DEPARTAMENTO DE ELECTRÓNICA
FACULTAD DE **I**NGENIERÍA DE LA **U**NIVERSIDAD DE **B**UENOS **A**IRES
CIUDAD AUTÓNOMA DE BUENOS AIRES ∼ REPÚBLICA ARGENTINA

Nota Técnica

Entorno de desarrollo de *firmware* sobre arquitectura ARM Cortex-M3, basado en herramientas libres

GCC + GDB + OpenOCD + Eclipse + GNU/Linux

Sebastián E. García ∼ GPSIC-FIUBA

Trabajo preparado para el Seminario de Sistemas Embebidos

v. 0.1 ∼ 2012.08.01

2012-SSE-FIUBA-NT01-01

Tabla de control de cambios

Versión	Fecha	Autor(es)	Comentario
0	2012.07.31	S. García.	Versión preliminar (difusión y obtención de *feedback*).
0.1	2012.08.01	L. Chiesa.	Correcciones menores.

Índice

- página en blanco -

1. Propósito

En esta Nota Técnica se resume la configuración de un entorno de desarrollo de *firmware* C/C++ *bare-metal* orientado a microcontroladores de las familias NXP LPC175x/6x, con núcleo procesador Cortex-M3 (32 bits, arquitectura ARM v. 7), basado en herramientas de *software* de desarrollo cruzado, libres, corriendo sobre un sistema operativo GNU/Linux.

Se exponen los pasos de configuración a modo de tutorial. La información brindada no es novedosa pero intenta agilizar la puesta en marcha de las herramientas, dando una base unificada y sólida, ya que las particularidades que aparecen al configurar un entorno de este tipo suelen consumir muchísimo tiempo.

Pretendemos facilitar así los primeros pasos con herramientas razonablemente eficientes, sin restricciones de uso y de muy bajo costo*, ampliamente utilizadas por desarrolladores de todo el mundo, aprovechando la experiencia de esa masa de usuarios a la hora de buscar la solución a los problemas que inevitablemente emergen durante el ciclo de vida de una dada *toolchain*, con las inexorables variantes que impone cada proyecto embebido en particular.

Los destinatarios inmediatos son los alumnos de la asignatura Seminario de Sistemas Embebidos de FIUBA pero, además, nos parece importante hacer pública esta Nota Técnica para todo desarrollador que la considere de utilidad, esperando además obtener una realimentación que enriquezca el documento, desde escenarios de aplicación bien reales y diversos.

Todos los pasos descritos fueron probados, pero por supuesto que pueden surgir inconvenientes debido a errores colados en el documento, diferentes sistemas locales, configuraciones previas, distintos casos de uso y complejidad de proyectos, etc. En este y todo otro sentido, serán bienvenidas colaboraciones, sugerencias y correcciones; por favor dirigirlas a la dirección de correo electrónico que aparece en la carátula.

2. Colaboraciones

En el contenido de este documento se incluyeron valiosos aportes técnicos (realizados, de manera consciente o inconsciente, en diferentes momentos de proyectos durante el último año) de las siguientes personas: Martín Ribelotta (UTN-FRBB/Emtech), Gastón Rodríguez (Emtech), Ariel Burman (FIUBA), Lucas Chiesa (FIUBA), Mauro Koenig (Emtech). Además, parte del *hardware* utilizado fué provisto por Guillermo Güichal (Emtech).

3. Referencias

Enlaces web válidos a la fecha de esta versión del documento.

[1] Embedded Artists. *LPC1769 LPCXpresso board*. http://www.embeddedartists.com/products/lpcxpresso/lpc1769_xpr.php.

[2] NXP. *LPC1769 microcontroller*. http://www.nxp.com/products/microcontrollers/cortex_m3/lpc1700/LPC1769FBD100.html.

[3] Emtech. *FTHL (v. 3)*. http://www.emtech.com.ar/downloads/productos/FTHL/Esquematico_FTHL_v3.pdf.

[4] Embedded Artists. *LPC1768 LPCXpresso board schematics (rev. A)* . http://laboratorios.fi.uba.ar/lse/tools/2012-ide/LPCXpressoLPC1768revA.pdf.

[5] Joint Test Action Group. *IEEE 1149.1 Standard test access port and boundary-scan architecture*.

[6] FTDI. *FT2232H Dual high speed USB to multipurpose UART/FIFO IC*. http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232H.pdf.

[7] Tex Live. *Tex document production system*. http://www.tug.org/texlive/.

[8] Open On-Chip Debugger. *OpenOCD User's guide* . http://laboratorios.fi.uba.ar/lse/tools/2012-ide/openocd.pdf.

[9] Eclipse foundation. *Eclipse Indigo*. http://www.eclipse.org/downloads/packages/release/indigo/sr2.

* Costo expresado, esencialmente, en términos de horas de ingeniería para la puesta a punto.

[10] ARM. *Cortex Microcontroller Software Interface Standard* . http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php.

[11] ARM. *CMSIS Downloads* . http://www.onarm.com/cmsis/download.

[12] NXP. *LPC175x and LPC176x CMSIS-Compliant Standard Peripheral Firmware Driver Library* . http://ics.nxp.com/support/documents/microcontrollers/zip/lpc17xx.cmsis.driver.library.zip, 2011-06-21.

[13] S. García. *Paquete de archivos base para proyectos GCC LPC17xx* . http://laboratorios.fi.uba.ar/lse/tools/2012-ide/prjt-base-lpc17xx-gcc.tar.gz.

4. Acrónimos y abreviaciones

API *Application Programming Interface*.

ARM Empresa *ARM Holdings, plc* (ex- *Advanced RISC Machines*), Reino Unido.

BSP *Board Support Package*.

CDI *C/C++ Debugging Interface*.

CDT *C/C++ Development Tooling*.

CMSIS *Cortex Microcontroller Software Interface Standard*, ARM.

DSF *Debugger Services Framework*.

EABI *Embedded-Application Binary Interface*.

Emtech Empresa Emtech SA, Argentina.

FIUBA Facultad de Ingeniería de la Universidad de Buenos Aires.

FTDI Empresa *Future Technology Devices International, Ltd*.

FTHL Dispositivo adaptador USB-JTAG, Emtech SA.

FW *Firmware*.

GPIO *General-Purpose Input-Output*.

GUI *Graphical User Interface*.

HAL *Hardware Abstraction Layer*.

HW *Hardware*.

I/F Interfaz.

IDE *Integrated Development Environment*.

IEEE *Institute of Electrical and Electronics Engineers*.

JTAG *Joint Test Action Group*.

LPX Placa *LPCXpresso LPC1768, Embedded Artists AB*.

MPSSE *Multi-Protocol Synchronous Serial Engine*.

NXP Empresa *NXP Semiconductors* (ex- *Philips Semiconductors*), Holanda.

OCD *Open on-Chip Debugger*.

PCB *Printed Circuit Board*.

SoC *System-on-Chip*.

SSE Seminario de Sistemas Embebidos, FIUBA.

SW *Software*.

TBC Pendiente, a completar.

TBD Pendiente, a definir.

TAP *Test Access Port*.

USB *Universal Serial Bus*.

UTN-FRBB Universidad Tecnológica Nacional, Facultad Regional Bahía Blanca.

5. Introducción

Las herramientas de *software* consideradas en esta Nota, son:

- Interfaz OpenOCD para *debugging* "sobre el chip",
- *Toolchain* GNU, adaptación "Mentor Graphics Sourcery CodeBench Lite",
  - compilador cruzado `gcc` C/C++ ,
  - ensamblador `as` ,
  - *linker* `ld` ,
  - librerías std. C/C++ ,
  - *debugger* `gdb` ,
  - y un conjunto de herramientas accesorias.
- Entorno gráfico de desarrollo Eclipse.

En cuanto a las herramientas de *hardware*, en este trabajo se aplica la conocida plataforma de bajo costo LPCXpresso LPC1768*, fabricada por la firma sueca Embedded Artists AB. Se utilizará sólo la sección correspondiente al microcontrolador *target*, ya que el adaptador USB-JTAG incluído en la misma placa, "LPC-Link" no es compatible con las herramientas de *software* propuestas.

Si bien se cubre la aplicación sobre la familia LPC175x/6x de NXP, es deseable la extensión de esta Nota para cubrir dispositivos *target* de otras arquitecturas (e.g., Cortex-M0/Cortex-M0+) u otros fabricantes (e.g., TI Stellaris, AT91SAM3x, STM32, Microsemi SmartFusion, etc.)†.

En relación al adaptador físico USB-JTAG, existe una gran variedad de alternativas compatibles con OpenOCD. Aquí se eligió el dispositivo adaptador "FTHL" fabricado en Argentina por la firma Emtech SA. Este módulo se basa en el conocido chip FT2232H. El diagrama esquemático se encuentra disponible ([3]), siendo un circuito de fácil construcción.

La distribución GNU/Linux sobre la cual se ensayaron los procedimientos, es Ubuntu 10.04 LTS, 32-bit, actualizada a julio de 2012‡. Por simplicidad para los usuarios que provienen de Windows, en esta Nota se hace mención a la aplicación gráfica `gedit` para la edición de archivos (de configuración) de texto, pero es bueno mencionar que para esto habitualmente se utiliza, por practicidad, un editor en modo línea de comandos (e.g. `vim`, vale la pena dedicar unos minutos a aprender sus comandos básicos).

6. *Hardware*

El objetivo de esta sección es ofrecer una guía con cierto detalle para la preparación de la plaquita propuesta (LPCXpresso *target*), especialmente orientada a quienes estén poco experimentados en temas de HW. Desde ya, en caso de utilizar otro *hardware* al propuesto aquí, se deberán saltear estos párrafos y en su lugar hacer los preparativos equivalentes. Durante el desarrollo de una placa prototipo§, se dejarán disponibles las señales del puerto JTAG en un conector de formato conveniente.

6.1. Placa LPCXpresso 1768: Preparación

Para acceder vía I/F JTAG al microcontrolador LPC1768 presente en la placa LPCXpresso, reemplazando el adaptador original LPC-Link por un adaptador USB-JTAG abierto, será necesario interrumpir eléctricamente la interconexión entre la sección *target* y la sección correspondiente al LPC-Link.

La opción mas simple consiste en quitar los puentes de soldadura entre ambas secciones de la placa. En la versión (2010) disponible de la placa utilizada en este trabajo, las 8 pistas que unen estos sectores lo hacen a través de un conector serigrafiado con etiqueta J4, de 2x8 *pins*, cuyos terminales pares e impares se encuentran unidos por mini-puentes de soldadura. Tener presente la página 4 del diagrama esquemático [4]. Para quitar estos mini-puentes entre filas de *pins* pares e impares, se recomienda:

- Utilizar una malla desoldante de buena calidad (e.g., Chemtronics *Chem-Wick Rosin SD* #4 o menor), aportando una pequeña cantidad de flux (líquido o en pasta) adicional.

* Cabe notar que, actualmente, esta placa ha sido reemplazada por una muy similar [1], basada en el LPC1769 [2]. † Son bienvenidos los eventuales comentarios sobre pruebas con alguna de estas variantes. ‡ Son bienvenidos los comentarios de instalación y uso en Ubuntu 12.04 LTS, u otras distribuciones. § Donde el microcontrolador, en general, podría formar parte de una cadena JTAG junto a otros dispositivos.

- Tener en cuenta el mayor aporte de calor necesario al desoldar el puente entre los *pins* J4.15 y J4.16 (señal GND, conectada a área cobreada *copper-pour* de PCB a través de un relieve térmico).
- Remover el flux sobrante con alcohol isopropílico.

En la figura 1 se muestra el resultado.

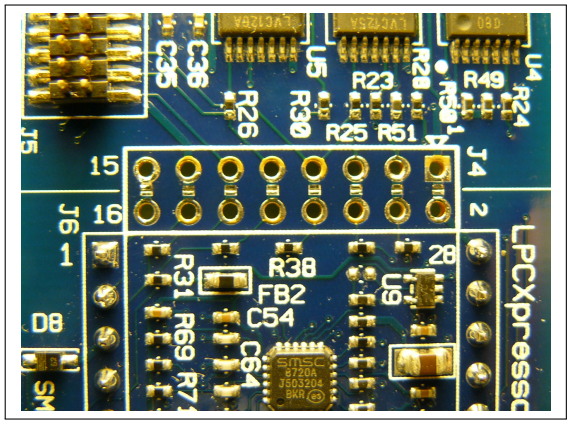

Figura 1: Mini-puentes desoldados

Si es requerido (e.g., por cuestiones de espacio en la aplicación), la placa original podría cortarse en medio del *land-pattern* de J4, teniendo cierto cuidado (la versión sobre la que se trabajó no posee troqueles ni *v-scoring*), separando definitivamente ambas secciones. Notar que es mas fácil separarlas cuando aún no se han soldado las 2 tiras de 27 *pins* del lado *target*. Esto puede hacerse trazando una línea con la ayuda de una regla metálica, mediante varias pasadas de un *cutter* o herramienta tipo bisturí, bien afilado, de ambos lados de la placa, procurando remover la fibra del PCB con una sección transversal en forma de "V".

En nuestro caso se realizó el corte de la placa, quitando definitivamente el adaptador LPC-Link. Entonces, la alimentación se aplicará mediante una fuente de tensión regulada de 3.3V entre los *pins* J6.28 (VIO_3V3X) y J6.1 (GND), y será compartida con el resto del circuito de aplicación presente en la placa madre donde finalmente se inserte el módulo*.

Finalmente, soldar en los *pads* impares de J4 una tira de 8 *pins* con *pitch* estándar de 100mils. A través de este conector se accederá con el adaptador JTAG.

6.2. Conexión entre el adaptador JTAG y la placa *target*

Deberá armarse un cable de interconexión entre la placa adaptadora USB-JTAG (en nuestro caso el dispositivo FTHL) y la placa *target*. Siguiendo la nomenclatura de los diagramas esquemáticos correspondientes [3, 4], la tabla 1 indica el mapeo necesario entre señales de ambas placas.

señal FTHL	*pin* FTHL	*pin* LPX *target*	señal LPX *target*
TMS	J2.11	J4.4	JTAG_TMS_SWDIOX
TCK	J2.5	J4.6	JTAG_TCLK_SWCLKX
TDO	J2.7	J4.8	JTAG_TDO_SWOX
TDI	J2.9	J4.10	JTAG_TDIX
RESET	J2.4	J4.12	JTAG_RESETX
DGND	J2.3, J2.6	J4.16	GNDX

Tabla 1: Conexiones JTAG entre adaptador y placa *target*

En la placa *target*, las líneas VIO_3V3X (J4.2) y EXT_POWX (J4.14) quedarán sin conectar. La tensión de alimentación se proveerá desde la placa madre de aplicación (donde va enchufado el módulo *target*) hacia las líneas VIO_3V3X (J6.28) y GNDX (J6.1 y J6.54) a través de las tiras de *pins* J6. Por otra parte, el adaptador FTHL toma su propia alimentación desde el puerto USB de la computadora *host*.

* Por supuesto que, si la placa LPCXpresso se mantuviese sin cortar y sin desoldar los puentes correspondientes a líneas de alimentación, se podrían utilizar los medios de alimentación originales del LPC-Link (usando su regulador de tensión): 5V provistos por su puerto USB, o 5V vía EXT_POWX (aplicada en J6.2), con los cuidados pertinentes.

Vale mencionar que el adaptador FTHL provee, además, un *bridge* USB-UART con señales B_TXD y B_RXD de niveles lógicos LV-TTL (3.3V), el que normalmente suele aprovecharse para conectarlo en forma cruzada a las señales de una UART libre del microcontrolador LPC1768. De este modo, se lo puede utilizar como canal de monitoreo adicional, durante el *debugging* de la aplicación.

7. OpenOCD

La herramienta de SW libre* OpenOCD tiene como objetivo permitir *debugging*, programación de memorias "en circuito" e interacción *boundary-scan* en dispositivos de procesamiento embebido.

Para lograr el acceso físico al *target*, OpenOCD se asiste de una pequeña herramienta de HW (llamada habitualmente "cable adaptador", *"dongle"*, etc.). Esta permite la comunicación entre la computadora *host* (donde corre OpenOCD) y la placa conteniendo al microcontrolador *target*, mediante las señales eléctricas apropiadas. Entre otras opciones, OpenOCD soporta la norma ampliamente adoptada IEEE 1149.1 "JTAG" [5], para lo cual es necesario utilizar una herramienta adaptadora desde un puerto estándar de la computadora *host* (e.g., USB), a la interfaz definida por JTAG, que permita el acceso al puerto TAP del procesador objetivo. Esa función la cumple el mencionado dispositivo adaptador FTHL, el cual está basado en el circuito integrado FT2232H [6], *bridge* de USB 2.0(HS) a MPSSE, donde esta última máquina de estados implementa el *standard* JTAG.

Para mayor información, referirse a la guía de usuario de OpenOCD (ver sección §7.2).

7.1. Instalación de *driver* para chip FTDI

En los siguientes pasos, ejecutados desde una ventana terminal de comandos *bash* (lanzada con `[CTRL+ALT+t]`), se instalará el *driver* necesario para el chip FTDI FT2232.

Comenzamos descargando e instalando el paquete `libusb`.

```
$ sudo apt-get install libusb-dev
```

Descargamos los fuentes de la librería `libftdi` y extraemos sus archivos en el directorio *home*.

```
$ cd ~
$ wget http://www.intra2net.com/en/developer/libftdi/download/libftdi-0.20.tar.gz
$ tar -xvzf libftdi-0.20.tar.gz
```

Copiamos el archivo `~/libftdi-0.20/src/ftdi.h` en el directorio `/usr/include` , y creamos un enlace simbólico desde `/usr/local/include/ftdi.h` :

```
$ sudo cp ~/libftdi-0.20/src/ftdi.h /usr/include
$ sudo ln -s /usr/include/ftdi.h /usr/local/include/ftdi.h
```

Compilamos e instalamos `libftdi`:

```
$ cd ~/libftdi-0.20
$ ./configure
$ make
$ sudo make install
```

Creamos los enlaces simbólicos a las librerías:

```
$ sudo ln -s /usr/local/lib/libftdi.a /usr/lib/libftdi.a
$ sudo ln -s /usr/local/lib/libftdi.la /usr/lib/libftdi.la
$ sudo ln -s /usr/local/lib/libftdi.so.1.20.0 /usr/lib/libftdi.so.1.20.0
$ sudo ln -s /usr/local/lib/libftdi.so.1.20.0 /usr/lib/libftdi.so.1
$ sudo ln -s /usr/local/lib/libftdi.so.1.20.0 /usr/lib/libftdi.so
```

Conviene establecer una regla `udev`† para utilizar el *driver* sin necesidad de hacer `sudo`. Crear un archivo:

```
$ gksudo gedit /etc/udev/rules.d/70-jtag.rules
```

, con el siguiente contenido:

```
# Uso de driver FT2232, desde usuario raso
SUBSYSTEM=="usb" ENV{DEVTYPE}=="usb_device", SYSFS{idVendor}=="0403",
SYSFS{idProduct}=="6010", GROUP="plugdev", MODE="0660"
```

Finalmente, reiniciamos `udev`.

* Licencia GNU GPL v. 2. † Provee el manejo dinámico de dispositivos en Linux, e.g. equipos conectados espontáneamente a un puerto USB.

```
$ sudo service udev restart
```

Para mayor información sobre `udev` y el contenido de este archivo, hacer:

```
$ man udev
```

7.2. Instalación y configuración de OpenOCD

Se instalará OpenOCD en su versión 0.5.0. Comenzamos descargando y extrayendo en *home* el código fuente de OpenOCD.

```
$ cd ~
$ wget http://download.berlios.de/openocd/openocd-0.5.0.tar.bz2
$ tar -xvjf openocd-0.5.0.tar.bz2
```

A continuación descargamos las dependencias para poder compilar OpenOCD. Si en la instalación GNU/Linux no se cuenta con los paquetes `texlive` y `texinfo` (procesamiento de documentación), el siguiente comando los instalará, lo que puede ser indeseado por el usuario, debido a su gran tamaño (∼350MB).

```
$ sudo apt-get build-dep openocd
```

En lugar del comando anterior, si no se desea instalar los paquetes `texlive` y `texinfo` , puede hacerse:

```
sudo apt-get install cdbs debhelper autotools-dev libftdi-dev chrpath
```

Compilamos OpenOCD, habilitando el soporte al chip FT2232.

```
$ cd ~/openocd-0.5.0
$ ./configure --enable-maintainer-mode --enable-ft2232_libftdi
$ make
$ sudo make install
```

Para probarlo, vamos a hacer uso de unos *scripts* de configuración, y una pequeña aplicación de test.

```
$ cd ~
$ wget http://laboratorios.fi.uba.ar/lse/tools/2012-ide/openocd-scripts.tar.gz
$ wget http://laboratorios.fi.uba.ar/lse/tools/2012-ide/test-blinky-gold.elf.tar.gz
$ mkdir ~/temp
$ tar -xvzf openocd-scripts.tar.gz -C ~/temp
$ tar -xvzf test-blinky-gold.elf.tar.gz -C ~/temp
```

Previo a iniciar OpenOCD, verificamos que todo el HW involucrado se encuentra conectado y alimentado. Arrancamos entonces OpenOCD, pasándole el *script* de configuración `openocd.cfg` . En este archivo esencialmente se invoca a `fthl.cfg` y `lpc1768.cfg` *, a la vez que se cargan configuraciones a medida de la implementación particular. En el apéndice §A se encuentran documentados todos los archivos de configuración utilizados.

```
$ cd ~/temp
$ openocd -f openocd.cfg
```

En realidad, mas allá del ejemplo, si no pasamos como argumento un dado *script* al arrancar `openocd` , esta aplicación busca por defecto un archivo llamado `openocd.cfg` en el directorio actual.

La respuesta a esta llamada tendrá una forma similar a la siguiente†.

```
Open On-Chip Debugger 0.5.0 (date-time)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.berlios.de/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
10 kHz
adapter_nsrst_delay: 200
jtag_ntrst_delay: 200
10 kHz
srst_only separate srst_gates_jtag srst_open_drain
Info : max TCK change to: 30000 kHz
Info : clock speed 10 kHz
Info : JTAG tap: lpc1768.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : lpc1768.cpu: hardware has 6 breakpoints, 4 watchpoints
```

* Correspondientes al HW utilizado: adaptador y *target*. † Si en el *script* se ha habilitado la línea "debug_level 3", la salida en pantalla será mucho mas verbosa.

Al finalizar la etapa de configuración, OpenOCD verifica la cadena JTAG definida con estos comandos (en nuestro caso se trata del único dispositivo JTAG existente en la placa, el LPC1768). Luego OpenOCD corre como *daemon*, esperando conexiones desde clientes (`telnet`, GDB, etc.), y procesa los comandos producidos a través de esos canales.

Dejamos abierto el terminal *bash* que veníamos usando y abrimos otro terminal para probar el correcto acceso mediante `telnet` (OpenOCD reserva por defecto el puerto puerto 4444 para este tipo de conexiones).

```
$ telnet localhost 4444
```

La respuesta deberá tener una forma similar a la siguiente, ofreciéndonos un *prompt* para ingresar comandos.

```
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

Los siguientes pasos de esta sección tienen el objetivo de verificar que las operaciones básicas de OpenOCD funcionan correctamente. Los comandos son ingresados por el terminal `telnet`, mientras que en el terminal anterior, desde donde lanzamos `openocd`, podemos monitorear un detalle mas verboso de las operaciones.

Probamos borrar toda la memoria *flash* del microcontrolador:

```
> halt
> flash erase_sector 0 0 last
erased sectors 0 through 29 on flash bank 0 in 6.623687s
```

A continuación, chequeamos que el borrado haya sido exitoso. Este comando demora un poco; puede obtenerse *feedback* de las operaciones en el primer terminal *bash*, si le pasamos a `openocd.cfg` el comando con la opción "`debug_level 3`".

```
> flash erase_check 0
successfully checked erase state
  #  0: 0x00000000 (0x1000 4kB) erased
  #  1: 0x00001000 (0x1000 4kB) erased
  #  2: 0x00002000 (0x1000 4kB) erased
  #  3: 0x00003000 (0x1000 4kB) erased
  #  4: 0x00004000 (0x1000 4kB) erased
  #  5: 0x00005000 (0x1000 4kB) erased
  #  6: 0x00006000 (0x1000 4kB) erased
  #  7: 0x00007000 (0x1000 4kB) erased
  #  8: 0x00008000 (0x1000 4kB) erased
  #  9: 0x00009000 (0x1000 4kB) erased
  # 10: 0x0000a000 (0x1000 4kB) erased
  # 11: 0x0000b000 (0x1000 4kB) erased
  # 12: 0x0000c000 (0x1000 4kB) erased
  # 13: 0x0000d000 (0x1000 4kB) erased
  # 14: 0x0000e000 (0x1000 4kB) erased
  # 15: 0x0000f000 (0x1000 4kB) erased
  # 16: 0x00010000 (0x8000 32kB) erased
  # 17: 0x00018000 (0x8000 32kB) erased
  # 18: 0x00020000 (0x8000 32kB) erased
  # 19: 0x00028000 (0x8000 32kB) erased
  # 20: 0x00030000 (0x8000 32kB) erased
  # 21: 0x00038000 (0x8000 32kB) erased
  # 22: 0x00040000 (0x8000 32kB) erased
  # 23: 0x00048000 (0x8000 32kB) erased
  # 24: 0x00050000 (0x8000 32kB) erased
  # 25: 0x00058000 (0x8000 32kB) erased
  # 26: 0x00060000 (0x8000 32kB) erased
  # 27: 0x00068000 (0x8000 32kB) erased
  # 28: 0x00070000 (0x8000 32kB) erased
  # 29: 0x00078000 (0x8000 32kB) erased
```

Ahora cargaremos en memoria *flash* una pequeña aplicación que se manifieste externamente, por ejemplo actuando sobre el parpadeo del LED incluído en la placa LPCXpresso *target* [*].

```
> flash write_image erase /home/sgarcia/temp/test-blinky-gold.elf 0 elf
auto erase enabled
Padding image section 0 with 4 bytes
Verification will fail since checksum in image (0x00000000) to be written to flash is different
     from calculated vector checksum (0xefff79de).
To remove this warning modify build tools on developer PC to inject correct LPC vector checksum
    .
wrote 4096 bytes from file /home/sgarcia/temp/test-blinky-gold.elf in 14.548713s (0.275 KiB/s)
```

Tener presente que en el último paso debe ingresarse el *path* completo del archivo ELF.

Podemos leer un segmento[†] de la memoria *flash*, por ejemplo entre las posiciones `0` y `0x1000` :

```
> dump_image a-ver-si-esta.bin 0x0 0x1000
dumped 4096 bytes in 5.404576s (0.740 KiB/s)
```

Notar que en el archivo obtenido, además del código de máquina de la pequeña aplicación, aparecen posiciones vacías al final (`0xFF`) hasta completar el tamaño solicitado a `dump-image`. El contenido neto en bytes del *firmware* deberá coincidir con el tamaño del archivo binario (crudo) de la aplicación[‡].

Comenzamos la ejecución de la aplicación con la secuencia de comandos: `reset init` y `resume` :

```
> reset init
JTAG tap: lpc1768.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x1fff0080 msp: 0x10001ffc
> resume
```

Congelamos la ejecución del *firmware*:

```
> halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x81000000 pc: 0x000002f2 msp: 0x10007fd0
```

Mientras que con el comando `resume` la aplicación deberá continuar su ejecución.

```
> resume
```

Con `help` podemos ver los comandos disponibles, mientras que `help comando_x` nos entrega una breve descripción de `comando_x` .

El aplicativo de *debugging* GDB se conectará (por defecto) al puerto TCP/IP 3333.

Para mayor información, referirse a la guía del usuario de OpenOCD (ver sección §7.2). Si disponemos de un conjunto de herramientas de procesamiento de fuentes LaTeX (e.g. Tex Live [7]) funcionando sobre el *host* Linux, podremos construir el manual de usuario, haciendo:

```
$ cd ~/openocd-0.5.0
$ make pdf
```

El archivo `openocd.pdf` se generará en el subdirectorio `~/openocd-0.5.0/docs` . Alternativamente, se hizo disponible este documento en el sitio web [8].

8. Herramientas *Mentor Graphics Sourcery CodeBench Lite*

Este conjunto de herramientas[§] C/C++ libres[¶], incluye esencialmente (en nuestro caso para *target* ARM EABI *bare-metal*) las herramientas: `gcc`, `as`, `ld`, y `gdb`.

Comenzamos esta sección descargando[‖] y expandiendo el paquete *tarball* con los archivos binarios (precompilados) de CodeBench.

```
$ cd ~
$ wget https://sourcery.mentor.com/sgpp/lite/arm/portal/package8734/public/arm-none-eabi/arm
    -2012.03-56-arm-none-eabi-i686-pc-linux-gnu.tar.bz2
$ tar -xvjf arm-2012.03-56-arm-none-eabi-i686-pc-linux-gnu.tar.bz2
```

[*] Se estima que la advertencia que aparece, *"Verification will fail. . . "*, es inócua y se debe a una inconsistencia en la versión de la *toolchain* utilizada para compilar el *firmware*. [†] Levantar la memoria *flash* completa (512KB), llevaría mucho tiempo. [‡] Fácilmente obtenible con el utilitario Objdump de la *toolchain* que instalaremos a continuación. [§] Previamente conocido como *CodeSourcery G++ Lite* [¶] Licencia GNU GPL v. 2. [‖] Si se copia y pega desde este PDF la dirección URL, corregirla mediante un editor de texto, ya que aparece un salto de línea en el medio.

Creamos el directorio de destino y movemos los binarios.

```
$ sudo mkdir /opt/codebench
$ sudo mv ~/arm-2012.03 /opt/codebench
$ rm -rf ~/arm-2012.03
```

Para actualizar el *path*, editamos el archivo:

```
$ gedit ~/.profile
```

, agregando la siguiente línea al final.

```
PATH=/opt/codebench/arm-2012.03/bin:$PATH
```

Para que se tomen los cambios en la sesión actual, debemos hacer por única vez:

```
$ source ~/.profile
```

Finalmente, hacemos una verificación mínima:

```
$ arm-none-eabi-g++ -v
Using built-in specs.
<snip>
gcc version 4.6.3 (Sourcery CodeBench Lite 2012.03-56)
```

La documentación de la *toolchain* quedará ubicada en el directorio: `/opt/codebench/arm-2012.03/share/doc/arm-arm-none-eabi/pdf` .

9. Eclipse

El entorno de desarrollo libre* Eclipse, permite la integración de las herramientas necesarias durante el ciclo de desarrollo de SW, a la vez que busca simplificar su uso a través de una misma interfaz GUI.

9.1. Instalación de Eclipse (C/C++)

En primer lugar, necesitamos instalar un *Java Runtime Environment*. Instalamos `openjdk-6-jre` .

```
$ sudo apt-get install default-jre
```

Bajamos† el paquete "Eclipse IDE for C/C++ developers" correspondiente a Eclipse Indigo Sr2 (v. 3.7.2), disponible en [9]. Lo expandimos, lo movemos al directorio `/opt` y creamos una carpeta de trabajo en *home*.

```
$ cd ~
$ wget http://espelhos.edugraf.ufsc.br/eclipse//technology/epp/downloads/release/indigo/SR2/
    eclipse-cpp-indigo-SR2-incubation-linux-gtk.tar.gz
$ tar -xvzf eclipse-cpp-indigo-SR2-incubation-linux-gtk.tar.gz
$ sudo mv ~/eclipse /opt
$ rm -rf ~/eclipse
$ mkdir ~/eclipse_prjts
```

Para actualizar el *path*, editamos el archivo:

```
$ gedit ~/.profile
```

, agregando la siguiente línea al final.

```
PATH=$PATH:/opt/eclipse
```

Para que se tomen los cambios en la sesión actual, debemos hacer por única vez:

```
$ source ~/.profile
```

9.2. Instalación de *plugins*

Arrancamos Eclipse para instalar desde su GUI los *plugins* necesarios.

```
$ eclipse &
```

Al iniciar, aparece una ventana emergente *"Select workspace"*; ingresar: `~/eclipse_prjts` . Cerrar la solapa *"Welcome"*.

* Licencia *Eclipse Public License* v. 1.0 . † Si se copia y pega desde este PDF la dirección URL, corregirla mediante un editor de texto, ya que aparece un salto de línea en el medio.

9.2.1. Paquetes CDT optativos

Vamos al menú y seleccionamos: *"Help -> Install New Software..."* En el campo *"Work with"* de la ventana emergente, ingresamos `http://download.eclipse.org/tools/cdt/releases/indigo` , y presionamos *"Enter"*. Luego de un breve intervalo de tiempo aparecerán los *plugins* disponibles para instalar.

Expandimos el grupo *"CDT Optional features"* y tildamos la casilla *"C/C++ GDB Hardware Debugging"*. Presionamos: *"Next", "Next", "Accept the license agreement", "Finish"*. Cuando se complete la instalación, reiniciar Eclipse seleccionando "Restart now" en la ventana emergente.

9.2.2. Paquetes GNU ARM

Nuevamente, vamos al menú y seleccionamos: *"Help -> Install New Software..."* En el campo *"Work with"* de la ventana emergente, ingresamos `http://gnuarmeclipse.sourceforge.net/updates` , y presionamos *"Enter"*. Luego de un breve intervalo de tiempo aparecerán los *plugins* disponibles para instalar.

Tildamos la casilla *"CDT GNU Cross Development Tools"* Presionamos: *"Next", "Next", "Accept the license agreement", "Finish"*. Si durante el proceso de instalación de este *plugin* aparece una notificación del tipo *"Unsigned plugin"*, presionar *"OK"*. Cuando se complete la instalación, reiniciamos Eclipse seleccionando *"Restart now"* en la ventana emergente.

10. Estructura base de un proyecto orientado a LPC17xx

En general, un proyecto embebido basado en las herramientas de compilación GNU necesita de, al menos, los siguientes archivos: librerías de paquete BSP*, código fuente de la aplicación particular, *linker script*†, *makefile* que permita simplificar (mediante la herramienta `make`) los pasos involucrados en un *build* del proyecto. Por supuesto, cuando utilizamos un entorno IDE como Eclipse, además se sumarán (automáticamente) los archivos correspondientes a configuraciones, *makefiles* automáticos, directorios de archivos de salida, etc.

Como parte del paquete BSP, aquí adoptamos la conocida capa HAL propuesta por ARM, incluída en su *standard* CMSIS‡ [10, 11]. A nuestro mejor conocimiento, la versión mas actualizada de librerías provistas por NXP para su familia de microcontroladores LPC1700, es la disponible en el sitio web [12] (basada en CMSIS v. 2). A grandes rasgos, ésta esencialmente comprende definiciones/funciones correspondientes a dos subsistemas: por un lado el *core* microprocesador licenciado por ARM, y por otro lado los periféricos provistos por el fabricante del chip, en este caso NXP. Cabe notar que en el paquete [12] (v. 2011-06-21) entregado por NXP, existen archivos adicionales: un conjunto de "drivers" (en realidad, APIs de bajo nivel) de periféricos que aplican las definiciones CMSIS, ejemplos de uso, archivos orientados a distintas herramientas, etc.

Acompañamos a esta Nota con una propuesta de un conjunto mínimo de archivos necesarios [13], a incluír en cada proyecto. Los archivos fueron seleccionados del paquete NXP mencionado [12], prescindiendo de aquellos que se consideraron no relevantes. Además, se tornó inevitable hacer algunas correcciones menores§ sobre los archivos seleccionados, por lo tanto se recomienda el uso de los mismos en lugar de los originales.

Esta es una organización simple de trabajo para nuestros ejemplos, disponiendo de los fuentes de CMSIS y *drivers* en la carpeta del proyecto de aplicación, para tenerlos a mano, consultarlos y efectuarles eventuales correcciones. Desde ya que una opción mas prolija y mantenible sería definir un directorio fijo y estándar para alojar los fuentes de CMSIS y drivers, compilándolos como librerías estáticas y enlazando éstas (en una única carpeta estándar) al compilar desde cada proyecto en particular.

En definitiva, el árbol de archivos propuesto para la creación de un proyecto genérico tiene la estructura general de la figura 13 del apéndice §B.

* Según arquitectura de procesador y periféricos del SoC, incluye: vectores de excepciones, código de arranque, manejo de periféricos a bajo nivel, etc. † Contiene directivas para: definición de secciones de memoria, enlace de ciertas librerías, etc. ‡ CMSIS define, tanto para el núcleo procesador como para los periféricos, los vectores de excepción y una forma estandarizada de acceder a los registros. § Problemas de incompatibilidad mayúsculas/minúsculas de nombres de archivos (originales de Windows) invocados sin tener esto en cuenta, unificación de variables de entorno de compilación, etc.

11. Integración de las herramientas en Eclipse, sobre una aplicación de prueba

Se ejemplificará la configuración de un proyecto administrado por Eclipse, con una aplicación minimalista que produce el parpadeo del LED presente en la placa LPCXpresso LPC1768 *target*.

11.1. Creación del proyecto Eclipse

Si bien hubiese sido mas simple entregar un *workspace* completo con el proyecto Eclipse ya preparado para importarlo desde Eclipse, en su lugar haremos los pasos necesarios para la creación del proyecto Eclipse, pues estos mismos (o sus equivalentes) serán los pasos básicos para la creación de un proyecto genérico. Por otra parte, no es el objetivo de este trabajo dar un tutorial sobre el uso y las opciones de Eclipse, así que no nos extenderemos demasiado y en todo caso se recomienda la lectura de la documentación correspondiente.

11.1.1. Preliminares: archivos y directorios

Comenzamos descargando los archivos necesarios en el directorio *home*:

```
$ cd ~
$ wget http://laboratorios.fi.uba.ar/lse/tools/2012-ide/prjt-base-lpc17xx-gcc.tar.gz
$ wget http://laboratorios.fi.uba.ar/lse/tools/2012-ide/test-blinky.tar.gz
```

Abrimos Eclipse.

```
$ eclipse &
```

Creamos un nuevo proyecto desde el menú: *"File -> New -> C Project"* . En la ventana emergente, ponemos un nombre al proyecto, "test-blinky", el tipo de proyecto será *"ARM Cross Target Application"*; seleccionamos *toolchain "Sourcery G++ Lite"*. Presionamos "Next >".

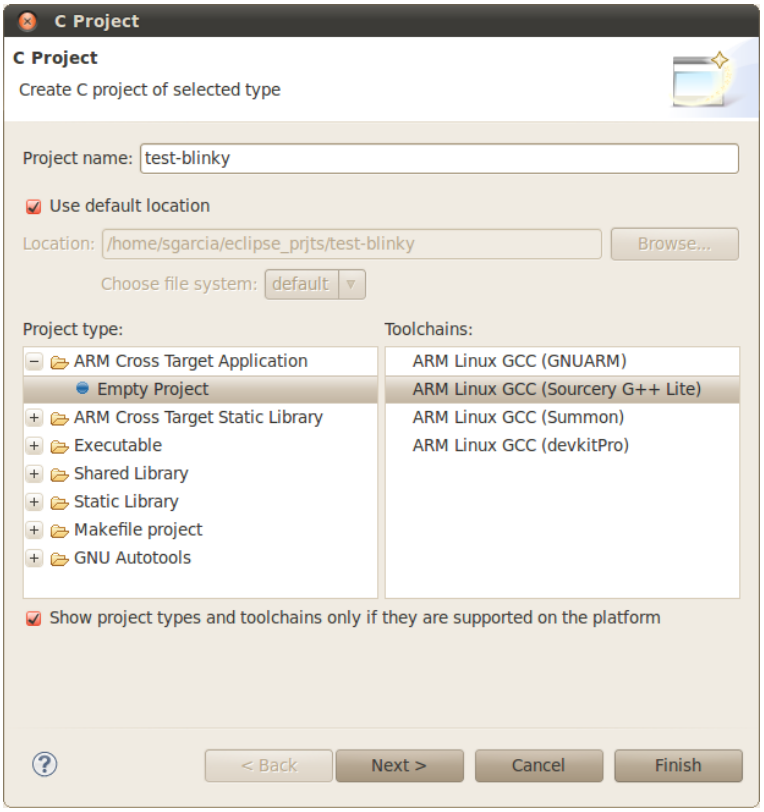

Figura 2: Nuevo proyecto (ventana 1)

En la siguiente pantalla, mantener los perfiles de proyecto que aparecen seleccionados por defecto, *"Debug"* y *"Release"*. Presionamos *"Finish"*.

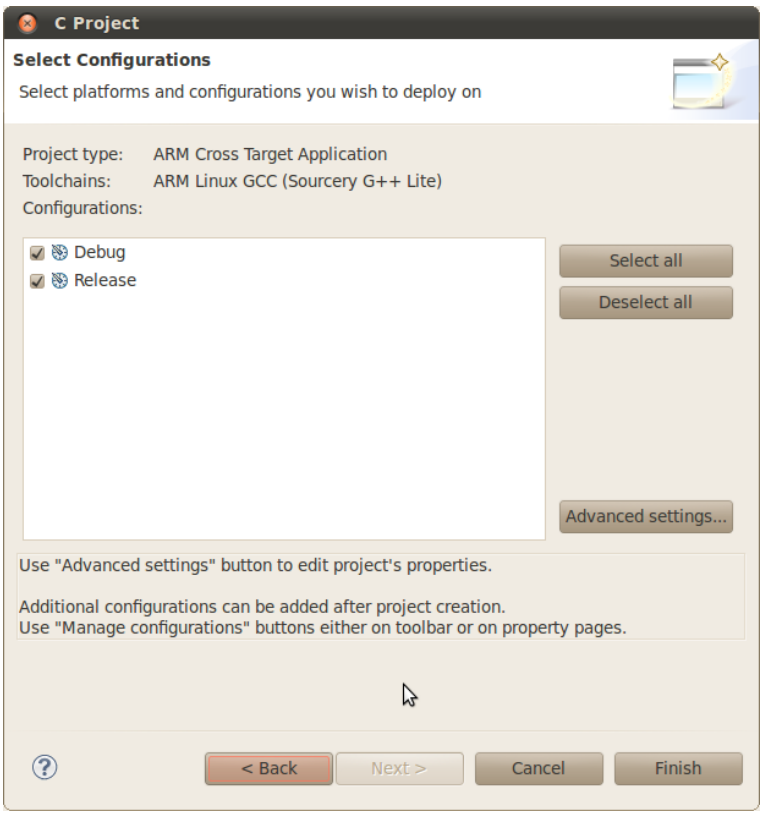

Figura 3: Nuevo proyecto (ventana 2)

Así, queda creado en el directorio *workspace* una carpeta `~/eclipse_prjts/test-blinky` , propia del proyecto, conteniendo inicialmente un par de archivos ocultos con las configuraciones administradas por Eclipse.

Extraemos ahora los archivos básicos provistos (archivo `ldscript_rom_gnu.ld` y carpetas `startup` , `cmsis` y `drivers`) al nuevo directorio de proyecto.

```
1 $ tar -xvzf ~/prjt-base-lpc17xx-gcc.tar.gz -C ~/eclipse_prjts/test-blinky
```

Tenemos entonces, en este punto, establecida la base de archivos para el proyecto (comentada en la sección §10) y el próximo paso es agregar/escribir el código fuente de aplicación.

En el siguiente paso, agregamos el código fuente de nuestro ejemplo. Esta operación creará el subdirectorio `app_src` con los archivos `led_blink.c` y `lpc17xx_libcfg.h` .

```
1 $ tar -xvzf ~/test-blinky.tar.gz -C ~/eclipse_prjts/test-blinky
```

Naturalmente, al escribir una aplicación desde cero en Eclipse, en lugar de la última acción (copiado de archivos de aplicación a la carpeta del proyecto Eclipse), lo que haríamos sería crear dentro de la carpeta `app_src` (i.e., *"File -> New -> Folder"*) los nuevos archivos de texto (i.e.,*"File -> New -> Source File"*, *"File -> New -> Header File"*).

Volviendo al entorno Eclipse, en la solapa *"Project Explorer"* donde aparece la carpeta de proyecto, si hacemos click con el botón derecho del *mouse* en una zona libre y en el menú emergente seleccionamos *Refresh*, deberán aparecer a la vista los archivos y directorios recientemente copiados.

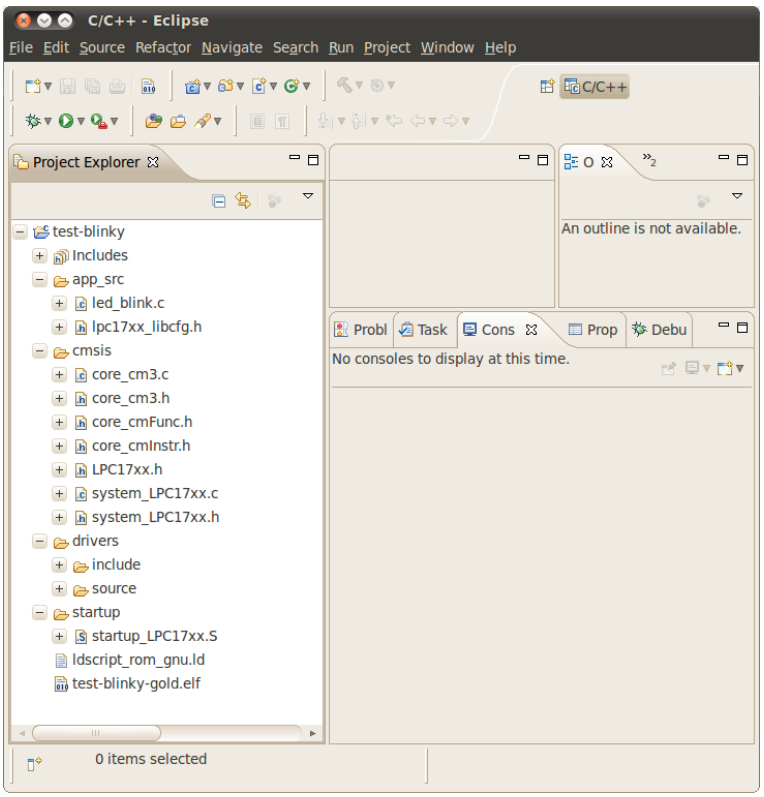

Figura 4: Vista de archivos de proyecto

11.1.2. *Paths*

Hacemos click con el botón derecho del *mouse* sobre la carpeta de proyecto *"test-blinky"*, y seleccionamos: *"Properties"* (de aquí en adelante se hará `[ALT+ENTER]`). En la ventana emergente* *"Properties for. . . "*, seleccionamos:

" *C/C++ General*" -> *"Paths and Symbols"* -> *"Includes"* -> (*"Languages"*) GNU C -> (*"Include directories"*)

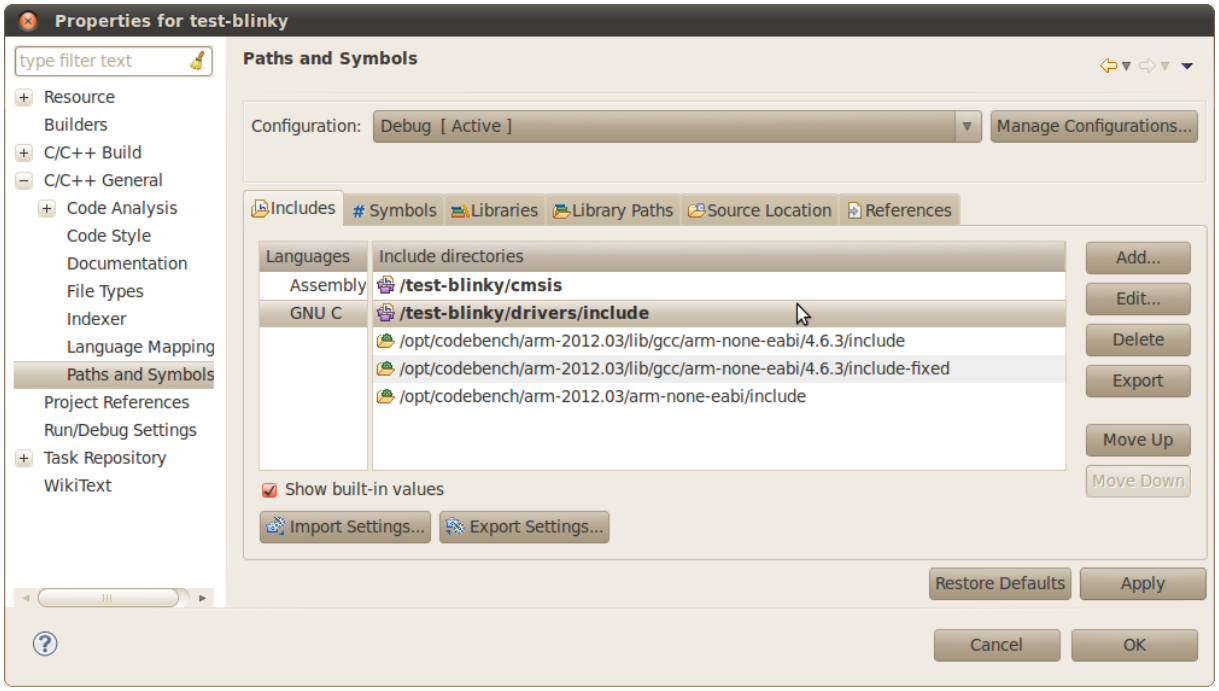

Figura 5: Eclipse - C/C++, configuración de *paths*

* Notar que el perfil de configuraciones activo por defecto es *"Debug"*; trabajaremos con este perfil.

Agregamos uno a uno los siguientes *paths*, presionando en cada caso *"Add..."* -> *"Workspace..."*:

- `/test-blinky/cmsis`
- `/test-blinky/drivers/include`

Presionamos *"Apply"*. En la ventana emergente *"Paths and Symbols"*, presionamos *"Yes"* para reconstruir el índice.

11.1.3. Configuración de arquitectura *target*

En el panel de la izquierda de la ventana *"Properties for..."*, seleccionamos:

"C/C++ Build" -> *"Settings"* -> *"Tool Settings"* -> *"Target Processor"* -> *"Processor:"*

; configuramos *"cortex-m3"* y presionamos *"Apply"*.

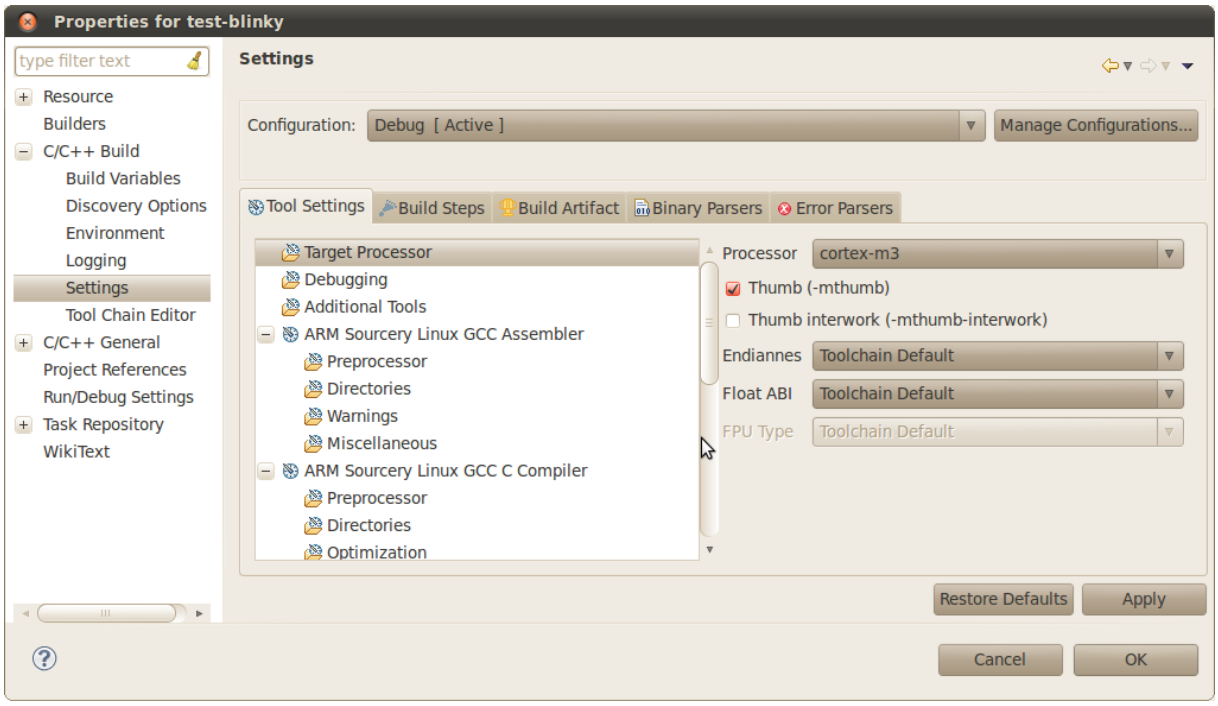

Figura 6: Eclipse - C/C++, configuración de arquitectura *target*

11.1.4. *Linker script*

En el panel de la izquierda de la ventana *"Properties for..."*, seleccionamos:

"C/C++ Build" -> *"Settings"* -> *"Tool Settings"* -> *"ARM Sourcery Linux GCC C Linker"* -> *"General"*

; en el sector derecho, campo *"Script file (-T)"*, presionamos *"Browse"* y buscamos en la carpeta de proyecto el *script* `ldscript_rom_gnu.ld` .

Debajo del campo anterior, removemos la opción *"Do not use standard start files (-nostartfiles)"*, y tildamos la opción *"Remove unused sections (-Xlinker –gc-sections)"* . Presionamos *"Apply"*

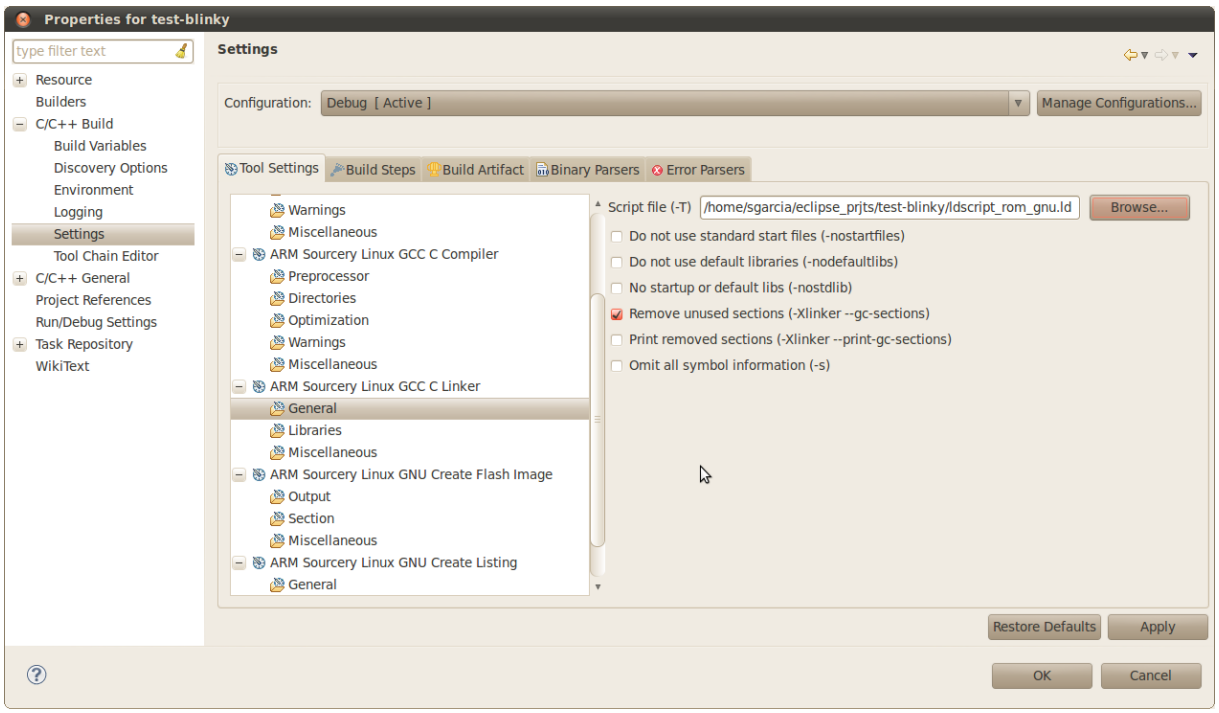

Figura 7: Eclipse - C/C++, configuración de *linker script*

11.1.5. Definición de símbolos

En el panel de la izquierda de la ventana *"Properties for. . . "*, seleccionamos:

"C/C++ General" -> *"Paths and Symbols"* -> *"Symbols"* tab -> (*"Languages"*) GNU C

; a la derecha, presionamos *"Add. . . "* y en la ventana emergente agregamos un nuevo símbolo:

"name:" `__RAM_MODE__` ; *"value:"* `0`

Seleccionamos *"Add to all configurations"* y *"Add to all languages"*. Presionamos *"Apply"*. En la ventana emergente *"Paths and Symbols"*, presionamos *"Yes"* para reconstruir el índice.

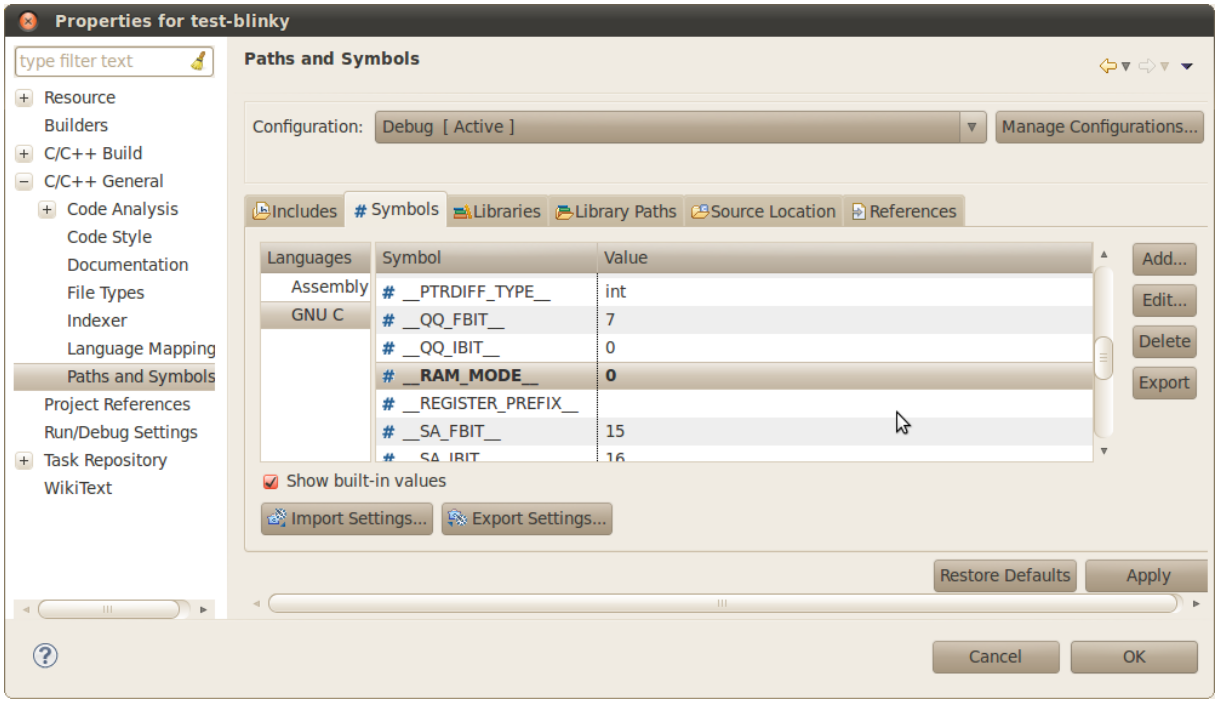

Figura 8: Eclipse - C/C++/assembly, adición de símbolo

Finalmente, presionamos *"OK"* para cerrar la ventana *"Properties for test-blinky"*.

11.2. Compilación

Ya tenemos todo preparado como para hacer un *build*. Para esto presionamos, en la barra de herramientas, el ícono con forma de martillo (o ejecutamos la secuencia de teclado: `[CTRL+B]`). Si todo está en orden, luego de la compilación y el enlace de las unidades, deberá obtenerse el binario de salida `test-blinky.elf` en formato ELF, dentro del subdirectorio *Debug* (creado por Eclipse). Además, con la configuración por defecto, se generarán los archivos *listing* (`test-blinky.lst`), *map* (`test-blinky.map`) y se traducirá el *firmware* a formato HEX Intel (`test-blinky.hex`).

El archivo `makefile` será generado automáticamente por Eclipse, en base a la configuración de opciones en sus menús gráficos. Cabe notar que, en ciertas aplicaciones, puede ser mas productivo o confiable trabajar con un *makefile* elaborado manualmente; esto es más laborioso pero permite mantener un mejor control del *build* del proyecto.

11.3. Eclipse: *Debug* con GDB + OpenOCD

En primer lugar, movemos los archivos auxiliares de OpenOCD previamente descargados, a un nuevo subdirectorio `openocd` dentro del directorio de proyecto.

```
1  $ mkdir ~/eclipse_prjts/test-blinky/openocd
2  $ tar -xvzf ~/openocd-scripts.tar.gz -C ~/eclipse_prjts/test-blinky/openocd
```

Resaltamos que entre ellos, además de los archivos de configuración ya mencionados en la sección §7.2, se cuenta con un *bash script* `openocd.sh` que facilitará el lanzamiento automático de la aplicación `openocd` desde Eclipse:

```
1 #!/bin/bash
2 CFGNAME=‘basename $1‘
3 RUNCMD="openocd -f $CFGNAME"
4 cd ‘dirname $1‘
5 xterm -fg green -bg black -geometry 200x40-0-0 -e $RUNCMD &
```

11.3.1. Configuración de herramienta externa OpenOCD

En Eclipse, creamos una nueva configuración de herramienta externa. Seleccionamos:

"Run" -> "External Tools" -> "External Tool Configurations..."

Hacemos doble click en *"Program"* y aparece un ítem *"New configuration"*. Completamos el campo *"Name:"* con: *"OpenOCD"*. En la solapa *"Main"*, completamos los campos:

- *"Location:"* `${workspace_loc:/test-blinky/openocd/openocd.sh}`
- *"Arguments:"* `${workspace_loc:/test-blinky/openocd/openocd.cfg}`

Verificamos que todo el HW involucrado se encuentra conectado y alimentado, y presionamos *"Run"*. Entonces, arrancará `openocd` en una nueva ventana terminal (útil para su monitoreo).

11.3.2. Configuración de *debug* GDB

En Eclipse, creamos una nueva configuración de *debug*. Seleccionamos:

"Run" -> "Debug Configurations..."

Hacemos doble click en *"GDB Hardware Debugging"* y aparece un ítem *"test-blinky Debug"*. Completamos el campo *"Name:"* con: *"test-blinky flash debug"*.

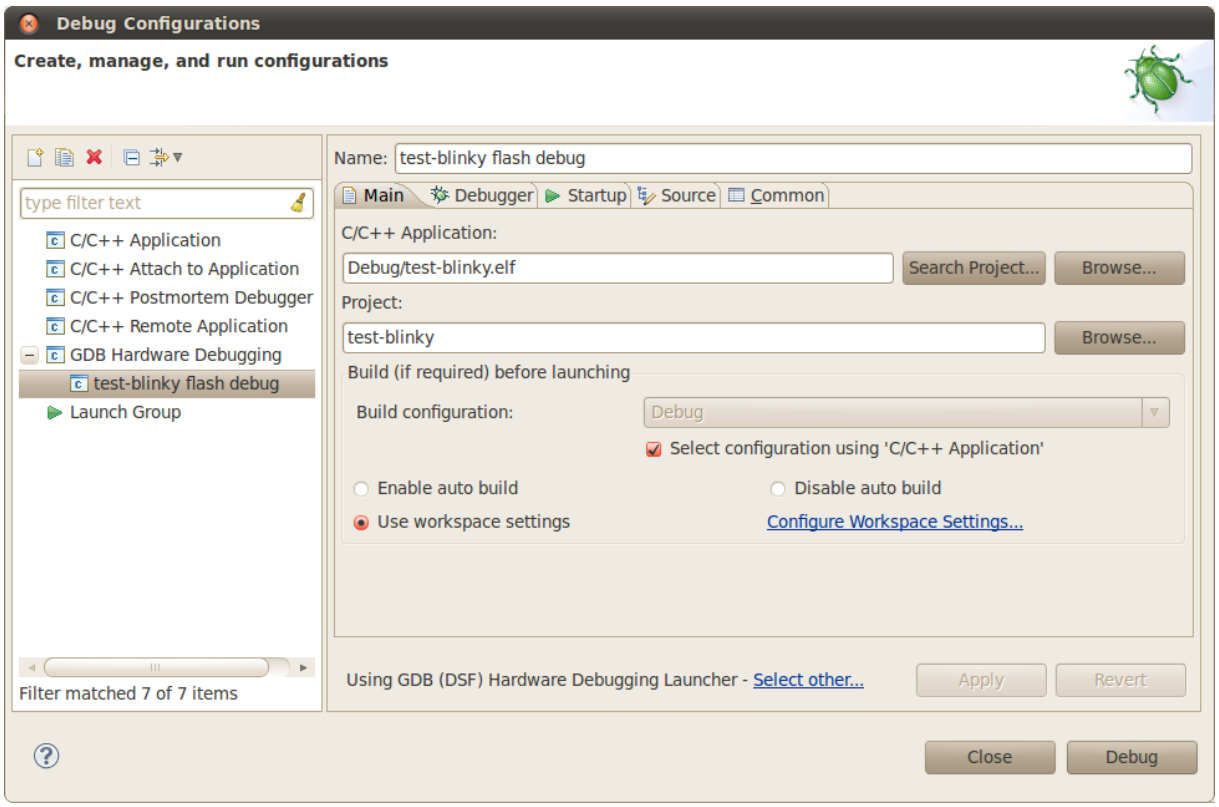

Figura 9: Eclipse - GDB, solapa *Main*

Pasamos a la solapa *"Debugger"* y completamos los campos:

- *"GDB Command:"* `arm-none-eabi-gdb`
- *"JTAG device:"* `generic TCP/IP`
- *"Host name or IP address:"* `localhost`
- *"Port number:"* `3333`

Tildar la opción *"Verbose console mode"*.

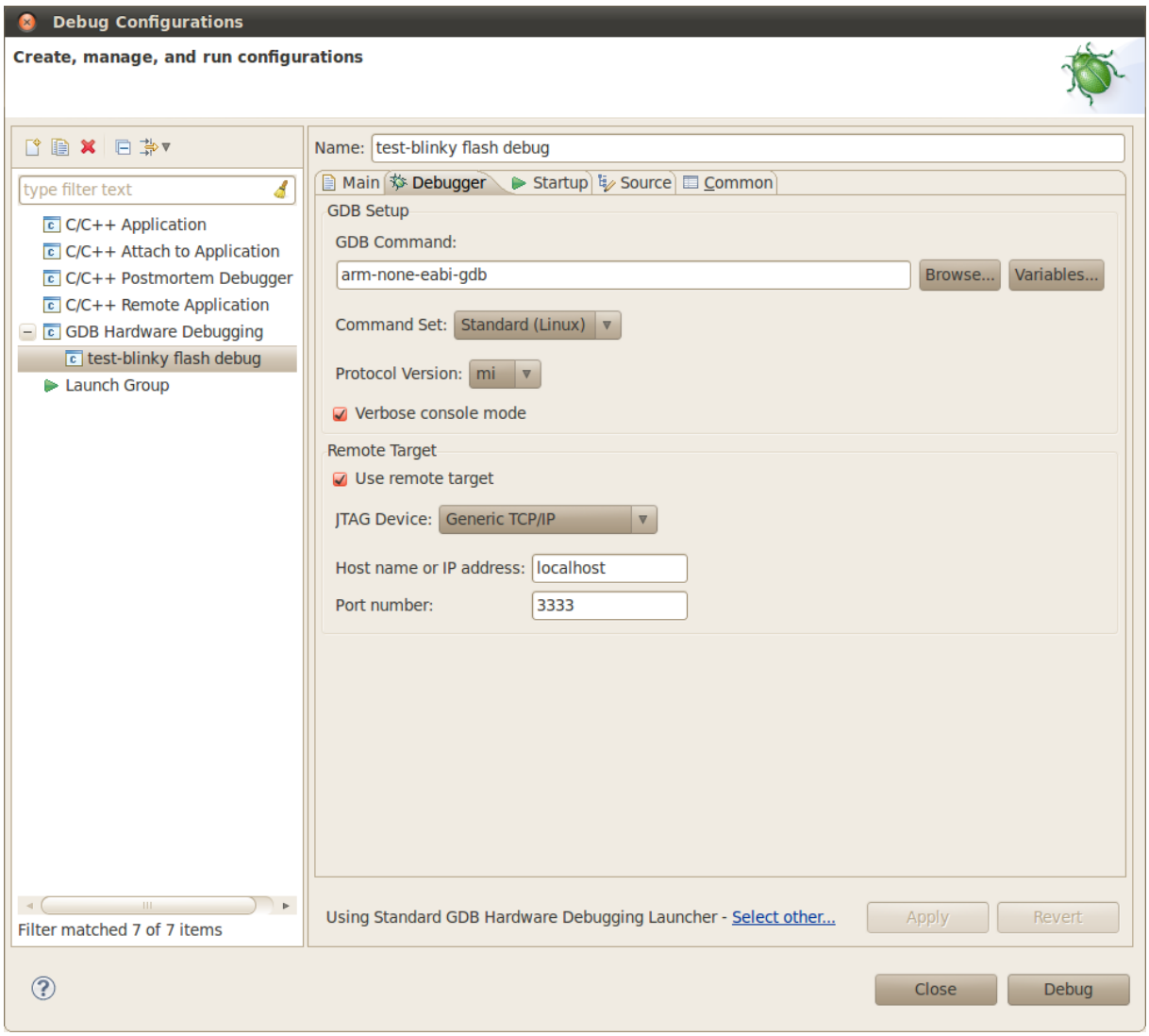

Figura 10: Eclipse: GDB, solapa *Debugger*

Pasamos a la solapa *"Startup"*.
Removemos las opciones *"Reset and Delay"* y *"Halt"*. Completamos en la caja de comandos de inicialización:

```
1 monitor reset init
2 file /home/sgarcia/eclipse_prjts/test-blinky/Debug/test-blinky.elf
3 load
4 monitor soft_reset_halt
```

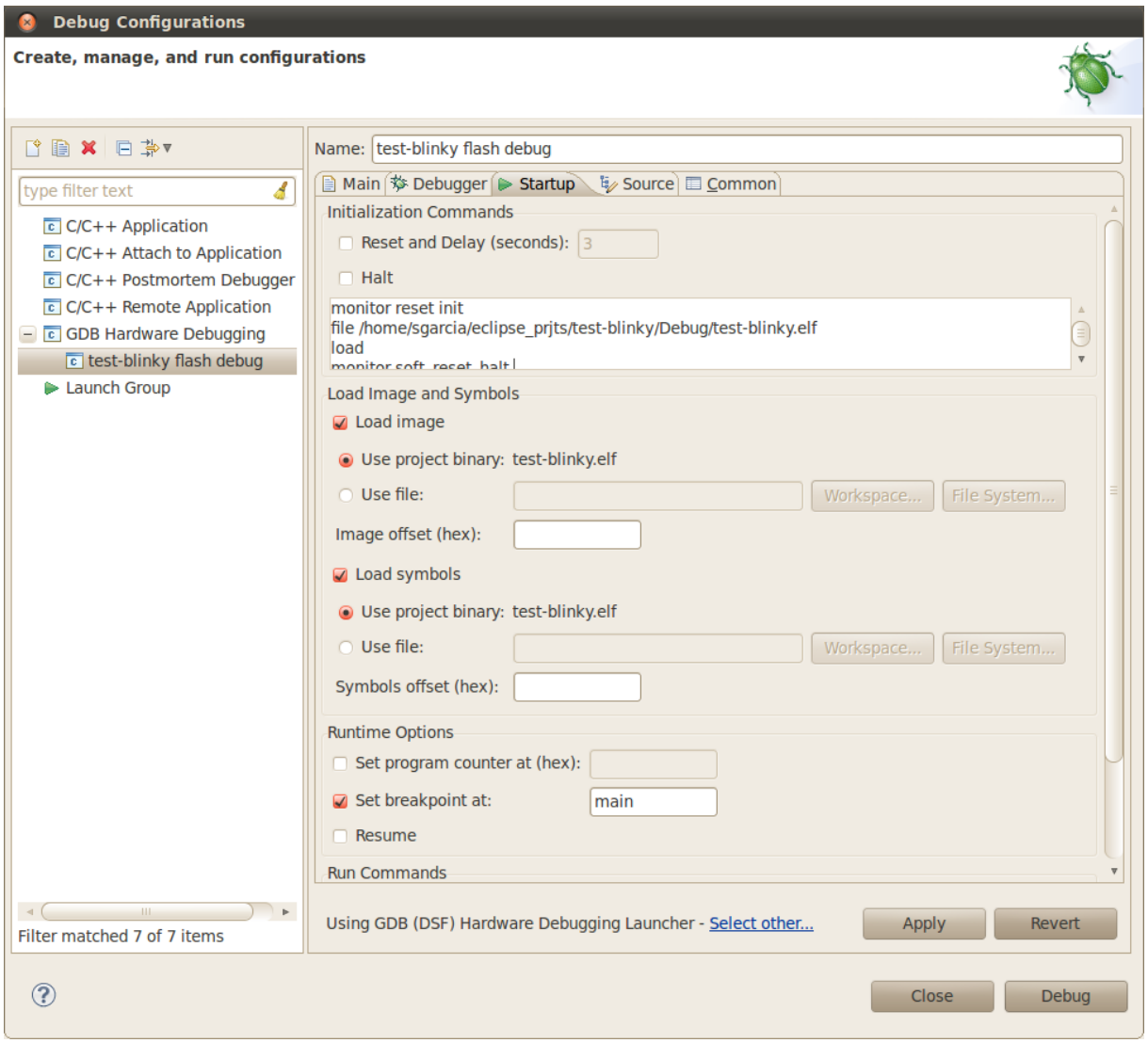

Figura 11: Eclipse - GDB, solapa *Startup* (parte superior)

En la sección de *"Runtime Options"*, seleccionamos *"Set breakpoint at"* y completamos con "`main`" . Completamos en la caja de comandos de ejecución:

```
1 continue
```

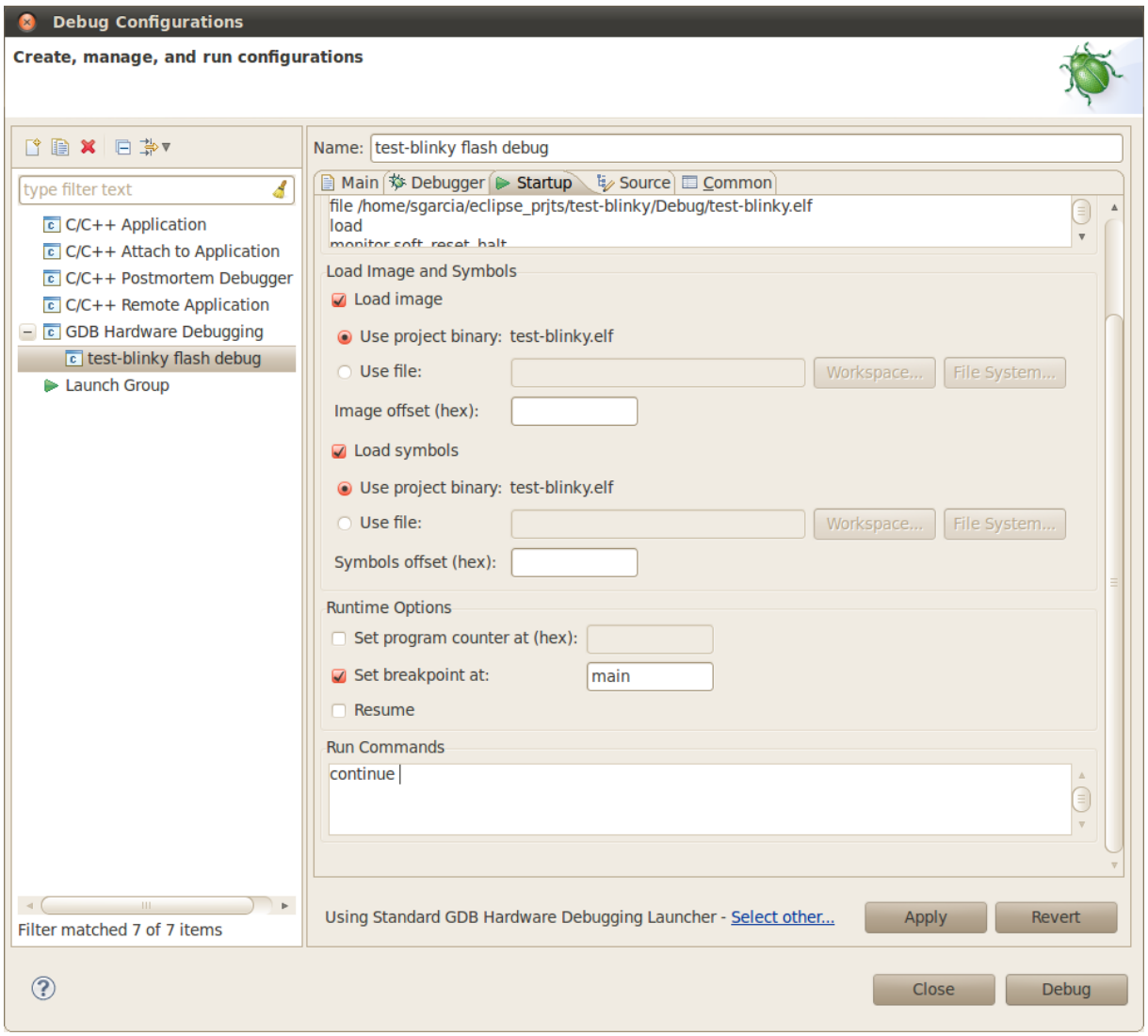

Figura 12: Eclipse - GDB, solapa *Startup* (parte inferior)

Finalmente, presionamos el botón *"Debug"*, que arrancará la aplicación `arm-none-eabi-gdb` , la cual se conectará con `openocd` a través del puerto TCP/IP 3333.

Teniendo ya almacenadas las configuraciones de la herramienta externa OpenOCD y del *debugger* GDB, pueden configurarse los botones de la barra de herramientas de Eclipse *"External tools"* y *"Debug"*, con los perfiles favoritos (*"Organize favorites..."*): "OpenOCD" y "test-blinky flash debug". La secuencia de utilización habitual consiste en:

1. Arrancar OpenOCD (*Run external tools*),
2. Lanzar GDB (al pasar, en Eclipse, de modo *build* a modo *debug*).

11.3.3. Entorno de *debugging*

Luego de lanzar GDB en el último paso, se abrirá una solapa global *"Debug"* (ocultando la solapa global *"C/C++"*), con una serie de ventanas, entre ellas: opciones de ejecución, vista de variables, vista de *breakpoints*, vista de registros, vista de *outline*, consola de comandos.

Este "pasaje a modo *debug*" es algo lento; tener presente que se está cargando el programa en memoria *flash* del microcontrolador (hecho que puede verificarse en la ventana terminal de `openocd`). Para hacer mas ágil este paso y, a la vez, evitar el desgaste innecesario de la memoria *flash* durante los ciclos de *debugging*, puede crearse otro perfil de configuración GDB, donde se cargue el programa a memoria SRAM del microcontrolador. Tener en cuenta que para esto, además, hará falta realizar el *linking* de la aplicación con un *linker script* que mapee todas las secciones a SRAM, y actualizar la variable de entorno correspondiente.

Las operaciones básicas son las habituales de un *debugger*: ejecución, suspensión, paso-a-paso *step into*/*step over*. Haciendo doble click sobre el margen izquierdo del código fuente podemos configurar

breakpoints simples. En la vista de variables, con el menú contextual del botón derecho del *mouse*, podemos agregar *watchpoints* de lectura o escritura, ingresando una expresión a evaluar.

En la consola de *debugging*, podemos interactuar con GDB mediante sus comandos, y también pasarle comandos (antepuestos con el comando GDB `monitor`) a OpenOCD.

12. Cuestiones conocidas

Durante el *debug*, en consola de Eclipse aparece frecuentemente el siguiente mensaje:
"warning: RMT ERROR : failed to get remote thread list.".
Esto pareció resolverse cuando se cambió la interfaz de *debug* de DSF a CDI, pero al configurar la consola GDB en modo verboso, vuelven a aparecer estos mensajes.

Apéndices

A. *Scripts* para configuración de OpenOCD

Documentamos como referencia los *scripts* de comandos utilizados en este trabajo. Para mayor información sobre los comandos, referirse a la guía del usuario de OpenOCD.

A.1. Archivo openocd.cfg

En la primera sección, se llama al *script* de configuración de la herramienta de *hardware*, en nuestro caso el dispositivo FTHL.

En el segundo grupo de comandos, invocamos al *script* de configuración del microcontrolador *target* LPC1768. Notar que, como paso previo, cargamos valores en determinadas variables que utilizará ese *script*.

Finalmente, se le informa a GDB sobre el mapa de memoria del *target*, y se habilita la programación de la memoria *flash*.

```
# [openocd.cfg] OpenOCD configuration script
# 2012-07 Sebastian Garcia
# Hardware:
# - FTHL USB-JTAG dongle
# - LPCXpresso LPC1768 (target side) board
#

# ("3": max. verbosity level)
debug_level 3

# ----------------------
# 1. Dongle configuration
#
source [find fthl.cfg]

# ---------------------------------
# 2. Target processor configuration
#
set CHIPNAME lpc1768
# CCLK: "LPCXpresso LPC1768" default SoC System Clock: 100MHz
set CCLK 100000
# CPUTAPID: JTAG IDCODE register
set CPUTAPID 0x4ba00477
source [find target/lpc1768.cfg]

# ---------------------------------
# 3. Other configs
#
# Backup: remember default values
#telnet_port 4444
#gdb_port 3333

gdb_memory_map enable
gdb_flash_program enable
init
```

A.2. Archivo fthl.cfg

En este conjunto de comandos se configura el tipo de adaptador (familia de chips FT2232), el *layout* de interconexión del adaptador (compatible con Oocdlink), los números VID/PID (USB) pertenecientes al chip utilizado, y la velocidad de acceso JTAG (frecuencia de señal TCK).

Además, en nuestro HW no utilizamos la señal (JTAG) nTRST, por lo tanto configuramos el uso del *reset* de sistema.

```
# [fthl.cfg] OpenOCD configuration script for Emtech's "FTHL"
# (FT2232H-based, Joern Kaipf's OOCDLink compatible, USB-JTAG I/F)
# 2012-07 Sebastian Garcia
#
interface ft2232
ft2232_device_desc "Dual RS232-HS"
ft2232_layout oocdlink
ft2232_vid_pid 0x0403 0x6010
reset_config srst_only
adapter_khz 10
```

A.3. Archivo lpc1768.cfg

Aquí utilizamos el archivo provisto por `openocd` en el directorio `~/openocd-0.5.0/tcl/target` , sin modificación alguna. Igualmente documentamos su contenido, para que quede como referencia para quienes utilicen otras versiones de esta aplicación.

```
# NXP LPC1768 Cortex-M3 with 512kB Flash and 32kB+32kB Local On-Chip SRAM,

# LPC17xx chips support both JTAG and SWD transports.
# Adapt based on what transport is active.
source [find target/swj-dp.tcl]

if { [info exists CHIPNAME] } {
  set  _CHIPNAME $CHIPNAME
} else {
  set  _CHIPNAME lpc1768
}

# After reset the chip is clocked by the ~4MHz internal RC oscillator.
# When board-specific code (reset-init handler or device firmware)
# configures another oscillator and/or PLL0, set CCLK to match; if
# you don't, then flash erase and write operations may misbehave.
# (The ROM code doing those updates cares about core clock speed...)
#
# CCLK is the core clock frequency in KHz
if { [info exists CCLK ] } {
  set _CCLK $CCLK
} else {
  set _CCLK 4000
}
if { [info exists CPUTAPID ] } {
  set _CPUTAPID $CPUTAPID
} else {
  set _CPUTAPID 0x4ba00477
}

#delays on reset lines
adapter_nsrst_delay 200
jtag_ntrst_delay 200

#jtag newtap $_CHIPNAME cpu -irlen 4 -expected-id $_CPUTAPID
swj_newdap $_CHIPNAME cpu -irlen 4 -expected-id $_CPUTAPID

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME cortex_m3 -chain-position $_TARGETNAME

```

```
# LPC1768 has 32kB of SRAM In the ARMv7-M "Code" area (at 0x10000000)
# and 32K more on AHB, in the ARMv7-M "SRAM" area, (at 0x2007c000).
$_TARGETNAME configure -work-area-phys 0x10000000 -work-area-size 0x8000

# LPC1768 has 512kB of flash memory, managed by ROM code (including a
# boot loader which verifies the flash exception table's checksum).
# flash bank <name> lpc2000 <base> <size> 0 0 <target#> <variant> <clock> [calc checksum]
set _FLASHNAME $_CHIPNAME.flash
flash bank $_FLASHNAME lpc2000 0x0 0x80000 0 0 $_TARGETNAME lpc1700 $_CCLK calc_checksum

# Run with *real slow* clock by default since the
# boot rom could have been playing with the PLL, so
# we have no idea what clock the target is running at.
jtag_khz 10


$_TARGETNAME configure -event reset-init {
  # Do not remap 0x0000-0x0020 to anything but the flash (i.e. select
  # "User Flash Mode" where interrupt vectors are _not_ remapped,
  # and reside in flash instead).
  #
  # See Table 612. Memory Mapping Control register (MEMMAP - 0x400F C040) bit description
  # Bit Symbol Value Description Reset
  # value
  # 0 MAP Memory map control. 0
  # 0 Boot mode. A portion of the Boot ROM is mapped to address 0.
  # 1 User mode. The on-chip Flash memory is mapped to address 0.
  # 31:1 - Reserved. The value read from a reserved bit is not defined. NA
  #
  # http://ics.nxp.com/support/documents/microcontrollers/?scope=LPC1768&type=user

  mww 0x400FC040 0x01
}
```

B. Conjunto de archivos base de proyecto

En la figura 13 se resume la organización propuesta de los archivos a utilizarse en un proyecto genérico.

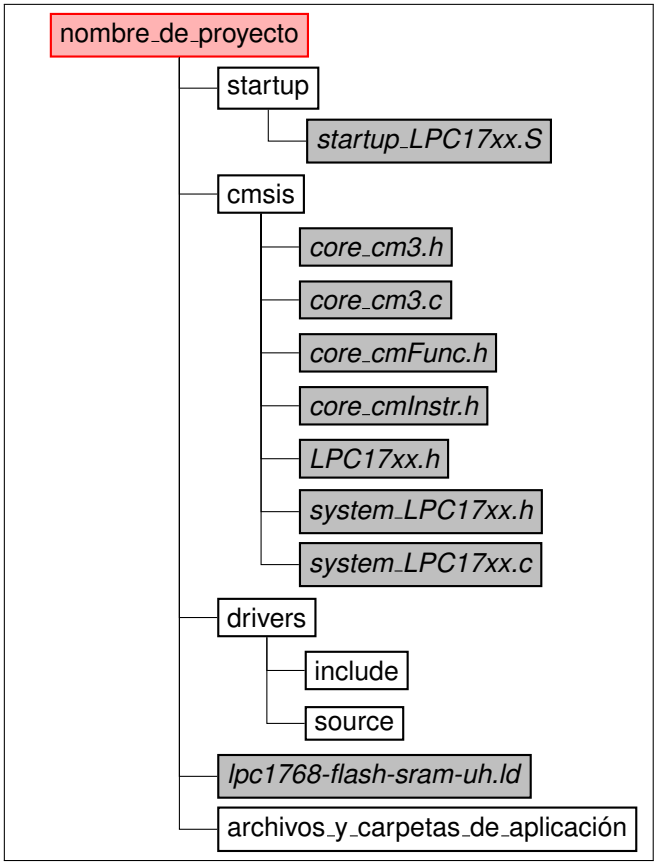

Figura 13: Arbol de archivos de proyecto