Práctica 8:
Linux embebido

Fundamentos de Sistemas Embebidos

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno aprenderá a generar una imagen de Linux optimizada para la Raspberry Pi usando *buildroot*.

2. Introducción

Existen múltiples herramientas para desarrollar sistemas embebidos, tales como Buildroot [1, 2], Yocto Project [2, 3] y Linaro Project [2], entre otros. De estos se ha elegido Buildroot por ser minimalista y sencillo.

De acuerdo con el sitio oficial de Buildroot:

> Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation. (The Buildroot developers [1])

Buildroot es una herramienta que simplifica y automatiza el proceso de creación de un sistema Linux embebido mediante compilación cruzada [1]. Esto se logra mediante un conjunto de interfaces unificadoras que operan sobre un conjunto de parches y archivos `Makefile` para cmake que generan y enlazan binarios para el kernel y los paquetes seleccionados, incluyendo la estructura del sistema de archivos raíz y la generación de una imagen de sistema operativo [1, 2].

En esta práctica se resumen los pasos para crear un sistema embebido mínimo y sumamente simple.

3. Material

Se aconseja encarecidamente el uso de *git* como programa de control de versiones.

- 1 PC o laptop con:
  i) sistema operativo POSIX (preferentemente Linux basado en Debian)
  ii) conexión a internet
  iii) 20GiB de espacio en disco disponible
- 1 Raspberry Pi
- 1 Memoria microSD de 1GB o más
- Alambrado de la práctica 7: «Desplegado de temperatura en un display digital»
- 1 fuente de alimentación regulada a 5V y al menos 2 amperios de salida
- Cables y conectores varios

Nota: Para esta práctica requerirá los programas desarrollados para la práctica *desplegado de temperatura en un display digital usando la Raspberry Pi*.

[0] En lugar del sensor DS18B20 es posible usar el alambrado de arduino con LM35 de la práctica 6.

4. Instrucciones

1. Configure su entorno de desarrollo tomando siguiendo los pasos de la Subsección 4.1.
2. Descargue y configure buildroot como se describe en las Subsecciones 4.2 y 4.3.
3. Siguiendo los pasos de la Subsección 4.4 modifique el sistema de archivos de la distro embebida.
4. Compile el kernel y genere la imagen de sistema tal como explica la Subsección 4.5
5. Grabe la imagen generada en la memoria microSD y pruebe el sistema operativo embebido en la Raspberry Pi de acuerdo con las instrucciones de la Subsección 4.6
6. Por último, realice los experimentos propuestos en la sección 5.

4.1. Paso 1: Configuración del entorno de desarrollo

Sistema base: Debian

Los pasos de esta sección suponen que la máquina de desarrollo o anfitrión cuenta con un sistema operativo Debian o compatible instalado (ubuntu, mint, etc...).
Si usted cuenta con otra distribución como Fedora, Arch, o incluso iOS, es posible realizar la práctica pero tendrá que utilizar los comandos apropiados para su sistema operativo.

Nota: No es posible llevar a cabo la práctica en Windows.

La configuración del entorno de desarrollo es relativamente sencilla. Para esto basta con instalar los paquetes de compilación cruzada para ARM, así como los paquetes adicionales requeridos por buildroot como `flex`, `bison` o `ncurses`.
Todo el procedimiento se reduce a una sóla línea:

```
# apt install -y git bc bison flex libssl-dev make libc6-dev libncurses5-dev build-essential
crossbuild-essential-arm64 crossbuild-essential-armhf
```

A continuación, cree un nuevo directorio en su *home* y sitúese allí; por ejemplo:

```
$ mkdir ~/myPiLinux
$ cd ~/myPiLinux
```

4.2. Paso 2: Configuración con buildroot

Para generar un kernel optimizado se utilizará *buildroot*. Buildroot permite no sólo configurar y compilar un kernel Linux personalizado, sino la imagen del sistema completo incluyendo el sistema de archivos con los programas embebidos y cualesquiera paquetes necesarios para ejecutarlos.

Paso 2.1. Descarga y descompresión

El primer paso consiste en descargar buildroot de `https://buildroot.org/download.html`, por ejemplo mediante la línea:

```
$ wget https://buildroot.org/downloads/buildroot-2025.02.tar.xz
```

A continuación descomprima buildroot en la carpeta de su proyecto, por ejemplo con:

```
$ tar xf buildroot-2025.02.tar.xz
```

e ingrese al directorio, por ejemplo `cd buildroot-2025.02`.

El siguiente paso consiste en verificar que tenemos todos los paquetes necesarios para el proyecto dentro de *buildroot*. Verifique que `python-smbus2` está disponible ejecutando el comando `ls package/python-* | grep smbus2`. Debería ver una salida como la siguiente:

```
$ ls package/python-* | grep smbus2
package/python-smbus2:
python-smbus2.hash
python-smbus2.mk
```

Si estos archivos no estuvieren disponibles, será necesario crearlos. El prodecimiento para crear dichos archivos se detalla en el Apéndice B.

Paso 2.2. Selección de arquitectura objetivo

buildroot permite la creación de imágenes de Linux para prácticamente cualquier arquitectura soportada. La lista de configuraciones incluidas se puede obtener con el comando `make list-defconfigs` cuya salida puede ser un poco larga, por lo que la recortaremos con `grep`, obteniendo una salida como la siguiente:

```
$ make list-defconfigs | grep raspberry
    raspberrypi0_defconfig              - Build for raspberrypi0
    raspberrypi0w_defconfig             - Build for raspberrypi0w
    raspberrypi2_defconfig              - Build for raspberrypi2
    raspberrypi3_64_defconfig           - Build for raspberrypi3_64
    raspberrypi3_defconfig              - Build for raspberrypi3
    raspberrypi3_qt5we_defconfig        - Build for raspberrypi3_qt5we
    raspberrypi4_64_defconfig           - Build for raspberrypi4_64
    raspberrypi4_defconfig              - Build for raspberrypi4
    raspberrypicm4io_64_defconfig       - Build for raspberrypicm4io_64
    raspberrypicm4io_defconfig          - Build for raspberrypicm4io
    raspberrypi_defconfig               - Build for raspberrypi
    raspberrypizero2w_defconfig         - Build for raspberrypizero2w
```

El siguiente paso consiste en seleccionar la configuración base para la arquitectura a generar. En nuestro caso tomaremos como base una Raspberry Pi 4 con sistema de 64 bits.

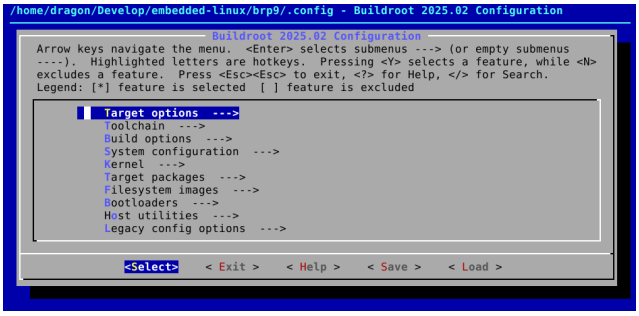
```
$ make raspberrypi4_64_defconfig
```

Paso 2.2. Configuración del sistema

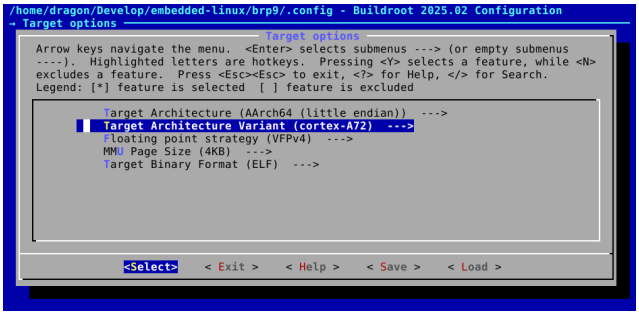
Invoque interfaz de buildroot con el comando

```
$ make menuconfig
```

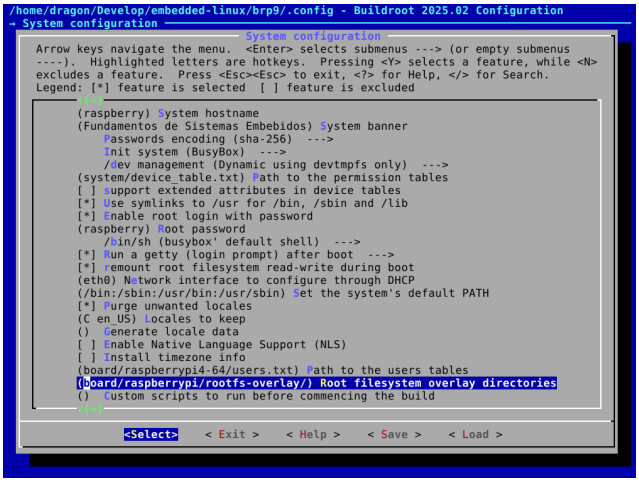
verá una pantalla similar a la de la Figura 1a.



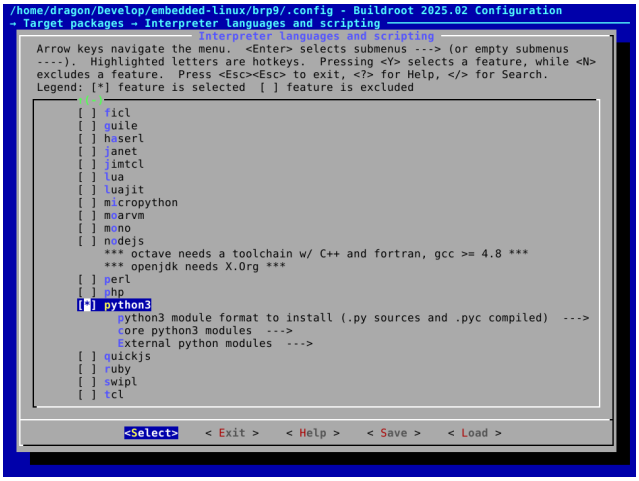
(a) Menú principal



(b) Arquitectura objetivo (menú *target options*)



(c) Configuración del sistema (menú *system configuration*)



(d) Paquete `python3` habilitado (menú *target packages*)

Figura 1: Interfaz gráfica de buildroot (modo consola)

A continuación, tómese unos minutos para familiarizarse con el entorno gráfico. En particular revise el `toolchain` y el `target`. En este último notará que la arquitectura es `AArch64` para el cortex-A72 tal como se muestra en la Figura 1b.

Bajo *Build Options* se cambiarán tres valores. El primero es establecer *Location to save buildroot config* con el valor `./.config` para evitar sobreescribir la configuración predeterminada. El segundo es *global patch and hash directories* con el valor `board/raspberrypi4-64/patches`; directorio en el cual se copiarán los parches que se aplicarán al Kernel previo a la compilación del mismo y que será usado en el siguiente paso (véase Subsección 4.3). El tercer y último valor es habilitar la compilación de librerías para que éstas estén disponibles tanto en su versión dinámica como estática; cambiando *libraries* a `both`.

En el menú de configuración del sistema (*System Configuration*) se definirán siete valores: a) el nombre del sistema (*system hostname*) se establece a raspberry, aunque se recomienda que los estudiantes utilicen el nombre de la brigada o equipo; b) el saludo inicial o estandarte(*system banner*) que se establece al nombre de la asignatura, c) la habilitación de los directorios `/bin`, `/sbin` y `/lib` como enlaces simbólicos dentro de `/usr`; d) la habilitación del usuario de root para iniciar sesión (*enable root login*) y, por consiguiente, e) establecer la contraseña de root (*root password*) a `raspberry`. f) Por último, se establece el path de la tabla de usuarios (*path to the users tables*) y g) el directorio que contendrá el sistema de archivos raíz (*root filesystem overlay directories*). Se presenta un resúmen de estos valores en la Tabla 1.

Advertencia de seguridad

Habilitar el inicio de sesión para el súper usuario *root* es una tremenda falla de seguridad ya que permite a los usuarios malintencionados tomar control del sistema. Se habilita en esta práctica para permitir al estudiante depurar su código *in situ*.
Los sistemas en producción deberán deshabilitar esta opción.

Paso 2.3. Selección de paquetes preinstalados

El siguiente paso es crítico, ya que es aquí donde se habilitará el intérprete de python con todas sus librerías, así como cualquier programa necesario. Dentro del menú de paquetes objetivo (*target packages*) ubique el sub-menú de lenguajes interpretados y de scripting (*interpreter languages and scripting*) Dentro de éste localice `python3` y habilítelo.

De manera automática se actualizará el menú mostrando nuevas opciones bajo la recién habilitada `python3`. El primer cambio importante consiste en cambiar el módulo de soporte de formatos a instalar (*python3 module format to install*). De forma predeterminada buildroot instala soporte sólo para archivos de python precompilados `pyc`; un tipo de archivo que contiene código en bytes similar a los formatos de Java y C# que pueden ser interpretados por Python sin exponer el código fuente del mismo. Sin embargo, estos archivos tienen el inconveniente de que deben ser generados para una versión específica del intérprete. Por este motivo es conveniente para el estudiante habilitar el soporte de archivos de scripting `.py`.

Los siguientes dos menús contienen la lista de módulos de python disponibles para embebir. Nuestro sistema embebido no estará configurado para conectarse a internet ni incluye un gestor de paquetes, por lo que no se podrá usar ni `apt` ni `pip3` para instalar paquetes. Estos tendrán que venir ya incluidos y compilados para nuestros sistema.

Navegue dentro del menú de modulos externos de Python (*external python modules*) y habilite los paquetes `python-pigpio` y `python-smbus2` que agregó previamente al inicio de esta sección.

De forma similar a como habilitó `python3`, ubique el submenú de componentes de hardware (Hardware handling) dentro del menú de paquetes objetivo (*target packages*) y habilite las herramientas para I^2C que aparecen como `i2c-tools`.

Paso 2.4. Configuraciones finales de la imagen

El siguiente paso consiste en reservar suficiente espacio en la imagen para el proyecto. Navegue al menú imágenes del sistema de archivos (*filesystem images*) localizado en la pantalla principal y cambie el el tamaño exacto de la imagen (*exact size*) a `250M` como se muestra en la Tabla 1.

Finalmente, tras cerrar `menuconfig`, es necesario habilitar en el kernel la carga de los controladores I^2C. Para este propósito, edite el archivo `config.txt` de su modelo de raspbery en el directorio correspondiente. En nuestro caso (una Raspberry Pi 4 con sistema operativo de 64 bits) es el archivo `board/raspberrypi4-64/config_4_64bit.txt`. Agregue las siguientes líneas al final:

```
# Enable kernel drivers for i2c1
dtparam=i2c1=on
```

O bien, puede simplemente ejecutar esta línea:

```
$ echo -e "# enable i2c\ndtparam=i2c1=on" >> board/raspberrypi4-64/config_4_64bit.txt
```

De forma similar, es necesario agregar un usuario con privilegios limitados (por seguridad no queremos que los programas se ejecuten con privilegios elevados). Esto se logra proporcionando a *buildroot* un archivo con la tabla de usuarios con sus grupos, directorios y permisos. Para este fin crearemos el archivo `board/raspberrypi4-64/users.txt` con el siguiente texto:

board/raspberrypi4-64/users.txt

```
1 # username uid group gid password home shell groups comments
2 pi  -1  pi  -1  =raspberry  /home/pi  /bin/sh i2c,dialout
```

Tabla 1: Lista de cambios a realizar en `make menuconfig`

Submenú	Valor
Build Options → Location to save buildroot config	`./.config`
Build Options → libraries	`both`
Build Options → global patch and hash directories	`board/raspberrypi4-64/patches`
System Configuration → System hostname	`raspberry`
System Configuration → System banner	`Fundamentos de Sistemas Embebidos`
System Configuration → Use symlinks...	`y`
System Configuration → Enable root login	`y`
System Configuration → Root password	`raspberry`
System Configuration → Path to the users tables	`board/raspberrypi4-64/users.txt`
System Configuration → RootFS overlay directories	`board/raspberrypi4-64/rootfs-overlay/`
Kernel → Custom boot logo file path	`board/raspberrypi4-64/logo.png`
Target packages → Interpreter languages and scripting →	
python3	`y`
python3 → python3 module format to install	`.py sources and .pyc compiled`
python3 → External python modules →	
`python-pigpio`	`y`
`python-smbus2`	`y`
Target packages → Hardware handling →	
i2c-tools	`y`
Filesystem images → exact size	`250M`

4.3. Paso 3: Parche de kernel y logotipo de arranque (Opcional)

En este paso daremos una apariencia más profesional al sistema embebido.

Debido a que modificar el kernel es un trabajo difícil y propenso a errores, *buildroot* nos facilita las cosas al permitirnos mantener los archivos originales del kernel íntegros y modificarlos al vuelo mediante la aplicación de parches que se aplican previo a la compilación del mismo.

Antes de compilar el kernel y los paquetes solicitados, *buildroot* tomará el directorio especificado y aplicará los archivos `.patch` a los archivos especificados en los mismos utilizando la herramienta `diff`. El directorio de superposición puede ser cualquiera en el disco duro. Sin embargo, se estila colocarlo como un subdirectorio de nombre `patches` dentro de `board/<company>/<boardname>`.
En nuestro caso la ruta relativa será:

```
board/raspberrypi4-64/patches
```

Procederemos a crear el directorio donde almacenaremos el parche que modificará el código de arranque para mostrar nuestro logotipo de inicio.

```
$ mkdir -p $board/raspberrypi4-64/patches/linux
```

A coninuación procedemos a crear el archivo de parche `logo.path` con un editor de texto plano como `nano` o `vim`, ingresando el siguiente contenido:

patches/linux/000-center-logo.patch

```
--- linux-custom.old/drivers/video/fbdev/core/fbcon.c 2024-04-18 04:08:18.000000000 -0600
+++ linux-custom/drivers/video/fbdev/core/fbcon.c 2025-04-10 19:09:10.786595689 -0600
@@ -569,6 +569,7 @@ static void fbcon_prepare_logo(struct vc
  if (fb_get_color_depth(&info->var, &info->fix) == 1)
    erase &= ~0x400;
  logo_height = fb_prepare_logo(info, ops->rotate);
+ logo_height+= (info->var.yres/2)-(logo_height/2);
  logo_lines = DIV_ROUND_UP(logo_height, vc->vc_font.height);
  q = (unsigned short *) (vc->vc_origin +
       vc->vc_size_row * rows);
@@ -1004,8 +1005,8 @@ static void fbcon_init(struct vc_data *v

  info = fbcon_info_from_console(vc->vc_num);

- if (logo_shown < 0 && console_loglevel <= CONSOLE_LOGLEVEL_QUIET)
-   logo_shown = FBCON_LOGO_DONTSHOW;
+// if (logo_shown < 0 && console_loglevel <= CONSOLE_LOGLEVEL_QUIET)
+//   logo_shown = FBCON_LOGO_DONTSHOW;

  if (vc != svc || logo_shown == FBCON_LOGO_DONTSHOW ||
      (info->fix.type == FB_TYPE_TEXT))



--- linux-custom.old/drivers/video/fbdev/core/fbmem.c 2024-04-18 04:08:18.000000000 -0600
+++ linux-custom/drivers/video/fbdev/core/fbmem.c 2025-04-10 19:15:32.124961062 -0600
@@ -56,7 +56,7 @@ int min_dynamic_fb __read_mostly;

 bool fb_center_logo __read_mostly;

-int fb_logo_count __read_mostly = -1;
+int fb_logo_count __read_mostly = 1;

 struct fb_info *get_fb_info(unsigned int idx)
 {
@@ -499,6 +499,7 @@ static int fb_show_logo_line(struct fb_i
    fb_set_logo(info, logo, logo_new, fb_logo.depth);
  }

+ fb_center_logo = true;
  if (fb_center_logo) {
    int xres = info->var.xres;
    int yres = info->var.yres;
```

Este archivo de parche `.patch` modifica dos archivos que controlan el desplegado de información en el *framebuffer* o manejador gráfico básico del kernel. El primero es `fbcon.c`, el controlador del *framebuffer* para el modo consola, donde se modifica la posición del logo para que aparezca en el centro de la pantalla; además de forzar que el logo siempre aparezca. El segundo es `fbmem.c` que tiene las rutinas de inicialización del *framebuffer* donde establecemos el número de logotipos de -1 o automático (igual al número de núcleos disponibles) a exactamente uno.

El siguiente paso consiste en indicarle a *buildroot* la ruta donde se encuentran los parches a aplicar. Navegue desde el menú principal a opciones de construcción (*build options*) y localice la entrada de directorios globales de parches y hashes (global patch and hash directories) e ingrese el siguiente valor:

```
board/raspberrypi4-64/patches
```

A continuación proveeremos la imagen a utilizar durante la carga del sistema operativo. Copie la imagen adjunta `logo.png` al directorio `board/raspberrypi4-64/` y después indique a *buildroot* la ruta donde se encuentran dicha imagen. Esto se hace definiendo ruta del logo de arranque personalizado (*custom boot logo file path*) dentro del menú Kernel. Se establece el valor:

```
board/raspberrypi4-64/logo.png
```

Por último, se deshabilitará el volcado de mensajes a la terminal para i) que esta no interfiera con el desplegado de la imagen de arranque y ii) acelerar la carga del sistema operativo.

Edite el archivo `cmdline.txt` en el directorio correspondiente. En nuestro caso (una Raspberry Pi 4 con sistema operativo de 64 bits) es el archivo `board/raspberrypi4-64/cmdline.txt`. Agregue el siguiente texto al final de la primera línea:

```
quiet loglevel=3
```

O bien, puede simplemente ejecutar esta línea:

```
$ sed -i '$s/$/ quiet loglevel=3/' board/raspberrypi4-64/cmdline.txt
```

Un inconveniente asociado a las modificaciones realizadas es que, una vez posicionado el logotipo, no nos permitirá mostrar información sino debajo del mismo, reduciendo el número de líneas que se muestran en pantalla hasta que se invoca el comando `clear`. Una forma de solucionar este incoveniente es automatizar la llamada del comando `clear` cada vez que se inicie sesión.

4.4. Paso 4: Configuración del *root filesystem*

El siguiente paso consiste en configurar el sistema de archivos raíz o *root filesystem*. Tras compilar el kernel y los paquetes solicitados, *buildroot* tomará el directorio especificado y sobrepondrá el contenido de éste al sistema de archivos raíz que está siendo generado, un proceso conocido como superposición u *overlay*.

El directorio de superposición puede ser cualquiera en el disco duro. Sin embargo, se estila colocarlo como un subdirectorio de nombre `rootfs-overlay` dentro de `board/<company>/<boardname>` que, si inspecciona con `ls`, notará que contiene ya varios archivos de configuración para nuestra tarjeta Raspberry Pi 4 de 64bits. En nuestro caso la ruta relativa será:

```
board/raspberrypi4-64/rootfs-overlay
```

Es en este directorio donde colocaremos todos los archivos y directorios que querramos contenga la imagen generada.

Ejecute los siguientes comando para crear el directorio *home* del usuario *pi* creado en el paso anterior y otros directorios que serán necesarios para configuar al sistema embebido.

```
mkdir -p board/raspberrypi4-64/rootfs-overlay/home/pi
mkdir -p board/raspberrypi4-64/rootfs-overlay/etc/init.d
```

El sistema creado por *buildroot* es un sistema mínimo. Esto quiere decir que no se incluye nada que no sea indispensable a menos que sea declarado de forma explícita, y esto se logra añadiendo los archivos de configuración pertinente al sistema de archivos raíz o *root filesystem* (en adelante rootFS). Por ejemplo, los controladores para I^2C no se cargan de forma predeterminada.

Para especificar el rootFS en *buildroot* ejecute `make menuconfig` y localice la opción *root filesystem overlay directories* bajo el menú de configuración del sistema (*system configuration*) con el valor del directorio creado anteriormente, es decir:

```
board/raspberrypi4-64/rootfs-overlay
```

Para indicarle al kernel que debe cargar los controladores para I^{2}C haremos uso de dos archivos. El primero será el archivo `S02modules` localizado en `etc/init.d`, directorio que contiene todos los scripts de arranque del sistema y que son ejecutados en orden.[1] Procedemos a crear el archivo, marcarlo como ejecutable y editarlo con los siguientes dos comandos:

```
touch board/raspberrypi4-64/rootfs-overlay/etc/init.d/S02modules
chmod +x board/raspberrypi4-64/rootfs-overlay/etc/init.d/S02modules
nano board/raspberrypi4-64/rootfs-overlay/etc/init.d/S02modules
```

Advertencia

Los archivos que está a punto de modificar son relativos al directorio de la tarjeta dentro de buildroot.

Por ningún motivo modifique los archivos en el directorio `/etc` de su sistema anfitrión.

A coninuación, agregue el siguiente contenido al archivo:

rootfs-overlay/etc/init.d/S02modules

```
#!/bin/sh
case "${1}" in
    start)
        # Exits if /etc/modules does not exists or lacks valid entries
        [ -r /etc/modules ] && egrep -qv '^($|#)' /etc/modules || exit 0
        # Load listed modules
        while read module args; do
            # Skip comments and empty lines.
            case "$module" in
                ""|"#"*) continue ;;
            esac
            # Try to load each module
            printf "Loading ${module}"
            modprobe ${module} ${args} > /dev/null
            [ $? = 0 ] && echo "OK" || echo "FAIL"
        done < /etc/modules
        exit 0
        ;;

    *)
        echo "Usage: ${0} {start}"
        exit 1
        ;;
esac
```

`S02modules` es un script de carga de controladores (módulos) simple y genérico que cargará de forma dinámica todos los controladores listados en otro archivo llamado `modules` en el directorio `/etc`, si existe. Procedamos pues a crear dicho archivo con un editor de texto como `nano`.

```
nano board/raspberrypi4-64/rootfs-overlay/etc/modules
```

A coninuación, agregue el siguiente contenido al archivo:

rootfs-overlay/etc/modules

```
# List of modules to be loaded during startup
i2c-bcm2835
i2c-dev
```

Estos archivos harán que el kernel cargue los controladores para I^{2}C al arranque, ¡pero `/dev/i2c-1` sólo será accesible por el superusuario *root*! Para solucionar este inconveniente crearemos otro script de arranque similar a

[1] Los controladores tienen muy alta prioridad, de allí que se tengan que cargar primero que otros servicios. Es por esto que el nombre del archivo comienza con S02, y será ejecutado después de todos los S00 y S01.

`S02modules` que cambie los permisos de todos los periféricos I^2C cambiando su grupo a i2c al cual el usuario *pi* que creamos en el paso anterior ya tiene acceso.

Procedemos a crear el archivo, marcarlo como ejecutable y editarlo con los siguientes dos comandos:

```
touch board/raspberrypi4-64/rootfs-overlay/etc/init.d/S10i2cperms
chmod +x board/raspberrypi4-64/rootfs-overlay/etc/init.d/S10i2cperms
nano board/raspberrypi4-64/rootfs-overlay/etc/init.d/S10i2cperms
```

A continuación, agregue el siguiente contenido al archivo:

rootfs-overlay/etc/init.d/S10i2cperms

```
#!/bin/sh
case "${1}" in
    start)
        # Loops over all i2c devices, if any
        for iic in /dev/i2c*; do
            chown root:i2c "${iic}"
            chmod ug+rw "${iic}"
        done
        unset iic
        ;;

    *)
        echo "Usage: ${0} {start}"
        exit 1
        ;;
esac
```

Por último, crearemos los archivos que permitan la ejecución automática de cualquer script de arranque que nos sea conveniente:

1. Copie el archivo anexo `test.pyc` al directorio del usuario *pi*; es decir a `board/raspberrypi4-64/rootfs-overlay/home/pi`.
2. Cree un archivo de inicio `start.sh` en el mismo directorio, y márquelo como ejecutable con los siguientes comandos:

   ```
   touch board/raspberrypi4-64/rootfs-overlay/home/pi/start.sh
   chmod +x board/raspberrypi4-64/rootfs-overlay/home/pi/start.sh
   ```

3. Utilizando editor de texto plano como `nano` o `vim`, edite el archivo anterior y agregue el siguiente contenido al archivo:

   rootfs-overlay/home/pi/start.sh

   ```
   #!/usr/bin/sh

   cd /home/pi || exit -1
   if   [ -f test.pyc ]; then
     python3 test.pyc
   elif [ -f test.py  ]; then
     python3 test.py
   fi
   ```

4. Cree un archivo de autoarranque o *daemon* `S99autostart` en el mismo directorio `etc/init.d/` del *overlay*, y márquelo como ejecutable con los siguientes dos comandos:

   ```
   touch board/raspberrypi4-64/rootfs-overlay/etc/init.d/S99autostart
   chmod +x board/raspberrypi4-64/rootfs-overlay/etc/init.d/S99autostart
   ```

5. Utilizando editor de texto plano como `nano` o `vim`, edite el archivo `S99autostart` y agregue el siguiente contenido al archivo:

rootfs-overlay/etc/init.d/S99autostart

```
#!/bin/sh
case "${1}" in
    start)
        # Exits if /home/pi/start.sh does not exist
        [ -f /home/pi/start.sh ] || exit 0
        # Else executes it
        su - pi -c /home/pi/start.sh
        exit 0
        ;;

    *)
        echo "Usage: ${0} {start}"
        exit 1
        ;;
esac
```

¡Listo! Ahora su sistema puede ejecutar programas como el usuario *pi* automáticamente al arrancar. Sólo tiene que modidificar el archivo `home/pi/start.sh` de acuerdo a sus necesidades.

Opcionalmente, en caso de que se haya llevado a cabo la personalización con logotipo (véase Subsección 4.3), procederemos a automatizar la solución del inconveniente y borrar la pantalla de forma automática cuando se inicie sesión. Esto se logra de forma simple añadiendo dicho comando al final del archivo `.profile` ubicado en cada uno de los directorios de usuario, en nuestro caso `/root` y `/home/pi`.

Para resolver el inconveniente el caso del súperusuario *root* ejecute los siguientes comandos:

```
$ mkdir -p board/raspberrypi/rootfs-overlay/root
$ echo "clear" > board/raspberrypi/rootfs-overlay/root/.profile
```

A continuación, repetimos el procedimiento para la cuenta de usuario *pi*:

```
$ echo "clear" > board/raspberrypi/rootfs-overlay/home/pi/.profile
```

Listo. Hemos terminado de configurar el sistema de archivos.

4.5. Paso 5: Generación de la imagen del S.O. embebido

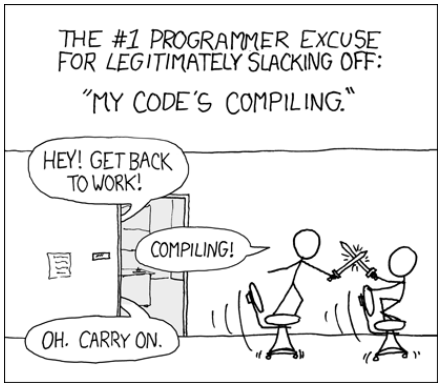
Aunque muy sencillo, compilar el kernel y los paquetes, y generar el archivo de imagen es un proceso largo que consume bastante tiempo.

Para crear la imagen del sistema embebido, sitúese en el directorio raíz de *buildroot* y ejecute el comando make.

```
$ make
```

Eso es todo. El proceso de compilación tardará aproximadamente entre 90 minutos y 4 horas dependiendo de la velocidad de la computadora huesped.

Relájese y espere.



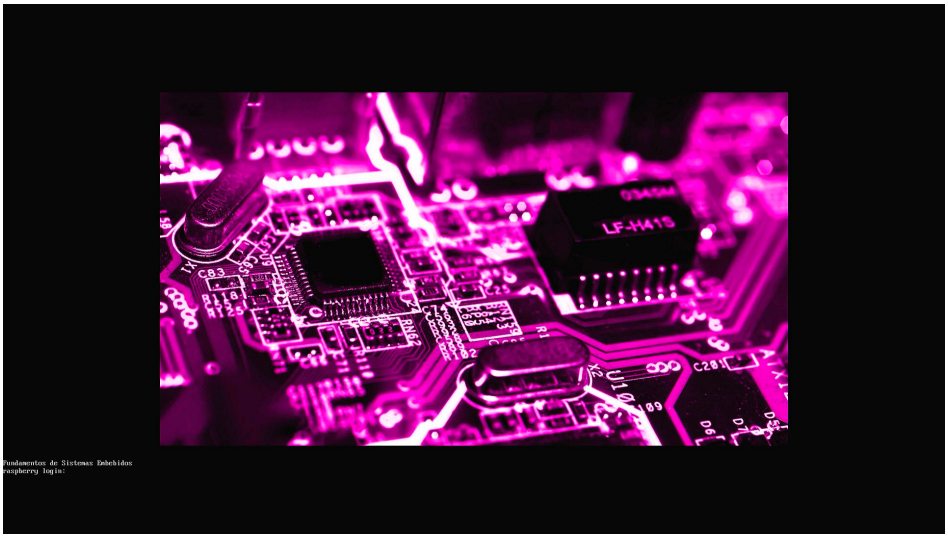


Figura 2: Sistema embebido en ejecución con logotipo personalizado

4.6. Paso 6: Grabación y prueba

Una vez que el sistema termine de compilar, encontrará la imagen generada `.img` dentro del directorio `output/images` de *buldroot*. Notará que el archivo de imagen no pesa más de 300MB. ¡Así de pequeño es su sistema operativo embebido!

Proceda a grabar la imagen generada en una memoria microSD utilizando Balena Etcher de la misma forma que ha grabado otras imágenes. Alternativamente puede grabar la imagen utilizando `dd` si conoce la ruta de la misma en su sistema con el comando:

```
# dd if=output/images/sdcard.img of=/dev/<memoriaSD> bs=10M status=progress
```

La grabación no debería tomar más que unos segundos.

Terminada la grabación, inserte la memoria microSD en la Raspberry Pi y observe el proceso de arranque, tendría que ver una pantalla similar a la de la Figura 2. Cuando se lo solicite, inicie sesión con el usuario `pi` y la contraseña `raspberry`.

Conecte el display al puerto I^2C (pines 3 y 5 del puerto o GPIO2 y GPIO3 en BCM) y ejecute el comando de detección de dispositivos

```
$ i2cdetect -y 1
```

tendría que ver una salida como la siguiente:

```
$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- 27 -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

5. Experimentos

1. [6pt] Grabe la imagen generada en la memoria microSD y pruebe el sistema operativo embebido en la Raspberry Pi. Como evidencia entregue un video donde se muestre el arranque de la Rasbperry Pi y el nombre de uno de los integranted de la brigada en el display.

2. [4pt] Con base en las instrucciones de la subsección 4.4 integre un programa modificado que despliegue en la primera línea del display el apellido paterno de cada integrante del equipo de trabajo, separados por espacio, como un corrimiento infinito de marquesina izquierda.

3. [+1pt] Con base en las instrucciones de la subsección 4.3 modifique la imagen generada para que despliegue un logotipo personalizado de su elección durante la carga del sistema operativo. Como evidencia entregue un video donde se muestre el arranque de la Rasbperry Pi y el texto en el display.

4. [+4pt] Modifique la solución para que el display muestre en la primera línea del display el apellido paterno de cada integrante del equipo de trabajo, separados por espacio, como un corrimiento infinito de marquesina izquierda además de la temperatura en grados centígrados reportada por el DS18B20 en la segunda línea.

   Hint: Necesitará habilitar la carga del controlador `w1-gpio` y recompilar la solución completa con `make clean && make`.

5. [+10pt] Combine la práctica actual con la práctica *Kiosco Multimedia* y desarrolle una imagen de sistema embebido capaz de reproducir videos y fotografías al mismo tiempo que despliega la marquesina y presenta la temperatura en un display.

6. Referencias

[1] The Buildroot developers. Buildroot 2025.02 manual. https://buildroot.org/downloads/manual/manual.html, 2008. [Online; accessed 18-March-2025].

[2] Peter Barry and Patrick Crowley. *Modern embedded computing: designing connected, pervasive, media-rich systems*. Elsevier, 2012.

[3] Linux Foundation and Yocto Project. Yocto project documentation. https://docs.yoctoproject.org/, 2010. [Online; accessed 18-March-2025].

[4] Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2016.

[5] The Buildroot developers. Buildroot website. https://buildroot.org, 2008. [Online; accessed 18-March-2025].

[6] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[7] Tammy Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012.

A. Speedrun: imágen lista en 15 minutos

En esta sección se explica cómo completar la configuración de la imagen del sistema operativo embebido (pasos 2, 3 y 4) en menos de 15 minutos.

1. Cree un directorio de trabajo y sítúese allí:

```
$ mkdir ~/myPiLinux
$ cd ~/myPiLinux
```

2. Descargue buildroot y descomprímalo. Luego cambie al directorio de buildroot:

```
$ wget https://buildroot.org/downloads/buildroot-2025.02.tar.xz -O - | tar -xJf -
$ cd buildroot-2025.02.tar.xz
```

3. Copie el archivo de fuentes anexo a esta práctica al directorio de buildroot.

4. Descomprma el archivo anexo con:

```
$ tar -h -xJf p08-sources.tar.xz
```

5. Seleccione la plataforma objetivo y ejecute `menuconfig`:

```
$ make raspberrypi4_64_defconfig
$ make menuconfig
```

6. Establezca los valores que se muestran en la Tabla 1.

7. Finalmente compile la solución con `make` para generar la imagen.

```
$ make
```

Paciencia

El proceso de compilación tardará entre 90 minutos y 4 horas.

Utilice ese tiempo para estudiar a fondo los pasos de la Sección 4 y preparar adecuadamente los experimentos propuestos de la Sección 5.

B. Añadir python-smbus2 a *buildroot*

Primero ubíquese en la raíz de buildroot.

A continuación cree un directorio para los archivos de configuración e inclusión de `smbus2` como sigue:

```
$ mkdir        package/python-smbus2
$ touch        package/python-smbus2/Config.in
$ touch package/python-smbus2/python-smbus2.hash
$ touch package/python-smbus2/python-smbus2.mk
```

Estos archivos le indicarán a *buildroot* que, en caso de que se seleccione el paquete, los binarios tendrán que ser descargados de internet, compilados para la arquitectura objetivo e incluidos en la imagen.

Edite el archivo `package/python-smbus2/Config.in` en un editor como `vi` o `nano` para que contenga exactamente el siguiente texto:

package/python-smbus2/Config.in

```
config BR2_PACKAGE_PYTHON_SMBUS2
  bool "python-smbus2"
  help
    smbus2 is a drop-in replacement for smbus-cffi/smbus-python
    in pure Python.

    https://github.com/kplindegaard/smbus2
```

Edite el archivo `package/python-smbus2/python-smbus2.hash` en un editor como `vi` o `nano` para que contenga exactamente el siguiente texto:

package/python-smbus2/python-smbus2.hash

```
# md5, sha256 from https://pypi.org/pypi/smbus2/json
md5  5708a6cbf052f45a3ad6dd83c00902a6  smbus2-0.5.0.tar.gz
sha256  4a5946fd82277870c2878befdb1a29bb28d15cda14ea4d8d2d54cf3d4bdcb035  smbus2-0.5.0.tar.
    gz
# Locally computed sha256 checksums
sha256  6ee9cf18c3a75dd76fb549a4b607ae34eedc31a796c48157895e2ad28d66ce79  LICENSE
```

Edite el archivo `package/python-smbus2/python-smbus2.mk` en un editor como `vi` o `nano` para que contenga exactamente el siguiente texto:

package/python-smbus2/python-smbus2.mk

```
PYTHON_SMBUS2_VERSION = 0.5.0
PYTHON_SMBUS2_SOURCE = smbus2-$(PYTHON_SMBUS2_VERSION).tar.gz
PYTHON_SMBUS2_SITE = https://files.pythonhosted.org/packages/10/c9/6
    d85aa809e107adf85303010a59b340be109c8f815cbedc5c08c73bcffef
PYTHON_SMBUS2_SETUP_TYPE = setuptools
PYTHON_SMBUS2_LICENSE = MIT
PYTHON_SMBUS2_LICENSE_FILES = LICENSE

$(eval $(python-package))
```

Hasta este momento el paquete `python-smbus2` aún no ha sido integrado a *buildroot*. Para integrarlo abra el archivo `package/Config.in` en un editor como `vi` o `nano`. Localice el paquete `python-cffi` aproximadamente en la línea 1400 y agregue una entrada similar abajo para smbus2. El archivo debería quedar como sigue:

package/Config.in

```
1399    source "package/python-smbprotocol/Config.in"
1400    source "package/python-smbus-cffi/Config.in"
1401    source "package/python-smbus2/Config.in"
1402    source "package/python-smmap2/Config.in"
1403    source "package/python-snappy/Config.in"
```

Por último, ejecute `make menuconfig` y verifique que la entrada smbus2 esté disponible.

C. Atajos

Dentro de *buildroot* `make` se convierte en una poderosa herramienta que permite tanto abrir utilerías de configuración como recompilar partes del sistema. Por practicidad (compilar un sistema completo es muy tardado), `make` no rastrea cambios de forma inteligente, por lo que en ocasiones es necesario recompilar paquetes de forma manual [1].
La siguiente tabla presenta una serie de atajos convenientes para el estudiante:

Tabla 2: Atajos para `make` [1]

Comando	Ejemplo	Descripción
`clean`	`make clean`	Borra todo el contenido del directorio de salida `build` para una recompilación limpia y desde cero.
`dirclean`	`make linux-dirclean`	Borra todo el contenido del directorio asociado al paquete especificado dentro de `build`.
`build`	`make python3-build`	Compila el paquete especificado (si no está compilado).
`rebuild`	`make python3-rebuild`	Recompila el paquete especificado sin actualizar ninguna configuración. Útil cuando se mdifica el código fuente en el directorio `build`.
`reconfigure`	`make linux-reconfigure`	Reconfigura y luego recompila el paquete especificado, pero no actualiza dependencias. Útil cuando se hacen cambios en `menuconfig`.

Asimismo, *buildroot* cuenta con alternativas gráficas a la interfaz de `menuconfig` como se muestra a continuación:

Tabla 3: Menús disponibles con `make` [1]

Comando	Descripción
`make menuconfig`	Interfaz clásica en modo consola.
`make nconfig`	Interfaz nueva y mejorada en modo consola.
`make xconfig`	Interfaz gráfica con Qt para entornos de escritorio.
`make qconfig`	Interfaz gráfica con GTK para entornos de escritorio.
`make linux-menuconfig`	Como `make-menuconfig` pero para configurar el kernel de linux.

Importante

Buildroot nunca borra cosas, sólo sobreescribe.

Si se eliminaron componentes, integraron parches nuevos o borraron archivos se recomienda invocar `make clean` para recompilar desde cero a fin de evitar problemas y conflictos.

Se puede encontrar más información en la sección 8.13 del manual de *buildroot*.