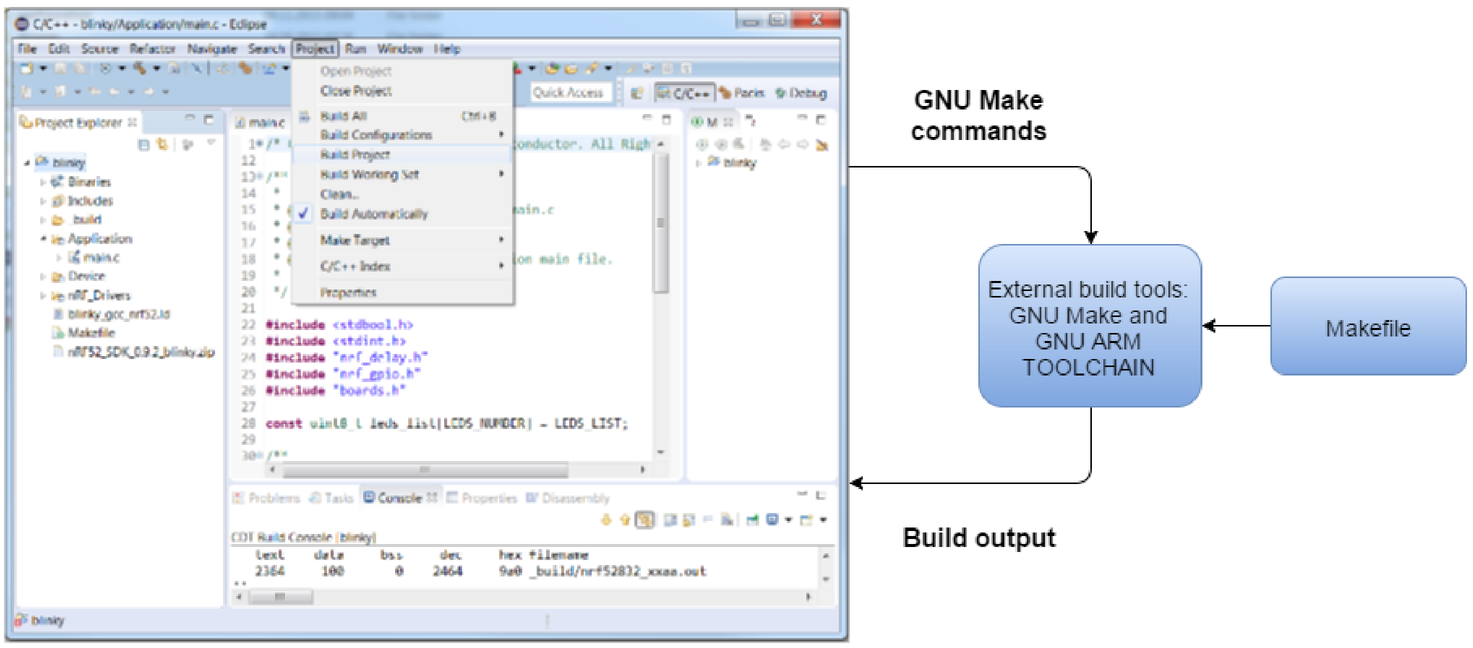
Development with GCC and Eclipse

Vidar Berg 21 Apr 2015

The nRF5x series Software Development Kits (SDK) come with Makefiles for use with the GNU ARM toolchain. This makes it possible to build example projects only using command line tools on your preferred platform (Windows, Linux or OS X). In this tutorial I will go through the necessary steps to integrate these examples with Eclipse in order to get common features associated with IDEs such as jump to declarations, code completion and debugger integration.

The scope of this tutorial will be limited to Makefile managed project in Eclipse to allow the existing SDK Makefiles to be used. A Makefile managed project is as the name implies, controlled by the Makefile and also built outside of the Eclipse environment using external tools, unlike other common IDE setups where the build recipe is a made based on project configurations. This means that the build will not be affected by project settings configured in Eclipse. I.e., adding source files, defines and include paths. So to actually manage your build you need to change the Makefile itself. We will later in the tutorial use the CDT output parser to parse the output of the Makefile to have the symbols and include paths added to the Eclipse project automatically in order to get some integration between the external build and project view in Eclipse as illustrated below.



```
#include <stdbool.h>
#include <stdint.h>
#include "nrf_delay.h"
#include "nrf_gpio.h"
#include "boards.h"

const uint8_t leds_list[LEDS_NUMBER] = LEDS_LIST;
```

```
   text    data     bss     dec     hex filename
   2364     100       0    2464     9a0 _build/nrf52832_xxaa.out
```

The other alternative not covered in this tutorial is Eclipse Managed projects that automatically generates Makefiles based on your project settings and therefore giving a tighter integration between Eclipse and the build process. It is an option. However, main reason for not including it here is because it takes more time to set up, and the end result is about the same in terms of usability.

Before we begin

Before we start setting up Eclipse we need to install some software components required to build source code, and also software to load it to the target device. These components enables us to test the SDK examples from the command line in addition to Eclipse.

GNU toolchain for ARM Cortex-M

GNU toolchain including compiler and GDB debugger can be downloaded using the following link. Download and install the latest version. Then make sure to add the path to your toolchain to your OS PATH environment variable:

```
<path to install directory>/GNU Tools ARM Embedded/4.9 2015q3/bin
```

Adding the path makes it possible to run the toolchain executables from any directory using the terminal. To verify that the path is set correctly, type the following in your terminal:

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

This will return the version of the C compiler if the executable is found in your path.

GNU make

Now with the toolchain installed we can build object files from source code, but to build projects based on makefiles, which can be seen as a recipes for the builds, we need to have GNU make installed on the system.

On windows it can be obtained by installing the *GNU ARM Eclipse Windows Build Tools* package from the GNU ARM Eclipse plug-in project. This package also adds support for shell commands such as rm and mkdir that are used by our Makefiles.

Linux and OS X already have the necessary shell commands, but GNU make may not be a part of the standard distro. Call "make -v" from the terminal to check whether it is installed or not. GNU make would need to be installed if it's not recognized as a command.

GNU make is bundled with Xcode tools if working on OS X.

On Linux it may be different ways to obtain GNU make depending on your distro, if not installed already. On Ubuntu you can get by entering this command:

```
sudo apt-get install build-essential checkinstall
```

Nordic nRF5x SDK

Download SDK 11.0.0 from http://developer.nordicsemi.com/, or SDK 10.0.0 if s110 is to be used on the nRF51.

To build an example in the SDK you first need to set the toolchain path in makefile.windows or makefile.posix depending on platform you are using. That is, the .posix should be edited if your are working on either Linux or OS X. These files are located in

```
<SDK>/components/toolchain/gcc
```

Open the file in a text editor, and make sure that the GNU_INSTALL_ROOT variable is pointing to your *Gnu tools for ARM embedded Processors* install directory.

Correct values for my current setup:

```
GNU_INSTALL_ROOT := $(PROGFILES)/GNU Tools ARM Embedded/4.9 2015q3   // Toolchain path
GNU_VERSION := 4.9.3
GNU_PREFIX := arm-none-eabi
```

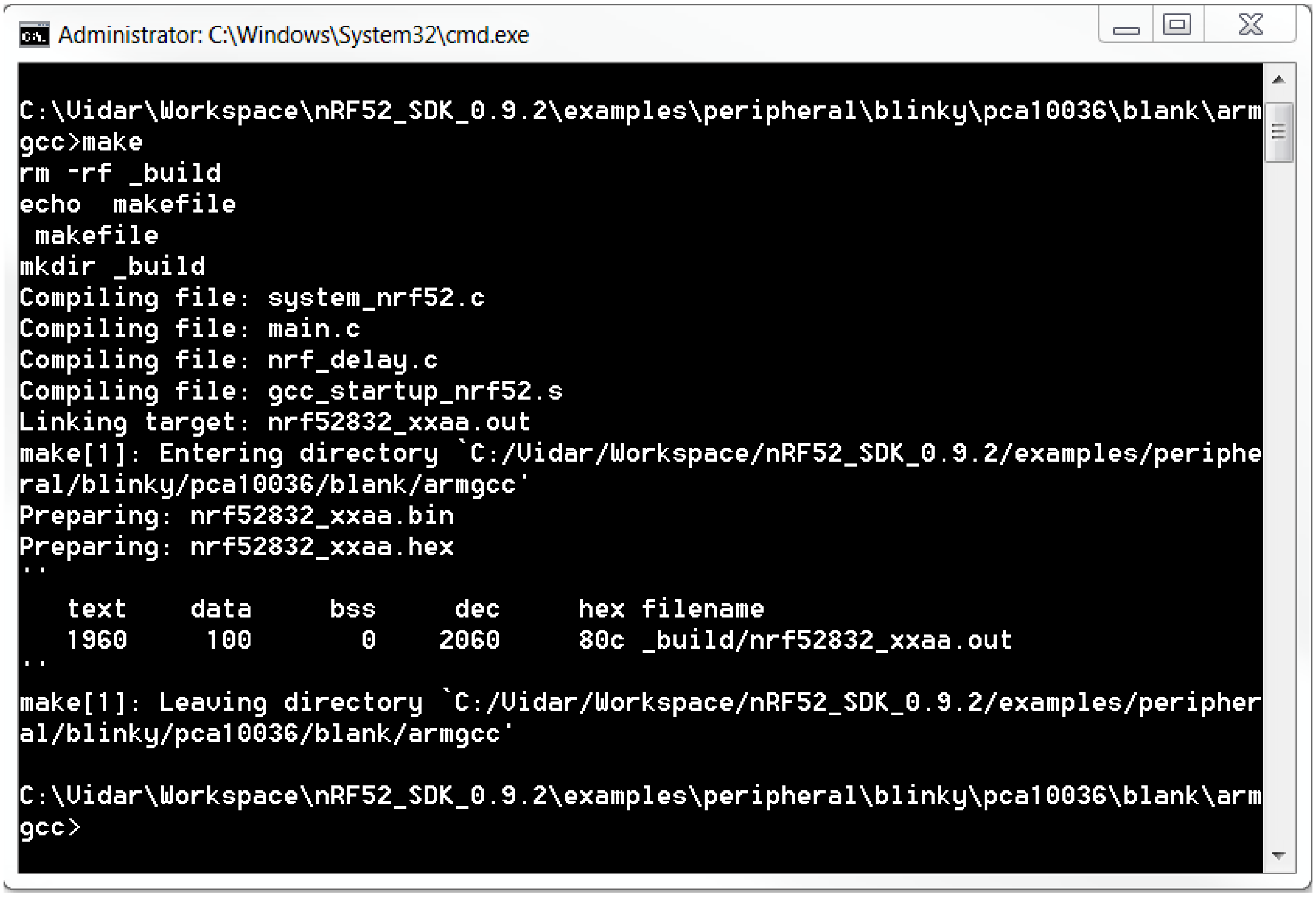
Now you can try to build one of the example projects. Will use the blinky example here to keep it simple:

Open terminal and change directory to

```
<SDK>/examples/peripheral/<board name>/blank/armgcc/
```

Alternatively, open the terminal in that directory. In windows, navigate to the directory in explorer, press 'Alt+D' and type 'cmd' in the address field followed by return.

Type 'make'. GNU Make should start the build using the Makefile and output the result in the _build directory. If everything works you should get the output shown in the screenshot below.



```
C:\Vidar\Workspace\nRF52_SDK_0.9.2\examples\peripheral\blinky\pca10036\blank\arm
gcc>make
rm -rf _build
echo  makefile
 makefile
mkdir _build
Compiling file: system_nrf52.c
Compiling file: main.c
Compiling file: nrf_delay.c
Compiling file: gcc_startup_nrf52.s
Linking target: nrf52832_xxaa.out
make[1]: Entering directory `C:/Vidar/Workspace/nRF52_SDK_0.9.2/examples/periphe
ral/blinky/pca10036/blank/armgcc'
Preparing: nrf52832_xxaa.bin
Preparing: nrf52832_xxaa.hex
''
   text    data     bss     dec     hex filename
   1960     100       0    2060     80c _build/nrf52832_xxaa.out
''
make[1]: Leaving directory `C:/Vidar/Workspace/nRF52_SDK_0.9.2/examples/peripher
al/blinky/pca10036/blank/armgcc'

C:\Vidar\Workspace\nRF52_SDK_0.9.2\examples\peripheral\blinky\pca10036\blank\arm
gcc>
```

If you instead get an error saying something like "the sysem cannot find the files specified" it typically means that the GNU toolchain path is set incorrectly. Verify the path in makefile.windows/posix if you get this.

Errors like "make: No rule to make target '_build/Nordic', needed by `nrf51422_xxac". Stop can also be an indication that build tools are unable to handle whitespaces in the path. E.g, c:/nordic semicondtor/SDK/... Workaround is to remove any whitespaces you may have with underscores.

nrfjprog - Programming Tool

nrfjprog bundled in the nRF5x-Command-Line-Tools is a command line tool for loading FW images to target via the debug interface (SWD). This tool has now been ported to all three platforms; Linux, Windows, and osx

Now you can try to load the FW image built in the previous step to your target.

nrfjprog:

```
/* Optional: erase target if not already blank */
<SDK>/examples/peripheral/<board name>/blank/armgcc/>nrfjprog --family <nRF51/52> -e
/* Load FW image to target */
<SDK>/examples/peripheral/<board name>/blank/armgcc/>nrfjprog --family <nRF51/52> --program _bu
/* Reset and run */
<SDK>/examples/peripheral/<board name>/blank/armgcc/>nrfjprog --family <nRF51/52> -r
```

Nrfjprog is tailored for the nRF5 series devices, but it is also possible to use Jlink commander for the same in case its needed:

```
/* Open Jlink Commander from terminal in _build directory */
JLinkExe (jlink on windows) -device <nRF51/nRF52>
> erase // Optional: erase target if not already blank
> loadfile <name>.hex // loads FW
> r // Reset and halt
> g // Run
> q //  Exit
```

You should see blinking LEDs on your kit once you have completed the steps above. Now we are ready for the next step which is to Download Eclipse and start configuring it.

Eclipse Neon IDE for C/C++ Developers

Download the latest Neon C/C++ developers package for your OS at https://www.eclipse.org/downloads/packages/release/Neon

Setting up Eclipse the first time

The following steps shows the general settings configurations of Eclipse that are not project specific.

Install GNU ARM Eclipse plug-in

The GNU ARM Eclipse plug-in simplifies debugger integration in Eclipse significantly in addition to adding support for the GCC ARM compiler so paths to the standard toolchain libraries are set automatically.

Instruction on how to install it are provided on their website: http://gnuarmeclipse.github.io/plugins/install/

Required packages are:

- GNU ARM C/C++ Cross Compiler
- GNU ARM C/C++ Packs
- GNU ARM C/C++ J-link Debugging

The other packages are optional.

Install device family pack for the nRF5x series (optional)

The device family pack includes CMSIS System View Description, which is basically is a memory map of all peripheral registers on the device. Although it is not required to have installed it can be quite useful for debugging purposes; it allows you to easily read and write to particular peripherals through a built in peripheral viewer (part of GNU ARM Eclipse Plug-in) without having to look up the the addresses and register descriptions in the datasheet first.

Steps to install:

- Select the Window item from the menu bar, and enter perspective -> open perspective -> other -> Packs.
- Click the refresh button in the top right corner of the window that got opened. This will fetch all packs from the repositories.
- Select Nordic Semiconductor in the list of vendors, and install the latest version of Device family pack.

Configure environment

Here we will set the global Eclipse configurations for development on nRF5x. Note that these configurations are not specific for a particular project, and will only be done the first time Eclipse is installed.

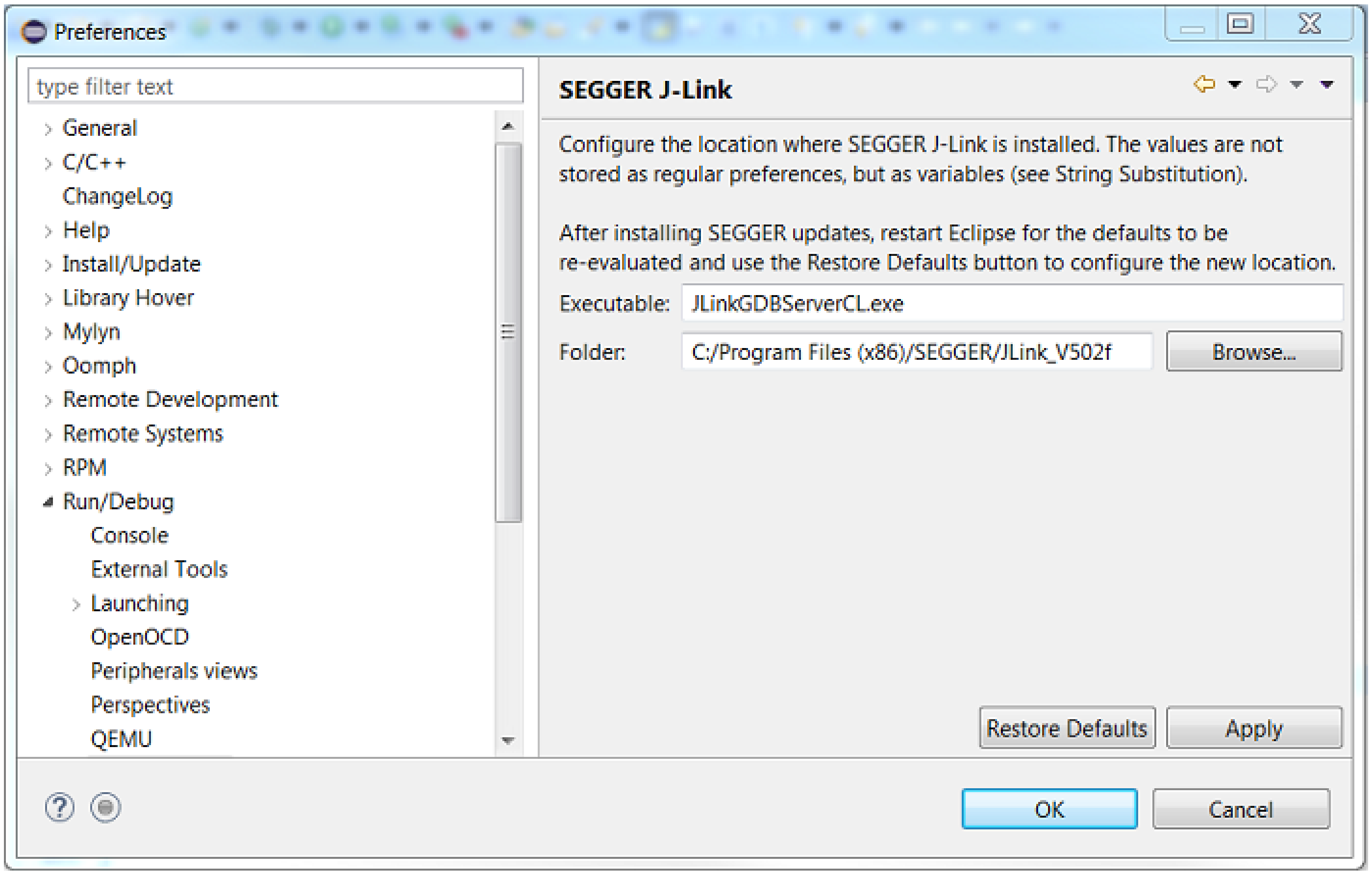
First, click on the window item in the menubar and select preferences. Then go to C/C++->Build->Global Tools path. Check if the paths to your Build tools and Toolchain are set, otherwise add the paths where you installed these earlier. On this setup the following paths were used:

```
C:\Program Files\GNU ARM Eclipse\Build Tools\2.7-201610281058\bin
C:\Program Files (x86)\GNU Tools ARM Embedded\5.2 2015q4\bin
```

On OS X or Linux the following shell command can be used to find the location of make:

```
$ which make
```

Then, enter Run/Debug, click on SEGGER J-link, and make sure that it is pointing to your latest version of the Segger software. In my case it was C:/Program Files (x86)/SEGGER/JLink_V502f. This is path will be needed later when debugging with the installed J-link debug plugin.



Configure the location where SEGGER J-Link is installed. The values are not stored as regular preferences, but as variables (see String Substitution).

After installing SEGGER updates, restart Eclipse for the defaults to be re-evaluated and use the Restore Defaults button to configure the new location.

Executable: JLinkGDBServerCL.exe

Folder: C:/Program Files (x86)/SEGGER/JLink_V502f

Import existing Eclipse project to workspace

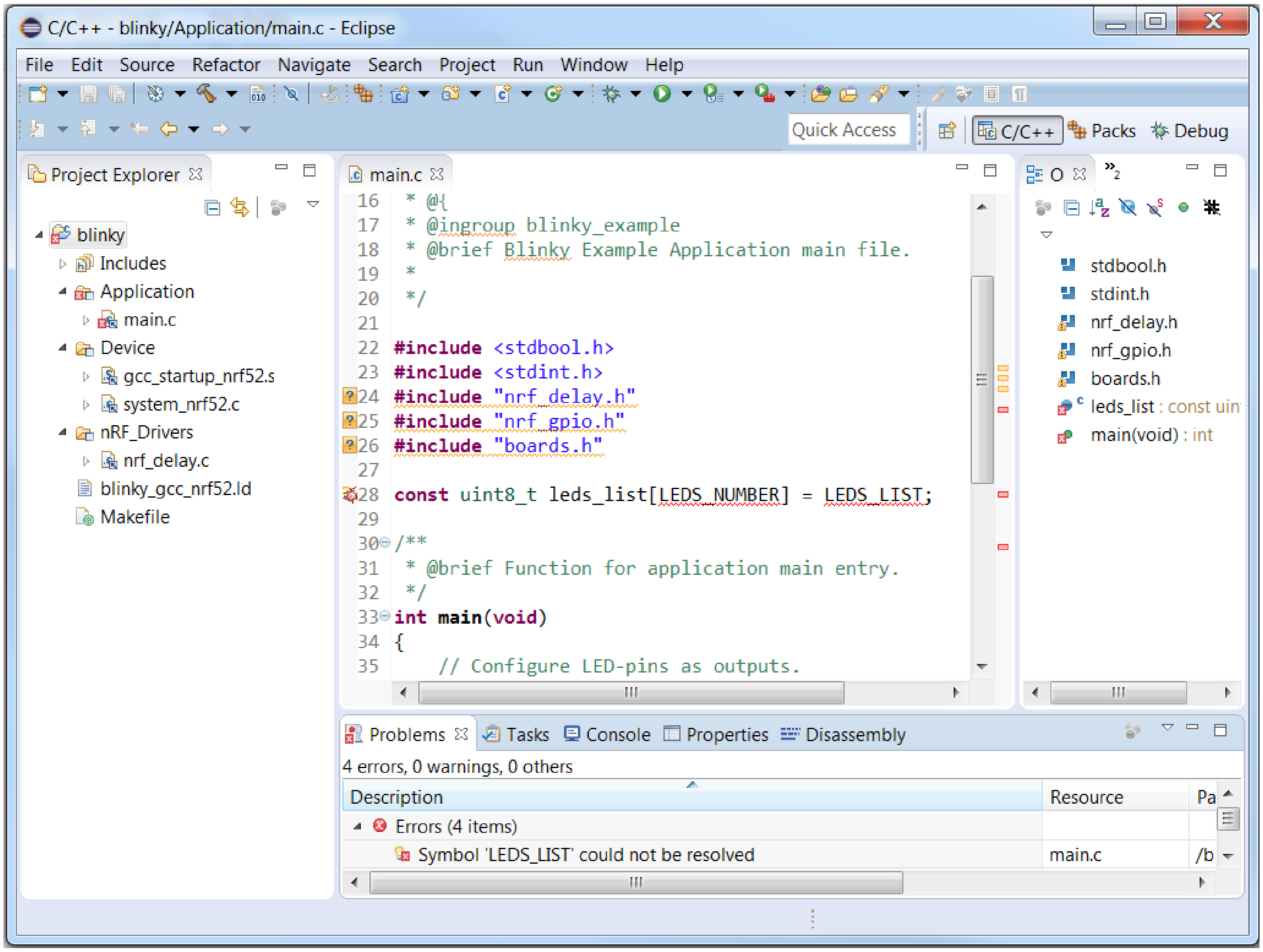
I have attached the ble_app_hrs and blinky example that can be used as a starting point, see the last section for a list of attachments. You may want to start with the blinky example first to keep it simple at first.

I guess some of you would prefer to start by integrating your own project right away instead as described in the next section. However, I would highly recommend trying one of these examples first in order to get something working first. Then if you experience problems with your project later on, you would have a working example compare against.

- Download the project you wish to use from the attachment list, extract it, copy and replace with the content into the existing ARMGCC folder in the respective SDK project. E.g., \examples\peripheral\blinky"board name"\blank\armgcc for the blinky example. Note that the directory structure **must** be kept. Otherwise it will break the paths. Also make sure that your SDK copy has the same version number as the project.

- Open Eclipse.
- Enter file->import->General -> Existing Projects into Workspace.
- Browse ARMGCC folder in your SDK
- Make sure to select the project.
- Click finish button.
- The project should now be available in the project explorer including the linked sources.

Now you will see your imported project in the Project explorer window as shown in screenshot below. Notice the errors. We will get to that in the *Enable auto discovery of symbols, include paths and compiler settings* section

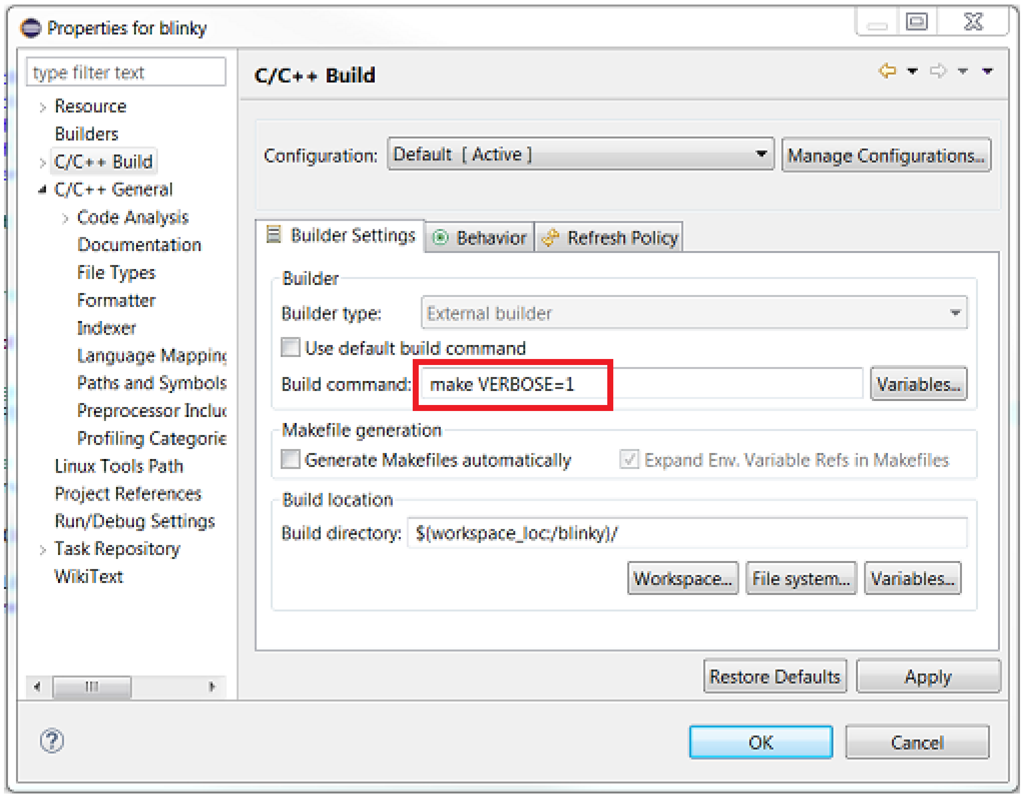


```
 * @{
 * @ingroup blinky_example
 * @brief Blinky Example Application main file.
 *
 */

#include <stdbool.h>
#include <stdint.h>
#include "nrf_delay.h"
#include "nrf_gpio.h"
#include "boards.h"

const uint8_t leds_list[LEDS_NUMBER] = LEDS_LIST;

/**
 * @brief Function for application main entry.
 */
int main(void)
{
    // Configure LED-pins as outputs.
```

4 errors, 0 warnings, 0 others

Description	Resource	Pa
Errors (4 items)		
Symbol 'LEDS_LIST' could not be resolved	main.c	/b

For now we just want to make sure that we can build the example as we did in the command line earlier. Right click on the project folder in *project explorer*, click properties, and click on the C/C++ item in the list. Then replace the default build command with *make VERBOSE=1*. Setting the VERBOSE variable to '1' makes the Makefile print command line options used in the build. In other words increase the verbosity level. This will later be used to parse compiler options to your Eclipse project.



C/C++ Build

Configuration: Default [Active]

Builder

Builder type: External builder

☐ Use default build command

Build command: make VERBOSE=1

Makefile generation

☐ Generate Makefiles automatically ✓ Expand Env. Variable Refs in Makefiles

Build location

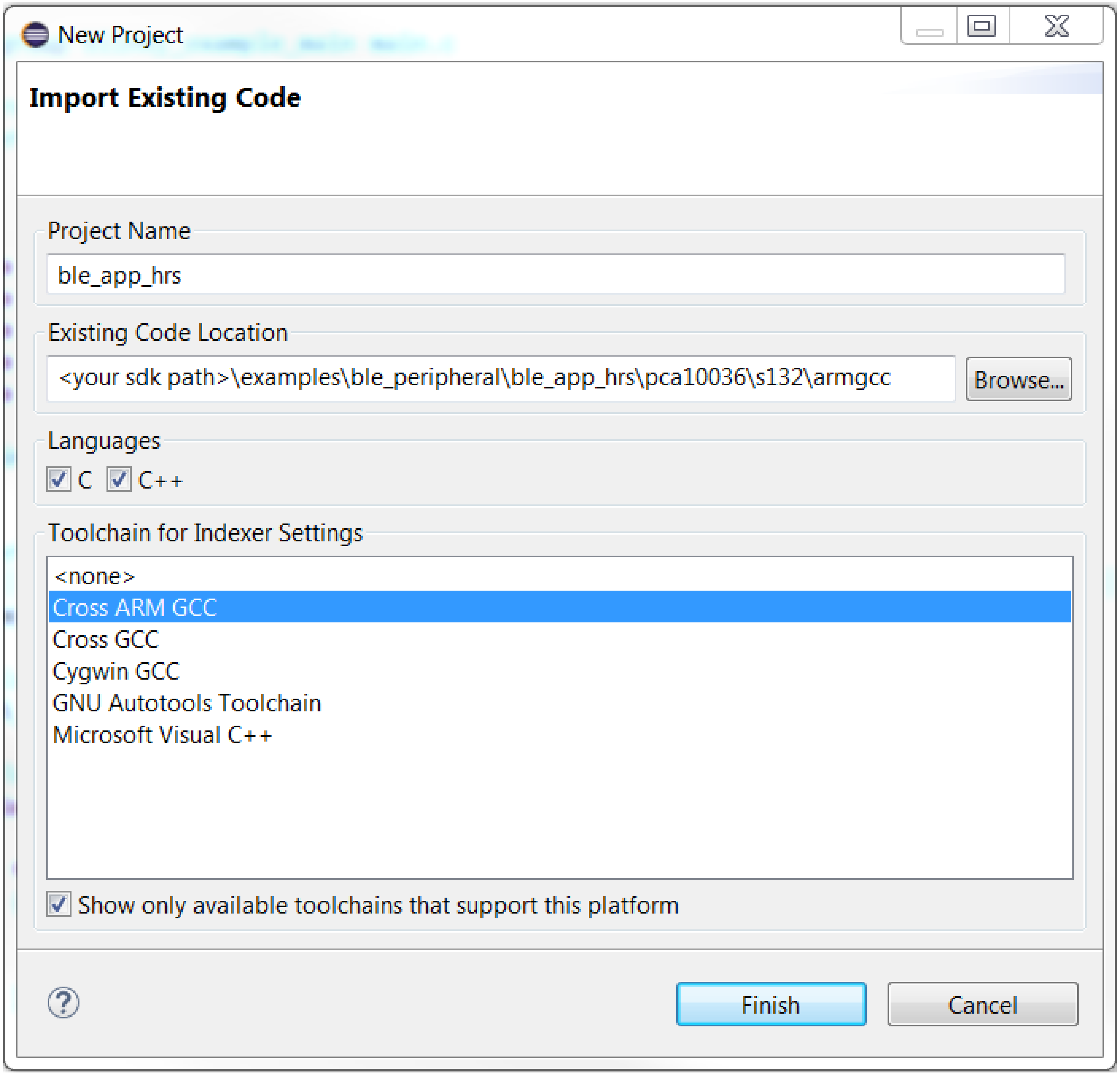
Build directory: ${workspace_loc:/blinky}/

Right click on project again and click on *Build Project*. Hopefully you will see the build output in the console window, same as when you built the example via the terminal. Note that actual build errors will be shown in the log at this point. The errors we see in Eclipse are due to the fact that Eclipse does not now what include paths to use nor symbols. E.g., *-DBOARD_PCA10036* which is used to determine what board support file to include, but you can see that it is used in the build log.

Create a new Eclipse project

This section is for those who wants to integrate an existing project into Eclipse, or just want to create a new one, but not use any of the attached examples as templates. These instructions assumes the Makefile configurations used in SDK 7.x.x. and later.

Start Eclipse, enter file -> new Makefile project with existing code in the menu bar. Name your project and browse the directory of your Makefile. Click *finish* once you are done.


New Project

Import Existing Code

Project Name: ble_app_hrs

Existing Code Location: <your sdk path>\examples\ble_peripheral\ble_app_hrs\pca10036\s132\armgcc

Languages: [✓] C [✓] C++

Toolchain for Indexer Settings:
- <none>
- Cross ARM GCC [✓]
- Cross GCC
- Cygwin GCC
- GNU Autotools Toolchain
- Microsoft Visual C++

[✓] Show only available toolchains that support this platform

Open and edit the SDK makefile to support debugging of your code. Locate the CFLAGS variables, and change '-O3' to '-O0' for debugging to produce expected results (stack variables loaded directly to CPU registers,etc). Then add '-g3' to CFLAGS to include debug symbols in the .out file (GNU manual).

Configure the build settings and build the project, see the instructions from the *Import existing Eclipse project to workspace* section above and verify that there are no build errors before moving on to the next step.

Now you can start linking the source files into your project viewer, but you might want to add some virtual folders to obtain a more clear project view first.

Create virtual folders:

- Right click on the project folder and enter New->folder.
- Set folder name.
- Click Advanced and choose virtual folder.
- Click finish.

Repeat above steps until you have the folders you want. The naming of the folders should reflect the types of the source files you intend to link to. In the blinky it was sufficient to have Application, Device and nRF_Drivers, same as the Keil example for blinky, but, for larger projects with more source files you should add more folders. E.g., ble_app_hrs: Application, Board Support, Device, nRF_BLE, nRF_Drivers, nRF_Libraries and nRF_Softdevice.

Now you can start to link the source files to the respective folders. The source files to include are listed in your Makefile->C_SOURCE_FILES, and ASM_SOURCE_FILES. Source files are then linked to the folder by right clicking on any of the virtual folders and clicking on import, General->File System and browse the source files in the SDK. The path to each source file can be a good way to determine what folder to place the file in:

../../../../../../components/**drivers_nrf**/delay/nrf_delay.c can be classified as a driver. Thus placed in nRF_Driver, etc.

Enable auto discovery of symbols, include paths and compiler settings

Note regarding SDK 12 9/30-16 - Makefile structure was changed in SDK 12.0.0; there are now two separate Makefiles for each example; the project makefile that defines the example dependencies/flags, and makefile.common to actually build the specified targets. It should only be necessary to modify the project makefile as before. However, apparently the CDT parser does not like that we added quoted toolchain strings. E.g., "C:/Program Files (x86)/GNU Tools ARM Embedded/4.9 2015q3/bin/arm-none-eabi-gcc".

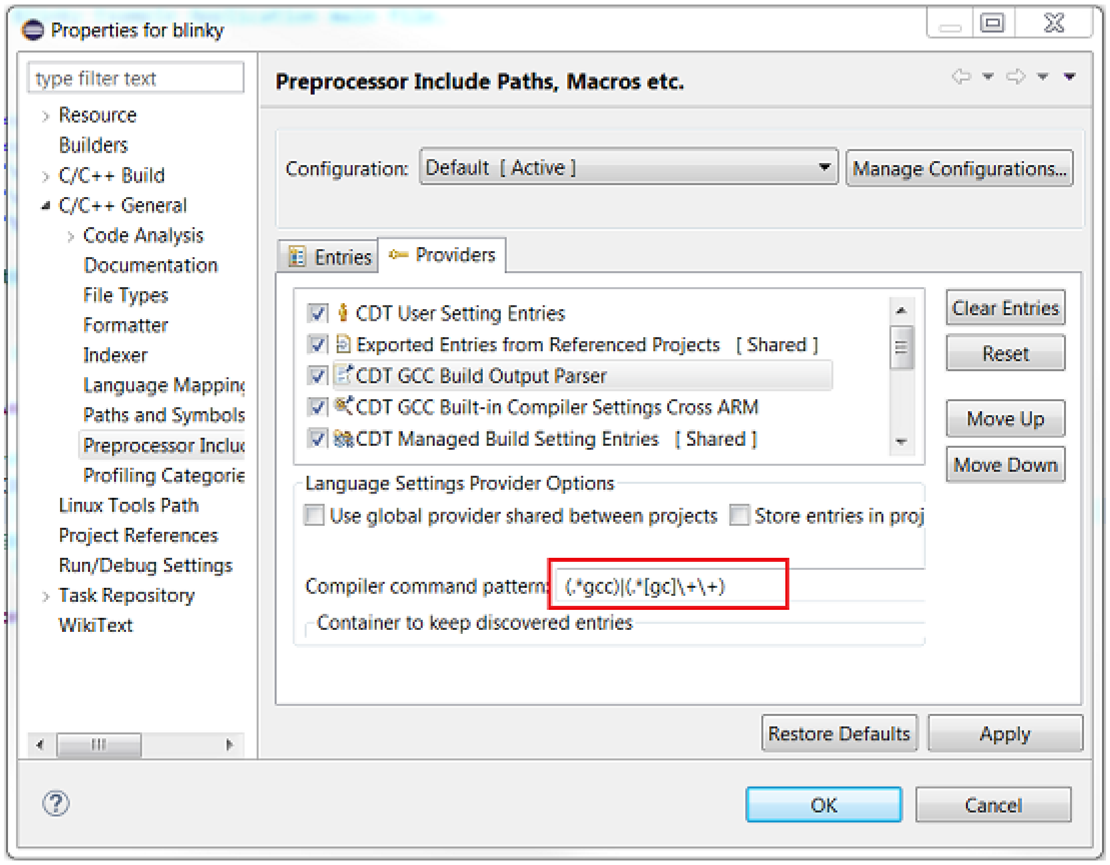
Included a patched version of Makefile.common, see list of attachments at the end of this tutorial.

Good news is that the new makefiles support incremental builds rather than re-building every time. That is, make will only rebuild modified objects and link in those that weren't.

The following steps configure the CDT build output parser and CDT GCC Built-in Compiler Settings Cross ARM to automatically discover symbols, include paths and compiler settings based on the output produced by the Makefile. These steps also apply to imported projects as these configurations are not included in the project files.

- Enter project properties -> C/C++->Preprocessor Include Paths,etc.->Providers

- Click on CDT GCC Build Output Parser and change the compiler command pattern from (gcc)|([gc]\+\+)|(clang) to (.*gcc)|(.*[gc]\+\+) then apply changes.
- Click on CDT Built-in Compiler Settings Cross ARM and replace ${COMMAND} with arm-none-eabi-gcc and click *Apply*.



Configuration: Default [Active]

- [✓] CDT User Setting Entries
- [✓] Exported Entries from Referenced Projects [Shared]
- [✓] CDT GCC Build Output Parser
- [✓] CDT GCC Built-in Compiler Settings Cross ARM
- [✓] CDT Managed Build Setting Entries [Shared]

[ ] Use global provider shared between projects [ ] Store entries in proj

Compiler command pattern: (.*gcc)|(.*[gc]\+\+)

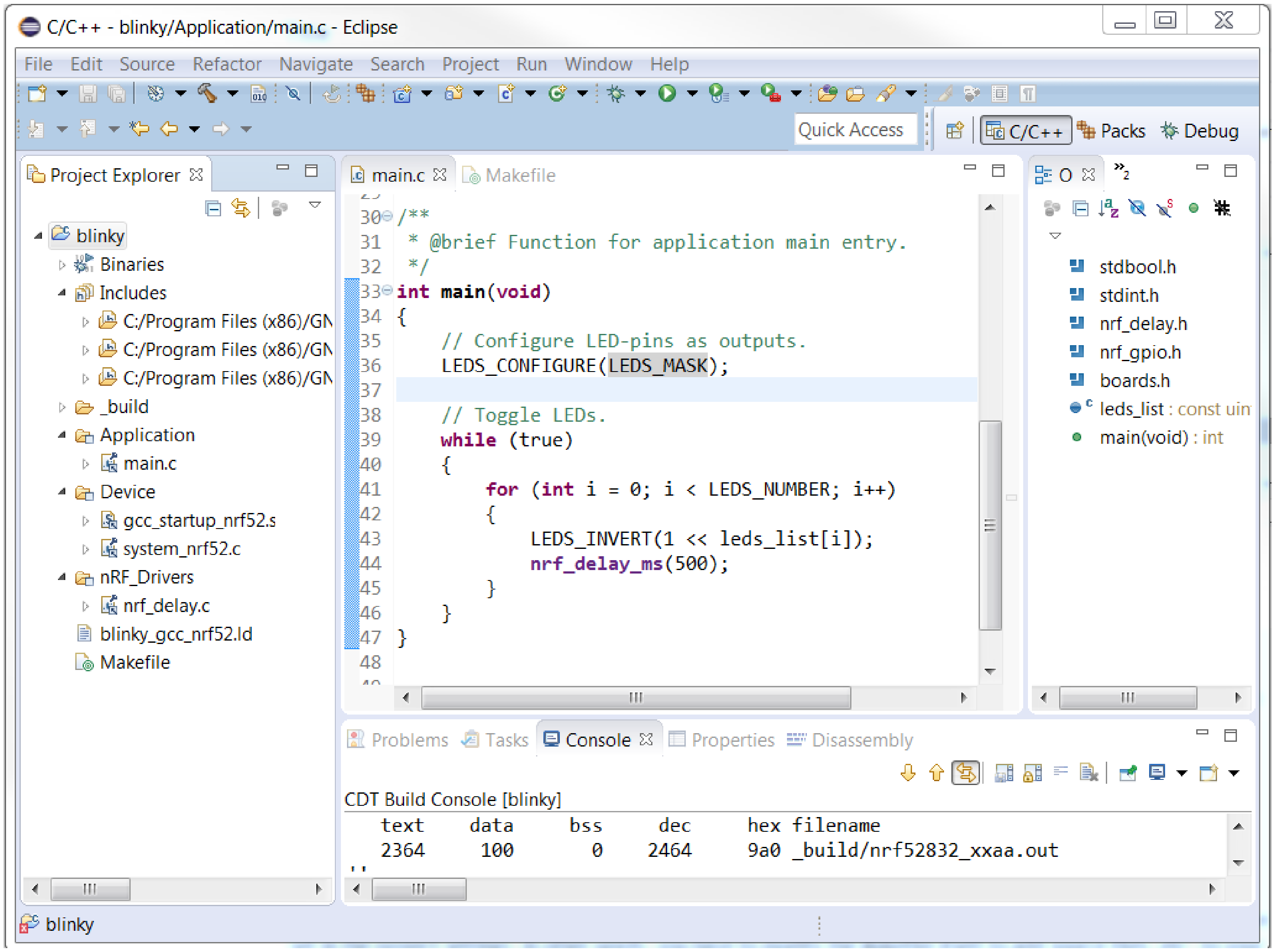
The CDT build output parser works by parsing everything in the output string that has either a '-D' for defines and '-I' for include paths, and 'C:/Program Files (x86)/GNU Tools ARM Embedded/4.9 2015q3/bin/arm-none-eabi-gcc' will be recognized as the compiler command pattern with the settings above.

Example string when compiling main.c in the blinky example:

```
'C:/Program Files (x86)/GNU Tools ARM Embedded/4.9 2015q3/bin/arm-none-eabi-gcc'
-DCONFIG_GPIO_AS_PINRESET
-DBOARD_PCA10036 -DNRF52
-DBSP_DEFINES_ONLY
-mcpu=cortex-m4
-mthumb
-mabi=aapcs
--std=gnu99
-Wall -Werror
-O0
-g3
-mfloat-abi=hard
-mfpu=fpv4-sp-d16
```

```
-ffunction-sections
-fdata-sections
-fno-strict-aliasing
-fno-builtin
--short-enums
-I../../../../../../components/toolchain/gcc
-I../../../../../../components/toolchain
-I../../..
-I../../../../../bsp
-I../../../../../../components/device
-I../../../../../../components/drivers_nrf/delay
-I../../../../../../components/drivers_nrf/hal -c -o _build/main.o ../../../main.c
```

Rebuild your project to index the source files again with the CDT parser enabled. If it works you will notice a key symbol appears on the source files, and all errors should be resolved by now.

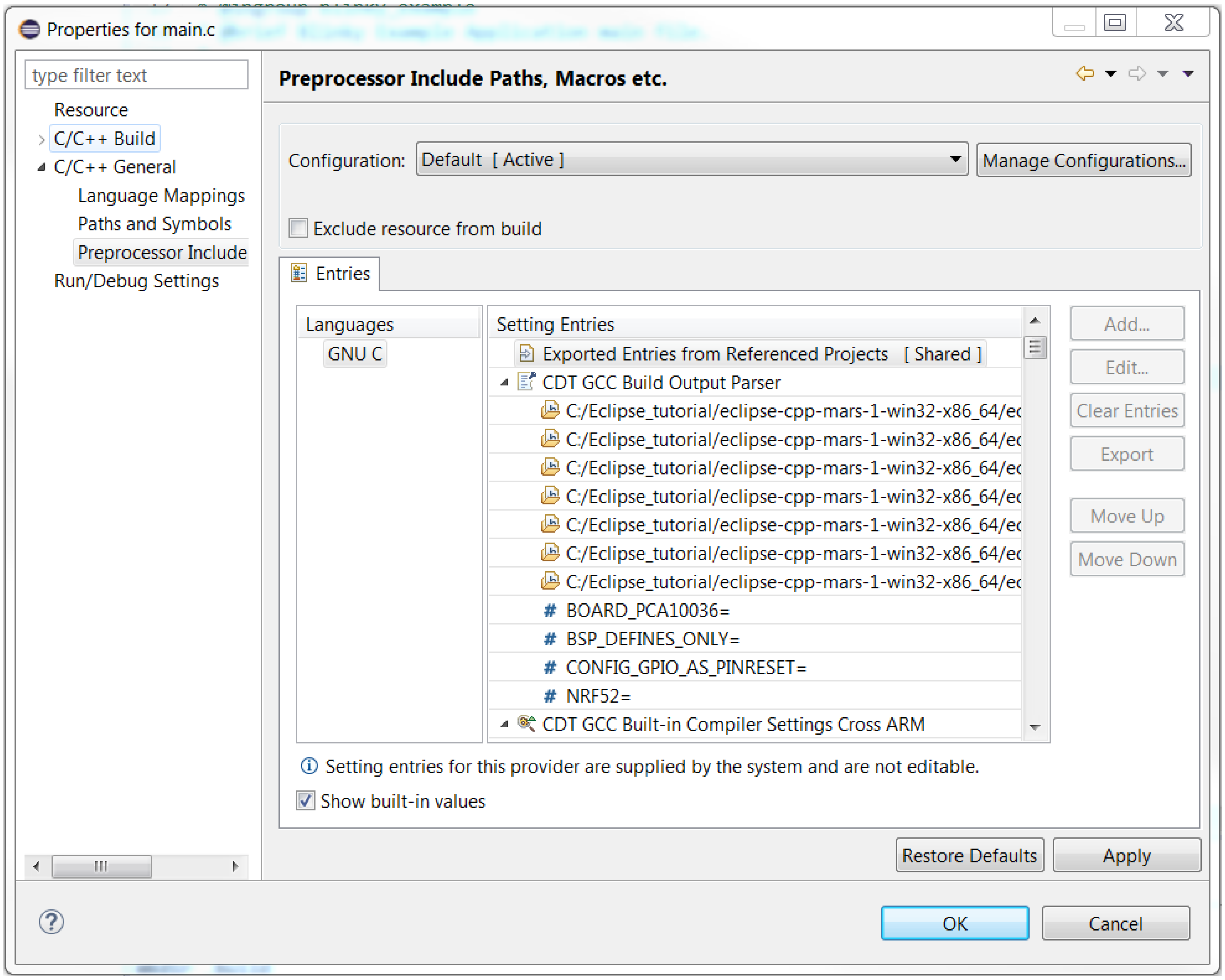


```c
/**
 * @brief Function for application main entry.
 */
int main(void)
{
    // Configure LED-pins as outputs.
    LEDS_CONFIGURE(LEDS_MASK);

    // Toggle LEDs.
    while (true)
    {
        for (int i = 0; i < LEDS_NUMBER; i++)
        {
            LEDS_INVERT(1 << leds_list[i]);
            nrf_delay_ms(500);
        }
    }
}
```

CDT Build Console [blinky]

```
   text    data     bss     dec     hex filename
   2364     100       0    2464     9a0 _build/nrf52832_xxaa.out
```

In case you want to see the paths and symbols added from the output, select one of the source files, click on properties, C/C++ item and preprocessor Include Paths you can see that the paths defines have been added based on the output:



Properties for main.c

Preprocessor Include Paths, Macros etc.

Configuration: Default [Active]

Exclude resource from build

Languages: GNU C

Setting Entries:
- Exported Entries from Referenced Projects [Shared]
- CDT GCC Build Output Parser
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - C:/Eclipse_tutorial/eclipse-cpp-mars-1-win32-x86_64/ec
  - # BOARD_PCA10036=
  - # BSP_DEFINES_ONLY=
  - # CONFIG_GPIO_AS_PINRESET=
  - # NRF52=
- CDT GCC Built-in Compiler Settings Cross ARM

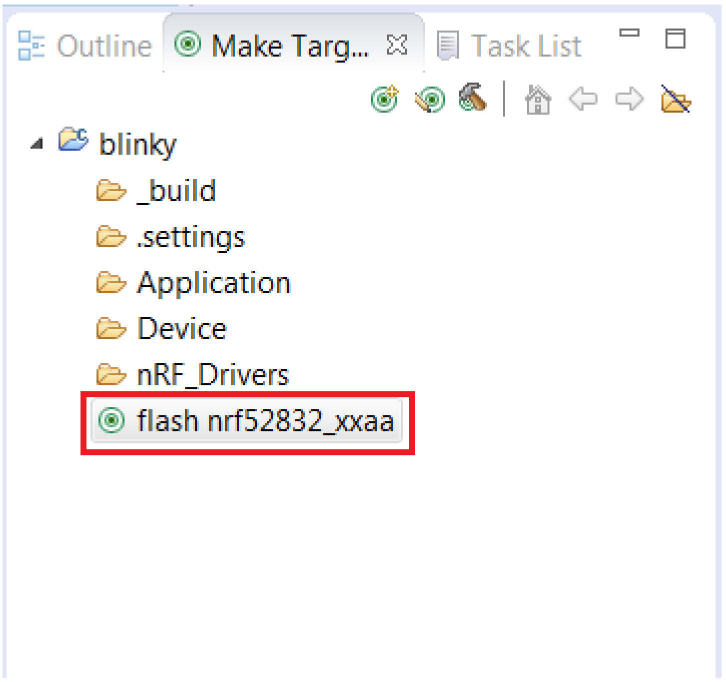
Setting entries for this provider are supplied by the system and are not editable.

Show built-in values

Flash download

Flash downloading and running a program on your device without debugging is possible by using flash targets included in the Makefiles.

In order to execute the flash targets you first need to add a new target to the project. Here is an example with the blinky example. There are no flash softdevice target in this example as it runs independent of the softdevice. For the heart rate or other BLE examples you will need to flash the softdevice by executing the "flash_softdevice" target.



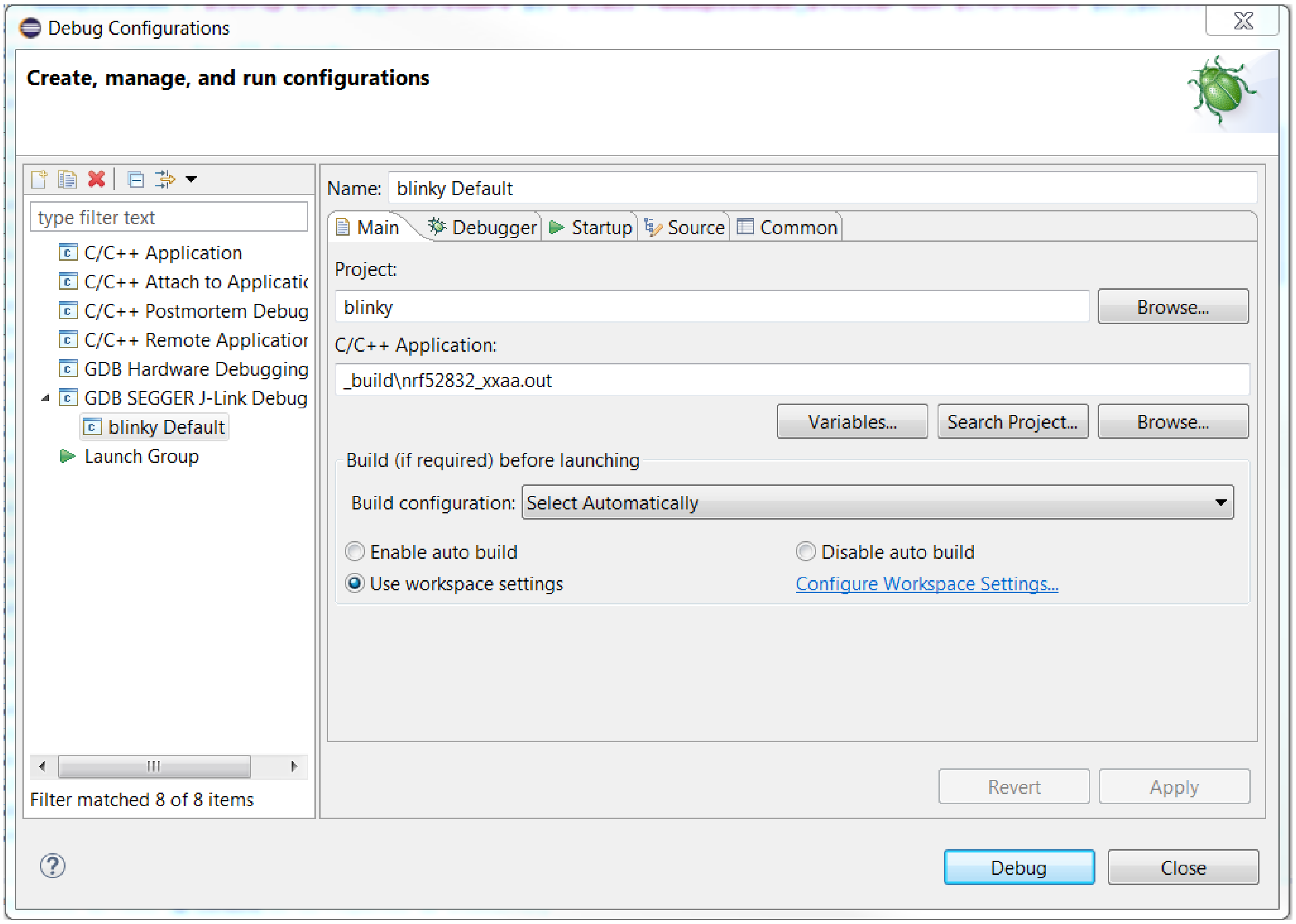
Then execute the flash target(s) to load the FW to target. You will see the commands invoked in the console window. Once you have done this your application will start running.

Note, makefiles from SDK 11.0.0 does not require the target name to be specified. so just 'flash'.

Setting up a project for debugging in Eclipse

The GNU ARM Eclipse plugin we installed earlier has already integrated J-link debugging with Eclipse. We only need to set the debug configurations for our project before we can launch a debug session.

In the project explorer select your project in the project explorer window. Click on the Run tab in the menu bar and go to debug configurations. Right click on GDB SEGGER *J-Link Debugging* and select new. Project name and location of executable should be added automatically as long as you selected your project before opening *debug configurations*.



Name: blinky Default

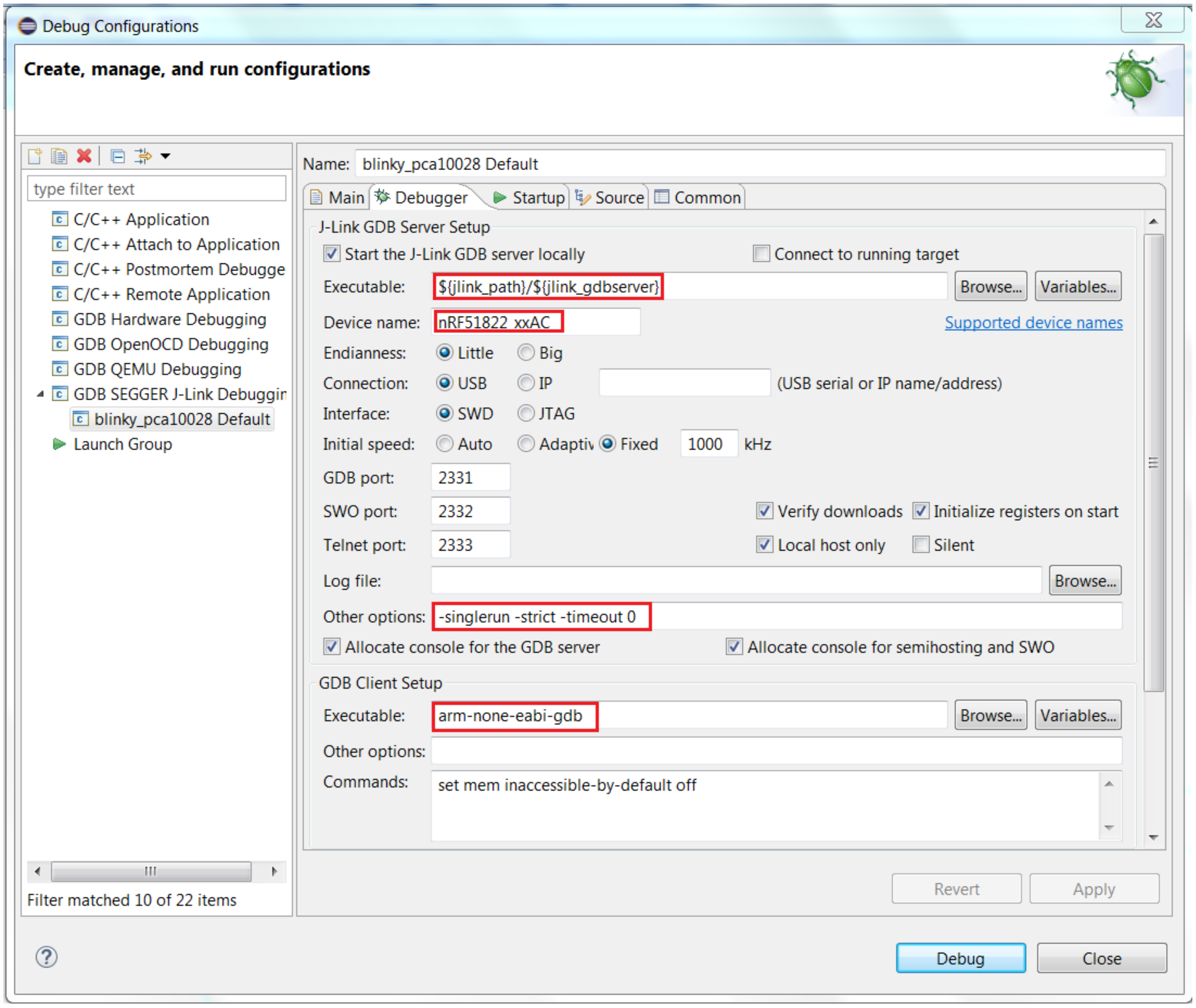
Project: blinky

C/C++ Application: _build\nrf52832_xxaa.out

Build (if required) before launching

Build configuration: Select Automatically

Enable auto build / Disable auto build / [✓ Use workspace settings]



Name: blinky_pca10028 Default

J-Link GDB Server Setup

[✓] Start the J-Link GDB server locally [ ] Connect to running target

Executable: ${jlink_path}/${jlink_gdbserver}

Device name: nRF51822_xxAC

Endianness: [● Little] ○ Big

Connection: [● USB] ○ IP (USB serial or IP name/address)

Interface: [● SWD] ○ JTAG

Initial speed: ○ Auto ○ Adaptiv [● Fixed] 1000 kHz

GDB port: 2331

SWO port: 2332 [✓] Verify downloads [✓] Initialize registers on start

Telnet port: 2333 [✓] Local host only [ ] Silent

Log file:

Other options: -singlerun -strict -timeout 0

[✓] Allocate console for the GDB server [✓] Allocate console for semihosting and SWO

GDB Client Setup

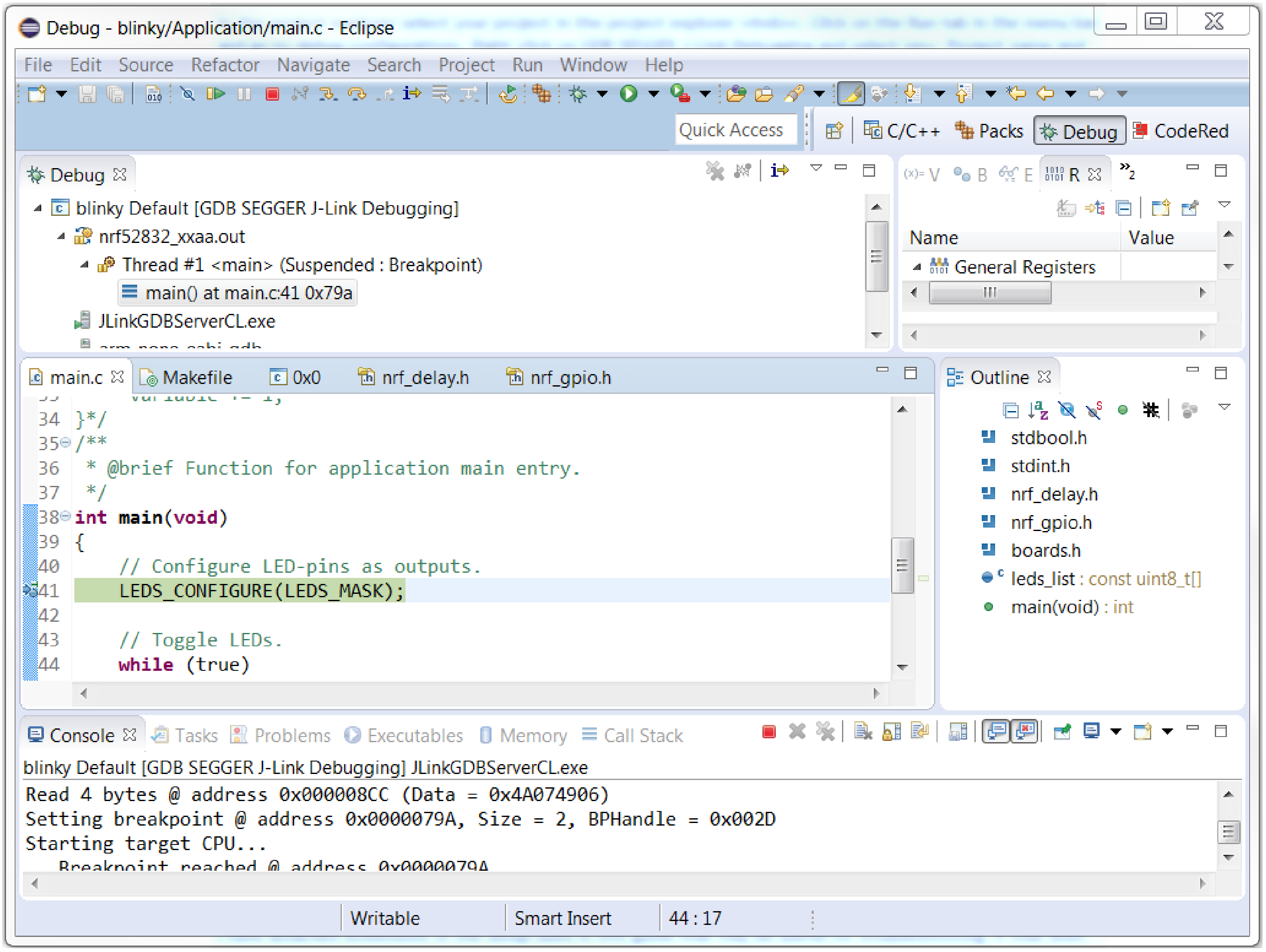
Executable: arm-none-eabi-gdb

Other options:

Commands: set mem inaccessible-by-default off

Enter the *Debugger* tab, and select the device name for the device you are targeting and make sure that the jlink variables are set correctly (must point to JLinkGDBServerCL executable). The 'nogui' option may be removed from *other options* field to get a device selection prompt if multiple J-link programmers are connected to the PC. Lastly, set the GDB client to *arm-non-eabi-gdb* to ensure that the correct client is used (may default to GDB client for host if not).

Then deselect SWO in *startup* tab unless you have configured it (not supported on the 51 series). Now you start the debug session by clicking on the *Debug* button. Eclipse will then show the debug view, and the application will break in main by default.


Debug - blinky/Application/main.c - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
C/C++ Packs Debug CodeRed
Debug
blinky Default [GDB SEGGER J-Link Debugging]
nrf52832_xxaa.out
Thread #1 <main> (Suspended : Breakpoint)
main() at main.c:41 0x79a
JLinkGDBServerCL.exe
Name Value
General Registers
main.c Makefile 0x0 nrf_delay.h nrf_gpio.h
Outline
stdbool.h
stdint.h
nrf_delay.h
nrf_gpio.h
boards.h
leds_list : const uint8_t[]
main(void) : int
Console Tasks Problems Executables Memory Call Stack
Writable Smart Insert 44 : 17

```
}*/
/**
 * @brief Function for application main entry.
 */
int main(void)
{
    // Configure LED-pins as outputs.
    LEDS_CONFIGURE(LEDS_MASK);

    // Toggle LEDs.
    while (true)
```

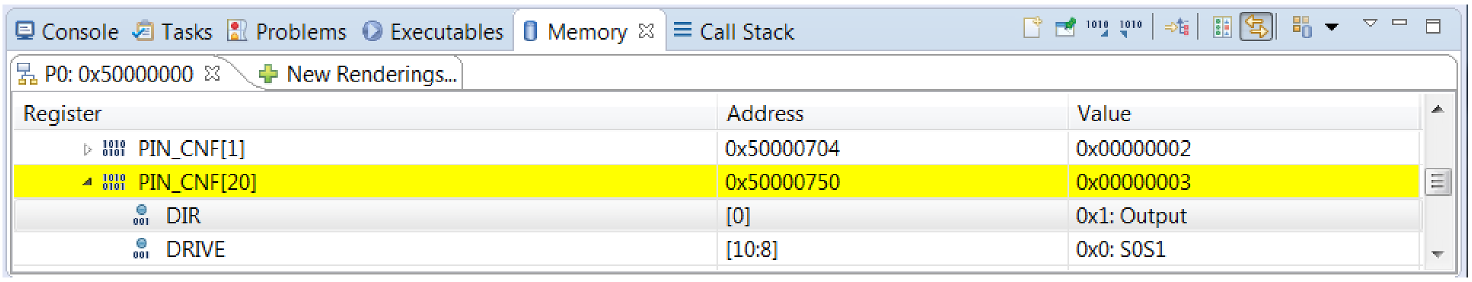
blinky Default [GDB SEGGER J-Link Debugging] JLinkGDBServerCL.exe

```
Read 4 bytes @ address 0x000008CC (Data = 0x4A074906)
Setting breakpoint @ address 0x0000079A, Size = 2, BPHandle = 0x002D
Starting target CPU...
```

Now you can start debugging; set break points, single step, read registers, etc.

Using the peripheral viewer

Go to project properties and select C/C++ Build item. Enter settings and click on the *Devices* tab. Here you must select the chip you are using to enable the view. Start a debug session and open the peripheral viewer, and select the Peripheral you want to debug. If you select the GPIO module you can get an visual indications by controlling the board LEDs (P0 on nRF52 and GPIO on nRF51). Open the Memory view and click on PIN_CNF[24] or PIN_CNF[20] to control the LED 4 GPIO on your DK kit (PCA10028 or PCA10036/40). Changing the GPIO direction in PIN_CNF[x].DIR should toggle your board LED.



Register	Address	Value
PIN_CNF[1]	0x50000704	0x00000002
PIN_CNF[20]	0x50000750	0x00000003
DIR	[0]	0x1: Output
DRIVE	[10:8]	0x0: S0S1

Troubleshooting

Below is a brief guide on how to troubleshoot some of the common problems related to GCC and Eclipse set-up with the Nordic SDK.

Development with GCC and Eclipse.pdf

Please post a detailed description of the problem in the questions section if the guide does not help in solving a particular issue. Also feel free to provide feedback if you think something is missing/unclear or any other suggestions.

Attachments

- nRF5_SDK_12.2.0_ble_app_hrs_pca10040_s132.zip
- nRF5_SDK_11.0.0_ble_app_hrs_dfu_pca10028.zip
- nRF5_SDK_11.0.0_ble_app_hrs_dfu_pca10040.zip
- nRF5_SDK_11.0.0_blinky_blank_pca10028.zip
- nRF5_SDK_11.0.0_blinky_blank_pca10040.zip
- nRF52_SDK_0.9.2_blinky_blank_PCA10040_36.zip
- nRF52_SDK_0.9.2_ble_app_hrs_PCA10040_36.zip
- nRF51_SDK_10.0.0_blinky_s110_pca10028.zip
- nRF51_SDK_10.0.0_ble_app_hrs_dfu_s110_pca10028.zip

Tested with Eclipse Neon:

- nRF51_SDK_12.1.0_blinky_blank_pca10028.zip
- nRF5_SDK13.0.0_ble_app_hrs_pca10040_s132.zip

Patch for SDK 12 - implements required workaround for CDT build output parser. Replace makefile.common in in $(SDK_ROOT)/components/toolchain/gcc with one of the files below:

- makefile.common for SDK 12.1.0

- makefile.common for SDK 12.2.0 and SDK 13.0.0

321 comments 0 members are here

ments

venerley over 1 year ago +6

Hi Vidar, I was wondering if you had a New Years resolution for 2018? :O) Given that this Tutori...

kramlevocs over 1 year ago +2

.cproject (/attachment/787501le0bc377ae64cad5c0b5a005b3)This tutorial was difficult for us s...

Henry 11 months ago +2

Unfortunately, it is extremely hard to follow-up or searching string from comments/replies due t...