheduler pre-emptive y debe ejecutarse de manera atómica para evitar corrupción del estado del sistema.

## 8. Shell interactiva

La shell interactiva proporciona una interfaz de línea de comandos a través del puerto serie UART0.

**Comandos especificados:**

### Help

**Formato:** help **Descripción:** Lista todos los comandos disponibles.



**Peek**

**Formato:** peek <addr\_hex> **Descripción:** Lee e imprime 4 bytes de memoria en la dirección especificada. **Ejemplo:** peek 0x20000000

**Poke**

**Formato:** poke <addr\_hex> <val\_hex> **Descripción:** Escribe 4 bytes en la dirección de memoria especificada. **Ejemplo:** poke 0x20000000 0xDEADBEEF

**GPIO Set**

**Formato:** gpio set <pin> <0|1> **Descripción:** Configura un pin GPIO en estado bajo (0) o alto (1). **Ejemplo:** gpio set 25 1 (enciende LED onboard)

**Show Tasks**

**Formato:** show-tasks **Descripción:** Muestra el estado de todas las tareas activas, incluyendo nombre, estado y uso de stack.

## 9. Gestión de memoria

El archivo `memory.x` define el layout de memoria específico para el RP2040.

**Configuración de memoria:** El layout de memoria está dividido en tres secciones principales:

- **BOOT2:** Segunda etapa del bootloader (256 bytes en 0x10000000)
- **FLASH:** Código del programa y datos constantes (desde 0x10000100)
- **RAM:** Variables, stacks y heap dinámico (256KB desde 0x20000000)

**Organización de la RAM:** La memoria RAM se organiza secuencialmente con variables globales, stacks de tareas individuales, heap dinámico opcional y el stack principal ubicado al final de la memoria RAM.

## 10. Visualización y monitoreo de resultados

La visualización de datos se realiza a través del terminal serie mediante la shell interactiva implementada.

**Estado actual del desarrollo:**

El proyecto no fue finalizado debido a problemas encontrados durante el desarrollo.

**Funcionalidades que se alcanzaron a implementar:**

- Bootloader personalizado (Boot2) funcional
- HAL modular implementado
- Gestión de memoria configurada

- Funcionalidad Blinky operativa

**Problemas identificados:** Hubo problemas con el driver durante el desarrollo que impidieron completar el proyecto.

**Solución:** Releer la documentación del RP2040, específicamente de la UART y proponer un nuevo driver.

### **11. Código fuente**

El código fuente del proyecto está compartido en github, en el siguiente enlace: <https://github.com/claaj/pico-kernel>