CAPÍTULO 1

CONFIGURACIÓN Y COMPILACIÓN DEL CÓDIGO

En este anexo se explican los conceptos básicos necesarios para trabajar con la *BeagleBoard*.

1.0.1. Obtención del código - Git

El código está almacenado en un repositorio git disponible en el DVD adjunto a esta documentación y en *github*: `git://github.com/rlrosa/uquad.git`. El repositorio ocupa aproximadamente 4GB, por lo que no es conveniente poner otras cosas en la tarjeta SD de la Beagleboard, ya que sino se excederá su capacidad.

1.0.2. Comunicación con la BeagleBoard

La forma básica de comunicarse con la Beagleboard es mediante el puerto serie, usando un conversor *RS2232* a USB. Durante el vuelo, la comunicación se establece mediante *WiFi*, usando un dongle USB. El procedimiento para hacerlo funcionar es el siguiente:

1. Copiar el firmware `scripts/rt73.bin` a la Beagleboard en `/lib/firmware/`.
2. Instalar el software necesario haciendo
   `opkg install kernel-module-rt73usb rt73-firmware`.

Una forma de conectarse es usando una red *Ad-Hoc*, aunque también se puede utilizar un servidor *DHCP*. Para configurar la interfaz `wlan0`[1] para trabajar en modo *Ad-Hoc* agregar las siguiente líneas a `/etc/network/interfaces`:

```
auto wlan0
iface wlan0 inet static
address 10.42.43.2
netmask 255.255.255.0
wireless-mode ad-hoc
```

[1] El dongle puede ser asociado a otra interfaz, como *wlan1*, configurar la apropiada.

```
wireless-essid uquad
wireless-key s:uquaduquad123
```

Luego, levantar una red en una laptop, en modo *Ad-Hoc*, con nombre *uquad* y contraseña *uquaduquad123* en *WEP 40/128-bit Key(Hex or ASCII)*. La laptop tendrá la IP *10.42.43.1*, y la Beagleboard *10.42.43.2*. El usuario en la Beagleboard es *root* y la clave cualquiera. Para conectarse ver la subsección sobre **ssh**.

Ssh

El usuario *root* acepta cualquier contraseña. Para evitar que la Beagleboard pida contraseña, se puede agregar la clave pública de una laptop a la Beagleboard, en[2].:

```
/home/root/.ssh/authorized_keys
```

Para conectarse hacer:

```
ssh root@10.42.43.2
```

Ethernet

La Beagleboard sabe conectarse a una red en la que haya un servidor *DHCP*. Elegirle un nombre en `/etc/hostname`, por ejemplo *beagle*, y luego hacer:

```
ssh root@beagle.local
```

Configuración del proxy

Para trabajar con la Beagleboard en el laboratorio de medidas, es necesario configurar el proxy. Para ello, ejecutar los siguiente comandos en la Beagleboard:

```
echo "option http_proxy http://httpproxy.fing.edu.uy:3128/" \
>> /etc/network/options.conf
echo "option http_proxy http://httpproxy.fing.edu.uy:3128/" \
>> /etc/opkg/arch.conf
```

1.0.3. Tiempos

El *main* corre sobre linux, lo cual simplifica algunas cosas, pero complica otras. Correr arriba de un sistema operativo que no es RT[3] inevitablemente introduce demoras. Se configura el *main* y el driver de los motores para tener máxima prioridad.

La performance del *main* es **INACCEPTABLE** si se accede a memoria no volátil o si se abren sesiones ssh durante su ejecución.

[2]Crear el archivo y/o el directorio, si estos no existen.
[3]RealTime.

1.0.4. Consideraciones de seguridad

Al comenzar el *main*, intentará establecer una conexión TCP con un servidor en la laptop, y abortará en caso de no tener éxito. Si en algún momento se pierde la conexión, por ejemplo por algún error en el driver del dongle *WiFi*, el *main* abortará, apagando los motores del cuadricóptero. Para iniciar el servidor en la laptop ubicarse en el directorio `src/build/check_net` y ejecutar `./server`.

En caso de perderse ls conexión ssh, y que el cuadricóptero siga funcionando (porque la conexión TCP sigue abierta), se puede detener el servidor que corre en la laptop, lo cual hará que el *main* aborte y apague los motores.

El *main* verifica que la inclinación del cuadricóptero no supere cierto límites durante un tiempo dado. Si esto sucede, apagará los motores. Esto puede ser útil si por algún motivo se pierde el control mediante software, pero se puede acceder al cuadricóptero, ya que inclinándolo lo suficiente se apagarían los motores.

El driver de los motores (*cmd*) tiene que estar funcionando para que los motores permanezcan prendidos. La pérdida de energía en la *BeagleBoard* detendría al driver, y por lo tanto también a los motores (El frenado tardaría unos segundos, ya que se omitiría la secuencia de frenado).

1.1. Salida - Logs

El *main* genera varios logs durante su ejecución, que sirven como realimentación al usuario para determinar la causa de errores o comportamientos extraños durante un vuelo:

- Datos crudos provenientes de la IMU: *imu_raw.log*
- Datos utilizados para estimar el estado: *kalman_in.log*
- Estado estimado: *x_hat.log*
- Acción de control: *w.log*

La primer columna de *imu_raw.log*, *kalman_in.log* y *w.log* es el tiempo en segundos desde el comienzo del *main*. La diferencia de tiempo entre *kalman_in.log* y *x_hat.log* es despreciable, por lo que *x_hat.log* no tiene columna de tiempos. Referirse al código fuente por información sobre como habilitar logs, nombres, contenido, etc.

Por cada log se crea un programa independiente (de ahora en más *logger*), que se comunica con el *main* mediante *pipes*. Cada logger tiene un espacio limitado en RAM (definido al iniciar el *logger*), donde irá almacenando lo que reciba del *main*. Cuando el espacio se acabe dejará de guardar información, mostrando una alerta en la consola. Solamente se escribirá a disco al finalizar la ejecución del *main*. Los logs se guardarán, por defecto, a `/media/sda1/`, una memoria flash comercial. Si no se dispone de una memoria flash se puede modificar los argumentos de arranque del *main* para que guarde en otro lado. Esto se implementó para evitar la escritura

a la tarjeta *microSD*, que es muy lenta y hace inaceptable la performance del *main*[4].

1.2. Debugging

Para probar el *main* sin volar el cuadricóptero, puede resultar cómodo apagar el control de connectividad. Para ello, setear `CHECK_NET_BYPASS` a 1 en

`src/common/uquad_config.h`

ADVERTENCIA: **NO** es recomendable hacer esto para una prueba con los motores prendidos, ya que la pérdida conexión implicaría la pérdida del control sobre el cuadricóptero.

1.2.1. Ejecución a partir de un log

Para debuggear el *main*, se lo puede correr con un log conocido como entrada, lo cual permite analizar el efecto de cambios concretos. Para esto se debe configurar el módulo que lee de la IMU, avisándole que debe leer de un log. Esto se logra seteando `IMU_COMM_FAKE` a 1, en `src/imu/imu_comm.h`.

C Vs. MatLab

Existe un script en *MatLab* que reproduce el comportamiento del *main*. Es muy útil para hacer pruebas y verificar el correcto funcionamiento del código en `C`:

`kalman/kalman_main.m`

Como argumento toma un log de datos crudos de la imu (*imu_raw.log*), como los generados por el *main* o por *imu_comm_test*:

`src/test/imu_comm_test/imu_comm_test.c`

1.3. Kernel

La Beagleboard corre linux *2.6.37*, de la distribución *Angstrom*:

- `http://www.angstrom-distribution.org/`

El kernel que viene por defecto es suficiente para *casi* todo. Solo hace falta configurar el puerto *i2c-2* para que trabaje a 333kHz (en lugar de 400kHz).

Para compilar el kernel se utilizó *Bitbake+OpenEmbedded*, y se lo compiló desde *Ubuntu* 11.10 (64bits).

La información que se presenta a continuación se obtuvo de [**?**], [**?**] y del canal IRC `#oe`.

[4]Dado que la versión actual del *logger* no accede a disco hasta que termina de ejecutarse el *main*, no debería ser un problema guardar a la *microSD*. Esto no ha sido probado.

1.3.1. Compilación: Bitbake+OpenEmbedded

Para poder compilar programas para la Beagleboard se puede usar un entorno de desarrollo como OpenEmbedded (de ahora en más *OE*) y la herramienta para compilar, *bitbake*. La herramienta *bitbake* maneja recetas que listan programas y sus dependencias, y se encarga de compilar las cosas en el orden apropiado.

Todo lo que usa *OE* se baja de internet mediante *git*. A veces hay problemas con los servidores, y es cuestión de probar nuevamente en otro momento (o cambiar de servidor, revisar proxy, etc). Compilar una imagen entera, como para una SD, no es fácil, lleva tiempo, requiere que todos los servidores funcionen y un poco de suerte.

Para compilar en *Ubuntu*:

1. Ejecutar `sudo dpkg-reconfigure dash` y en el menu elegir la opción *no*. Por más información ver `http://wiki.openembedded.org/index.php/OEandYourDistro`.

2. Descargar, configurar y actualizar el repositorio:

```
git clone git://git.angstrom-distribution.org/setup-scripts
cd startup-scripts
MACHINE=Beagleboard ./oebb.sh config Beagleboard
MACHINE=Beagleboard ./oebb.sh update
source ~/.oe/environment-oecore
```

3. Si se quisiera compilar ahora[5], hacer:

```
bitbake console-image
```

El script `~/.oe/environment-oecore` es responsable de generar variables que se utilizan durante la compilación. Cada vez que se abre una consola se cargan las variables globales y las declaradas en `~/.bashrc`, una almacena las rutas a los ejecutables instalados, otra las bibliotecas, etc. Las variables en `~/.oe/environment-oecore` hay que cargarlas cada vez que se abre una nueva consola. Una de las cosas que hace el script es indicar la ruta al programa *bitbake*. Para verificar que el script fue correctamente ejecutado se puede escribir el comienzo del comando `bit` y apretar *tab* para ver sugerencias. Entre las opciones debería aparecer el comando *bitbake*.

Las recetas que utiliza *bitbake* (y nos interesan más) se encuentran en:

```
setup_scripts/sources/meta-ti/
```

Algunos ejemplos:

- Configuración del kernel:

```
setup-scripts/sources/meta-ti/recipes-kernel/\
linux/linux-3.0/Beagleboard/defconfig
```

- Receta para el kernel:

[5] Compilar la imagen lleva mucho tiempo, mejor configurar todo antes de compilar.

```
setup_scripts/sources/meta-ti/recipes-kernel/\
linux/linux-omap_3.0.bb
```

- Receta para el u-boot:

```
setup-scripts/sources/meta-ti/recipes-bsp/\
u-boot/u-boot_2011.12.bb
```

Luego de haber incorporado las variables de `~/.oe/environment-oecore` ya no es necesario usar `MACHINE=Beagleboard ./oebb.sh`, se puede y debe usar directamente *bitbake*.

Es recomendable hacer `MACHINE=Beagleboard ./oebb.sh update` frecuentemente. Algunos paquetes necesarios para poder compilar correctamente (pueden faltar otros):

```
sudo apt-get install\
   python-ply python-progressbar\
   texi2html cvs subversion gawk\
   chrpath texinfo diffstat
```

Como modificar el contenido de OE

Para modificar un programa, como por ejemplo el *u-boot*:

```
bitbake -c devshell u-boot
# se abre una consola en el dir temporal del u-boot
emacs board/ti/beagle/beagle.h
# editar, por ejemplo habilitar la UART2
git add board/ti/beagle/beagle.h
git commit -m 'uquad: habilitando UART2'
git format-patch HEAD~1
cp 001-uquad:-habilitando-UART2.patch $OE_BASE/
# incrementar la línea que dice "PR = "r4"", ponerle r5
bitbake u-boot
# buscar el resultado en setup-scripts/build/tmp*/deploy/images
```

Para modificar el kernel y setear el *i2c-2* a 333kHz:

```
bitbake -c devshell virtual/kernel
# se abre una cosola en el dir temporal del kernel
# ANTES de cambiar nada, hacer:
quilt new uquad-set-i2c-2-333kHz.patch
# Si se quiere hacer cambios en board-omap3beagle.c:
quilt add arch/arm/mach-omap2/board-omap3beagle.c
emacs arch/arm/mach-omap2/board-omap3beagle.c
# editar, por ejemplo setear i2c a 333kHz
# Ahora pedirle a quilt que arme un patch
quilt refresh
# El patch queda en patches/uquad-set-i2c-2-333kHz.patch
# Se copia a meta-ti/recipes-kernel/linux-3.0/
# Se edita meta-ti/recipes-kernel/linux_3.0.bb, agregando una
```

```
# línea antes de la q dice defconfig:
file://uquad-set-i2c-2-333kHz.patch;patch=1 \
# Ahora se compila haciendo
bitbake virtual/kernel
```

Comandos útiles - bitbake+OE

Algunos comando que pueden ser de utilidad:

- Para compilar un paquete individual, sin tomar en cuenta las dependencias:

  `bitbake -b receta.bb`

  Ejemplo:

  `bitbake -b sources/meta-ti/recipes-bsp/u-boot/u-boot_2011.12.bb`

- Para ver todas las recetas que se ejecutan como dependencias, hacer:

  `bitbake <receta> -g`

  y luego mirar en `task-depends.dot`.
  Ejemplo:

  `bitbake console-base-image -g`

- Para borrar todo lo compilado sobre un paquete, por ejemplo el u-boot, hacer:

  `bitbake -c clean u-boot`

1.4. IMU

La Mongoose viene cargada con un *bootloader Arduino*, que permite bajarle código mediante el puerto serie.

El código original que trae la Mongoose fue modificado. El problema principal que tenía era que la frecuencia de muestreo no era estable.

Algunos cambios:

- Se implementó transmisión de datos en segundo plano, mediante interrupciones, evitando trancar el loop principal al momento de transmitir.
- Se agregó la posibilidad transmitir en binario (se mantuvo el modo `ASCII`, pero no es posible transmitir a 10ms en dicho modo).
- El barómetro demora varios milisegundos entre que se le pide un dato y que lo tiene disponible. El código original esperaba durante este tiempo. La versión modificada sigue trabajando y vuelve para recoger el dato luego que transcurrió el tiempo en cuestión.
- Se modificó el formato de los datos envíados al puerto serie.

1.4.1. Compilación

Para programar la Mongoose se utiliza *Arduino*. Para instalarlo hacer:

```
sudo apt-get install arduino
```

El código utiliza una biblioteca que no viene con *Arduino*. Asumiendo que el repositorio git fue descargado a `~/uquad`, para agregar la biblioteca al *Arduino*, hacer:

```
sudo ln -s ${HOME}/uquad/src/mongoose_fw/HMC58X3/\
 /usr/share/arduino/libraries/HMC58X3
```

1.5. Trabajo a futuro

- **Path planner:** Solamente está implementada la modalidad de *hovering*. Queda pendiente implementar rectas y círculos, y un sistema de *waypoints* que permita ir recorriendo trayectorias.
- **Control:** La matriz de control se carga de un archivo de texto. Para automatizar el cálculo de dicha matriz, permitiendo cambiar de trayectoria en tiempo real, habría que calcularla en `C`.
  En `src/control/control.c` está lista una implementación del algoritmo LQR, y de la linealización del sistema según la trayectoria elegida. Se verificó el correcto funcionamiento de la implementación comparando contra la versión en *MatLab*, pero no se lo incorporó al *main*.
  Un tema a considerar es el tiempo de computo, varios segundos según pruebas en una PC. Esto sería inaceptable durante el vuelo. Una solución simple sería hacer las cuentas de a poco, evitando pasar varios segundos sin ejecutar acciones de control.
- **Visión:** Queda pendiente implementar en `C` el algoritmo descrito en **??**.
- **Logger:** Una mejora sería que el *logger* guarde información en binario en lugar de usar `ASCII`, lo que permitiría ahorrar RAM. De cualquier forma, el RAM no es un problema por el momento.