Using Eclipse and GDB with Freescale MQX™

PRODUCT:	Freescale MQX™ RTOS
PRODUCT VERSION:	Freescale MQX 4.1.0
DESCRIPTION:	Using Eclipse and GDB with Freescale MQX™
RELEASE DATE:	December, 2013

Table of Contents

1 Read Me First

This document describes the steps required to configure the GNU and Eclipse CDT development tools and use it to build, run, and debug applications of the Freescale MQX™ RTOS operating system. See Getting Started with Freescale MQX™ RTOS and other user documentation included within the latest Freescale MQX RTOS installation for more details not specifically related to the IAR Embedded Workbench tools.

Get the latest Freescale MQX RTOS at freescale.com/mqx.

2 MQX Build – first steps

The MQX release provides the makefiles allowing convenient way to build MQX libraries and applications from Windows or Linux command line. The makefiles can be also integrated in various IDEs, this document describe integration with Eclipse IDE. Prior the use please make following setting to prepare your build environment.

Windows operation system – common scenario is to use mingw utilities with Windows "cmd" utility

- Install a mingw from http://sourceforge.net/projects/mingw/ to default location "c:\MinGW".
- Please ensure that your PATH variable contains "c:\MinGW\bin".
- Edit "build/common/make/global.mak" to setup a valid cross compiler directory path to variable TOOLCHAIN_ROOTDIR. The file contains some commented examples for each toolchain. The path cannot contain whitespaces, so you might want to use a Windows "command" utility to obtain an DOS path without spaces.

Linux operation system:

- Ensure you have installed "make" and "sed" programs.
- Edit "build/common/make/global.mak" to setup a valid cross compiler directory path to variable TOOLCHAIN_ROOTDIR. The file contains some commented examples for each toolchain. The path cannot contain whitespaces
- Edit "build/common/make/global.mak" to setup a HOSTENV variable to value UNIX

Note : The description below is provided for twrk60n512 BSP and Hello World example application. The same procedure applies for all other BSPs and examples.

2.1 Building the MQX using command line

See Chapter 2 of the MQX Getting started document for details on generic build process and compile time configuration. This chapter concentrates on steps related to makefiles only.

2.1.1 Batch Build (Windows only)

MQX library build

Launch one of batch files, placed in "build/<board>/make" directory, to run a build process of all libraries.

```
build/twrk60n512/make/build_gcc_arm.bat
```

MQX application build

For applications execute the batch file in application project directory. Navigate

```
/build/make/<project_name>_<board>/build_<tool>.bat
```

2.1.2 Makefile build

MQX library build

Navigate to the "<project/directory>/build/<project_name>_<board>/make" directory, for example "mqx/build/make/bsp_twrk60n512" and run following command to build selected MQX library (in this case BSP in debug configuration).

```
mingw32-make TOOL=gcc_arm CONFIG=debug build
```

Make parameters description:

TOOL	- name of the toolchain.
CONFIG	- represent the name of the configuration. Configuration is defined by specific flags and include paths. We provide two configurations "**debug**" (low optimization level) and "**release**" (high optimization level).
target	Allowed targets are

build	- run build process
clean	- run clean process
rebuild	- run clean and build processes
help	- default target, prints the help
debugme	- print the internal variables

MQX application build

To build or clean an application navigate to example directory run "make" command specify CONFIG, TOOL and LOAD or LINKER_FILE parameters.

In directory:

```
mqx/examples/hello/build/make/hello_twrk60n512
```

Run following command.

```
mingw32-make TOOL=gcc_arm CONFIG=debug LOAD=intflash build
```

Make parameters description:

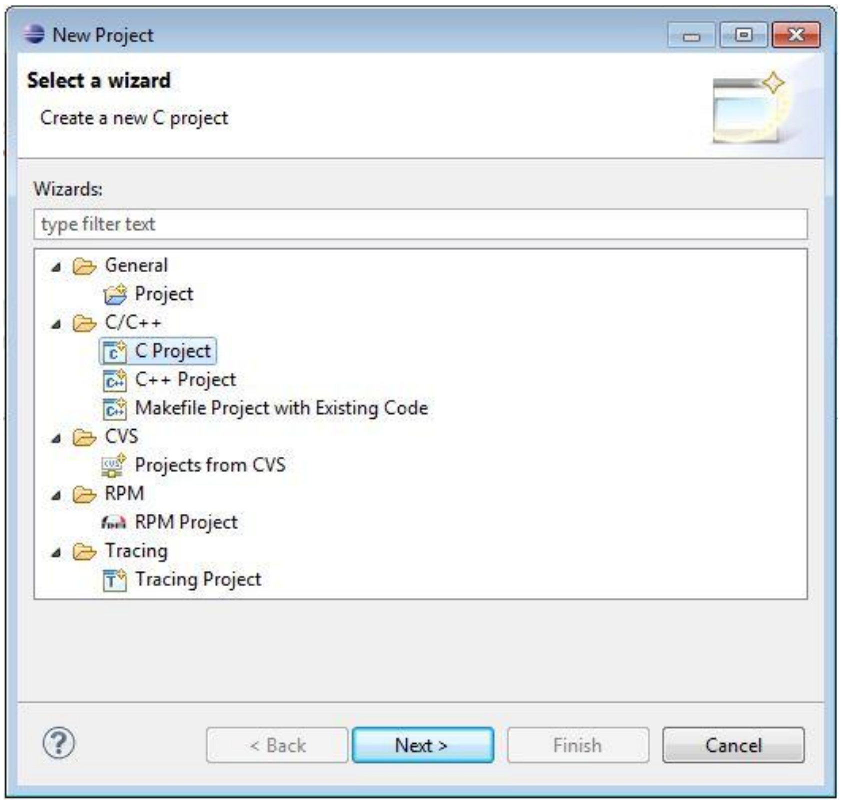
TOOL	- name of the toolchain.
CONFIG	- represent the name of the configuration. Configuration is defined by specific flags and include paths. We provide two configurations "**debug**" (low optimization level) and "**release**" (high optimization level).
LOAD	- name of linker file, use LOAD parameter in case linker command file is placed in BSP output directory. Other possibility is to use LINKER_FILE parameter with full path.

LINKER_FILE - full path to the linker file, use in case you want to configure full path to the linker command file and not to use default one from BSP output directory

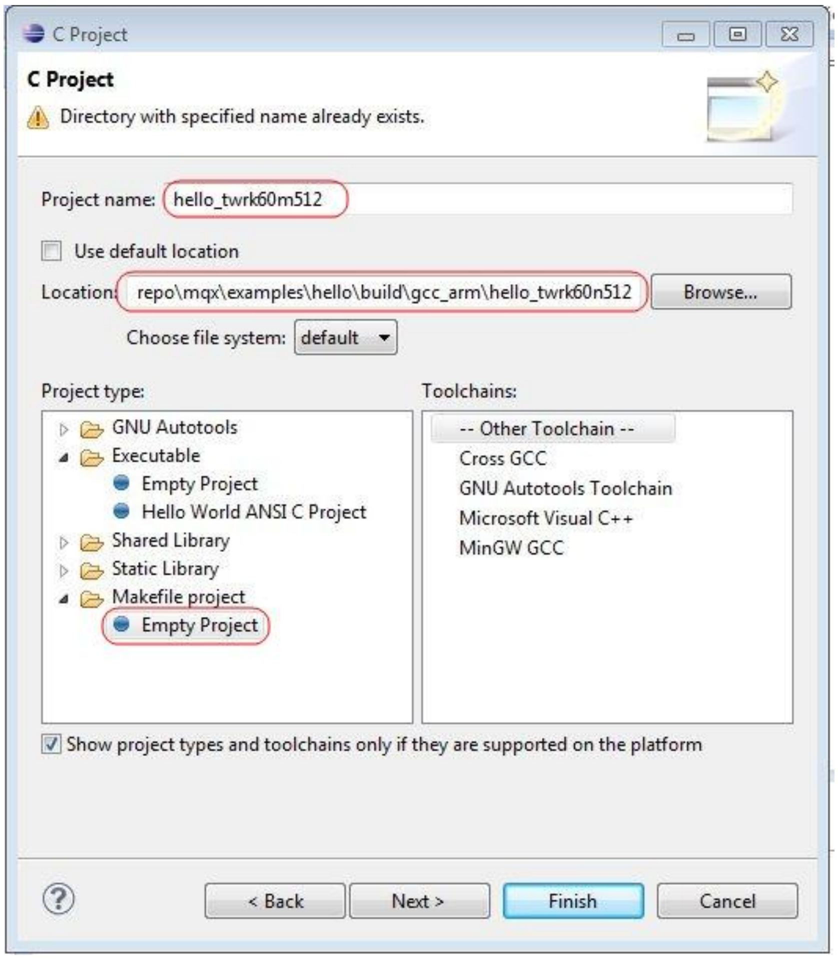
3 Using MQX Makefiles with Eclipse

The section below describes integration provided MQX makefiles with Eclipse CDT development environment. Follow the steps below.

- Create a new C Project



- Select the project name and place it in appropriate directory, for instance: **/mqx/examples/hello/build/gcc_arm/hello_twrk60n512/**


C Project

C Project

⚠ Directory with specified name already exists.

Project name: hello_twrk60m512

☐ Use default location

Location: repo\mqx\examples\hello\build\gcc_arm\hello_twrk60n512

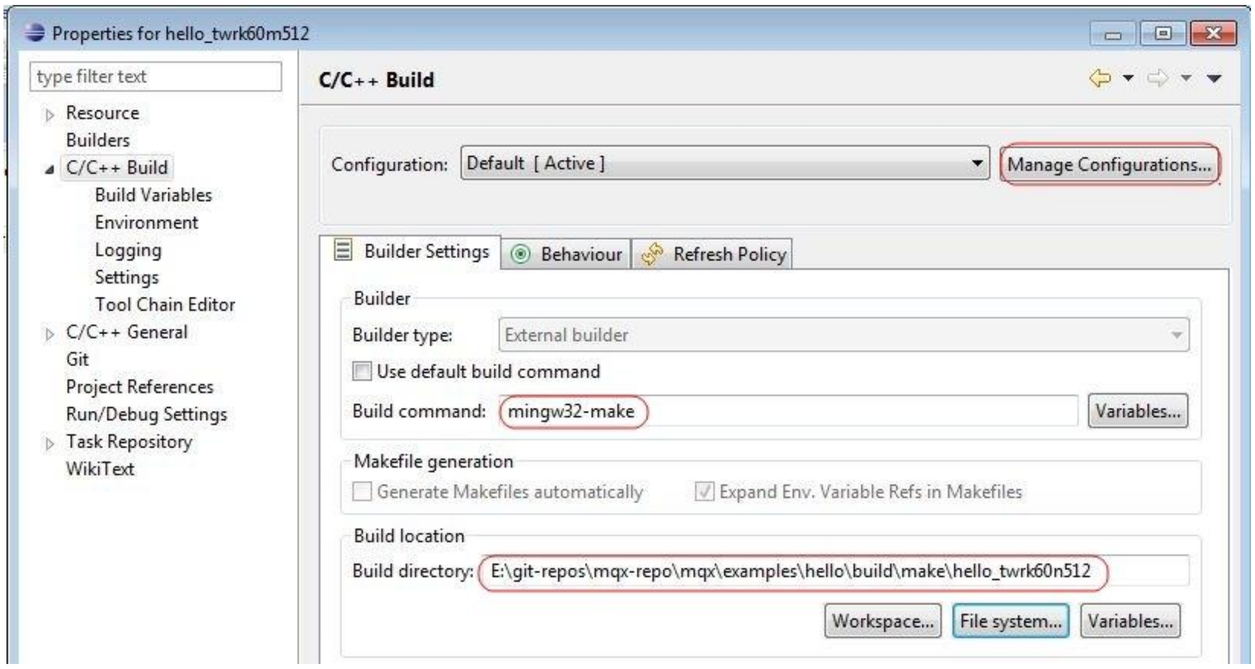
Choose file system: default

Project type:
- GNU Autotools
- Executable
  - Empty Project
  - Hello World ANSI C Project
- Shared Library
- Static Library
- Makefile project
  - Empty Project

Toolchains:
- -- Other Toolchain --
- Cross GCC
- GNU Autotools Toolchain
- Microsoft Visual C++
- MinGW GCC

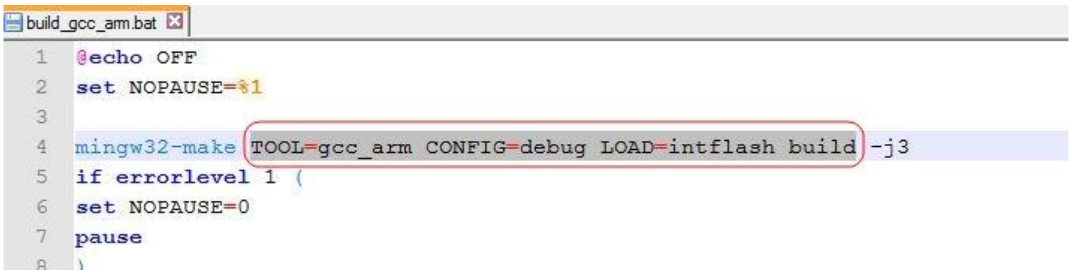
☑ Show project types and toolchains only if they are supported on the platform

- Setup a make utility (mingw32-make in this case) and build directory path to makefile directory, for instance: **/mqx/examples/hello/build/make/hello_twrk60n512/ .**
  You might want to change the configuration name by Manage Configurations.



Configuration: Default [Active]
Builder type: External builder
☐ Use default build command
Build command: mingw32-make
☐ Generate Makefiles automatically
☑ Expand Env. Variable Refs in Makefiles
Build directory: E:\git-repos\mqx-repo\mqx\examples\hello\build\make\hello_twrk60n512

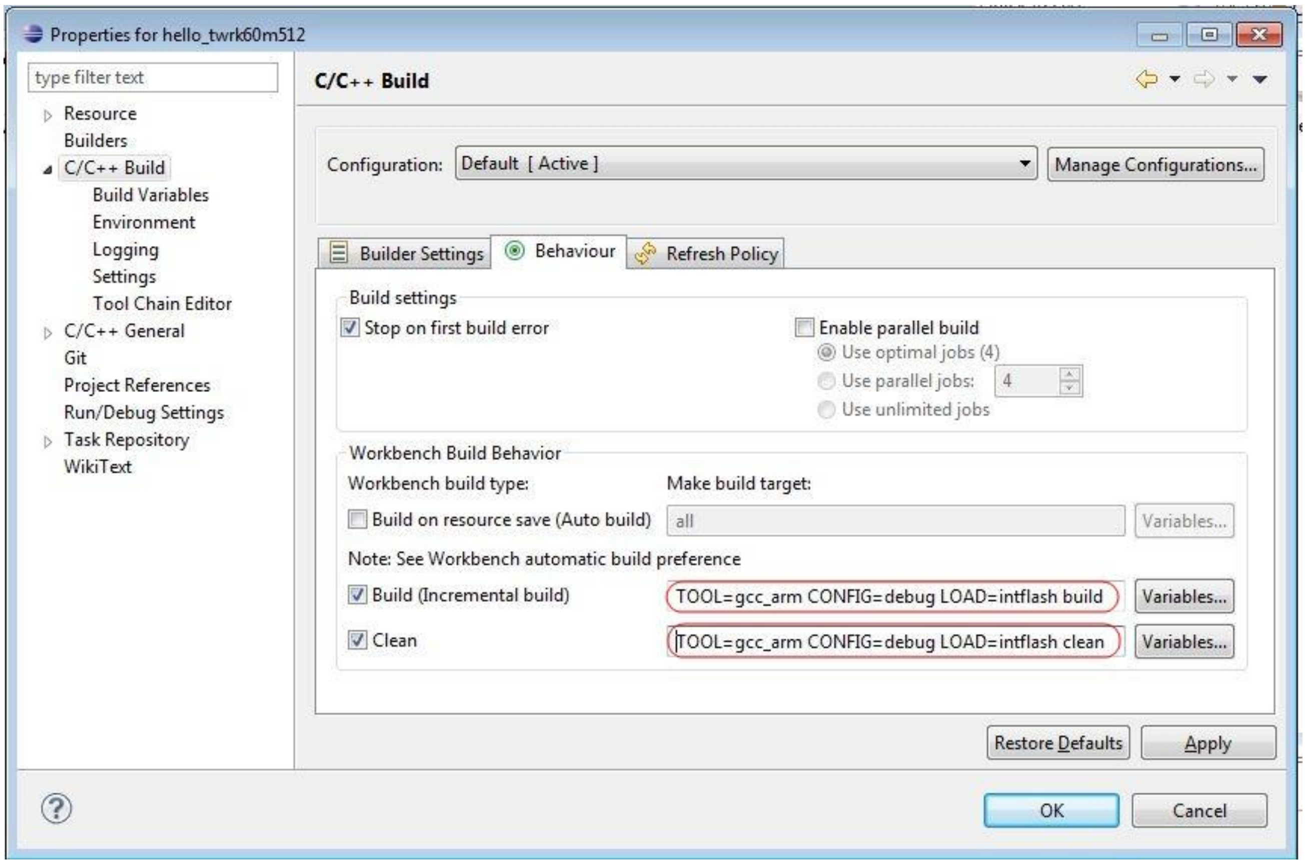
- Open a makefile batch file, for instance **/mqx/examples/build//make/hello_twrk60n512/**buid_gcc_arm.bat



```
@echo OFF
set NOPAUSE=%1

mingw32-make TOOL=gcc_arm CONFIG=debug LOAD=intflash build -j3
if errorlevel 1 (
set NOPAUSE=0
pause
)
```

- Setup commands to textbox field “build”and “clean”.


Properties for hello_twrk60m512

C/C++ Build

Configuration: Default [Active]

Builder Settings | Behaviour | Refresh Policy

Build settings

- [✓] Stop on first build error
- [ ] Enable parallel build
  - Use optimal jobs (4)
  - Use parallel jobs: 4
  - Use unlimited jobs

Workbench Build Behavior

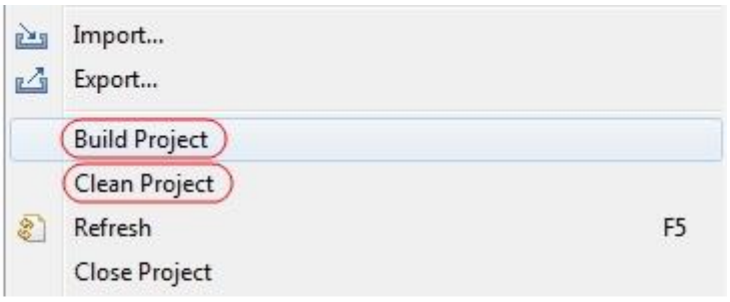
Workbench build type: Make build target:

- [ ] Build on resource save (Auto build) — all

Note: See Workbench automatic build preference

- [✓] Build (Incremental build) — TOOL=gcc_arm CONFIG=debug LOAD=intflash build
- [✓] Clean — TOOL=gcc_arm CONFIG=debug LOAD=intflash clean

- Now you are able to perform build and clean operation over project.



- If you want to see a project source files you need to copy and replace your “.project” file with “.project” file from “cw10gcc/hello_twrk60n512” directory.

Output elf file will be placed at:
/mqx/examples/hello/build/make/hello_twrk60n512/gcc_arm/intflash_debug/

If you want to change output directory of the elf file you need to specify APPLICATION_DIR or APPLICATION_FILE variable at step no. 5.
For instance: APPLICATION_DIR=../../gcc_arm/hello_twrk60n512

You may want to import the generated binary, so you will be able to debug it (described in the next chapter). To do so, go to the File/Import menu and select the C/C++ Executable under the C/C++ category.

4 Running and Debugging the MQX application

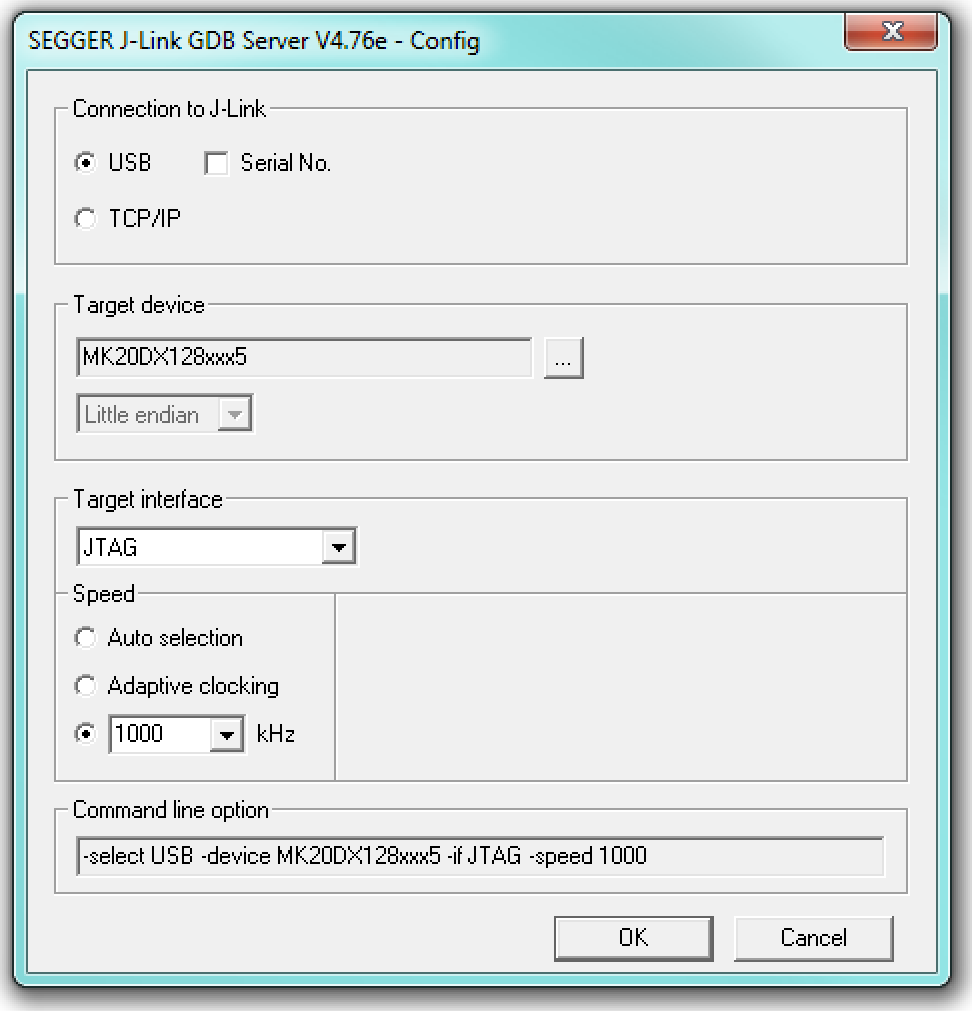
The description below has been prepared for twrk20d50m BSP, the Hello World example and the J-Link for ARM hardware debug probe. The same procedure applies for all other BSPs and examples.

4.1 Starting the GDB Server

The GDB Server in the described configuration is a bridge between the GDB debugger and the J-Link debug probe. You can find the GDB Server executable in the following folder:

```
<jlink_tools_install_dir>/JLinkGDBServer.exe
```

When you run it, you will see the connection dialog, where you may select the connection options for your board.


SEGGER J-Link GDB Server V4.76e - Config

Connection to J-Link: [✓ USB] Serial No. TCP/IP

Target device: MK20DX128xxx5

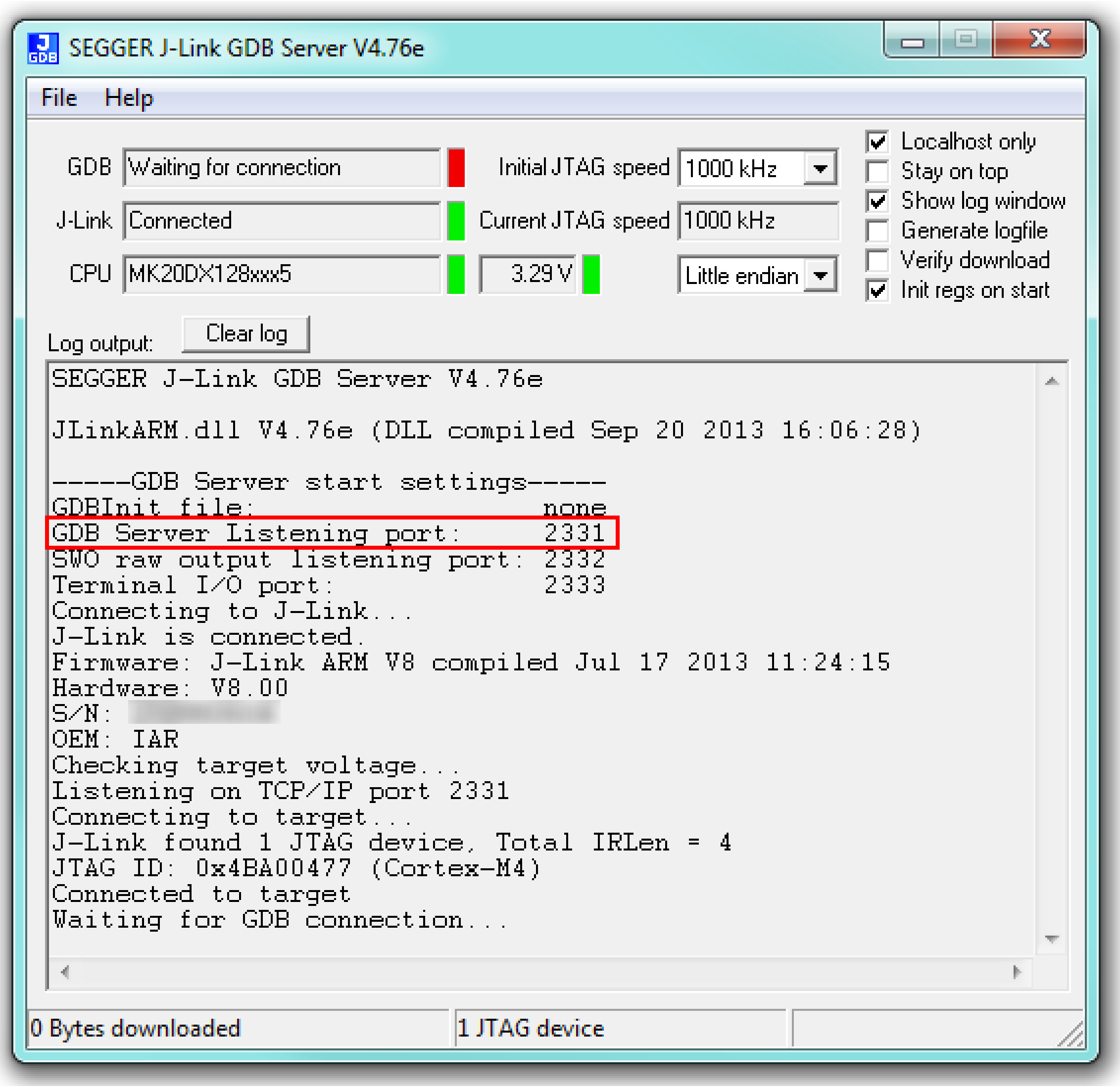
Little endian

Target interface: JTAG

Speed: Auto selection Adaptive clocking [✓ 1000] kHz

Command line option: -select USB -device MK20DX128xxx5 -if JTAG -speed 1000

After you have selected the options for your board, press the OK button. You will see the dialog indicating the GDB Server status and the configuration details like the listening port.



GDB: Waiting for connection 🟥
J-Link: Connected 🟩
CPU: MK20DX128xxx5 🟩
3.29 V 🟩
Initial JTAG speed: 1000 kHz
Current JTAG speed: 1000 kHz
Little endian

✓ Localhost only
☐ Stay on top
✓ Show log window
☐ Generate logfile
☐ Verify download
✓ Init regs on start

Log output:

```
SEGGER J-Link GDB Server V4.76e

JLinkARM.dll V4.76e (DLL compiled Sep 20 2013 16:06:28)

-----GDB Server start settings-----
GDBInit file:                  none
GDB Server Listening port:     2331
SWO raw output listening port: 2332
Terminal I/O port:             2333
Connecting to J-Link...
J-Link is connected.
Firmware: J-Link ARM V8 compiled Jul 17 2013 11:24:15
Hardware: V8.00
S/N: [redacted]
OEM: IAR
Checking target voltage...
Listening on TCP/IP port 2331
Connecting to target...
J-Link found 1 JTAG device, Total IRLen = 4
JTAG ID: 0x4BA00477 (Cortex-M4)
Connected to target
Waiting for GDB connection...
```

0 Bytes downloaded | 1 JTAG device

Now, when the GDB Server is running, you are ready to connect to it. You may debug the application either form the command line (not described in this document) or using Eclipse with CDT (see the next subchapter), the latter option seems more convenient for most developers.

4.2 Using Eclipse with the GDB Server

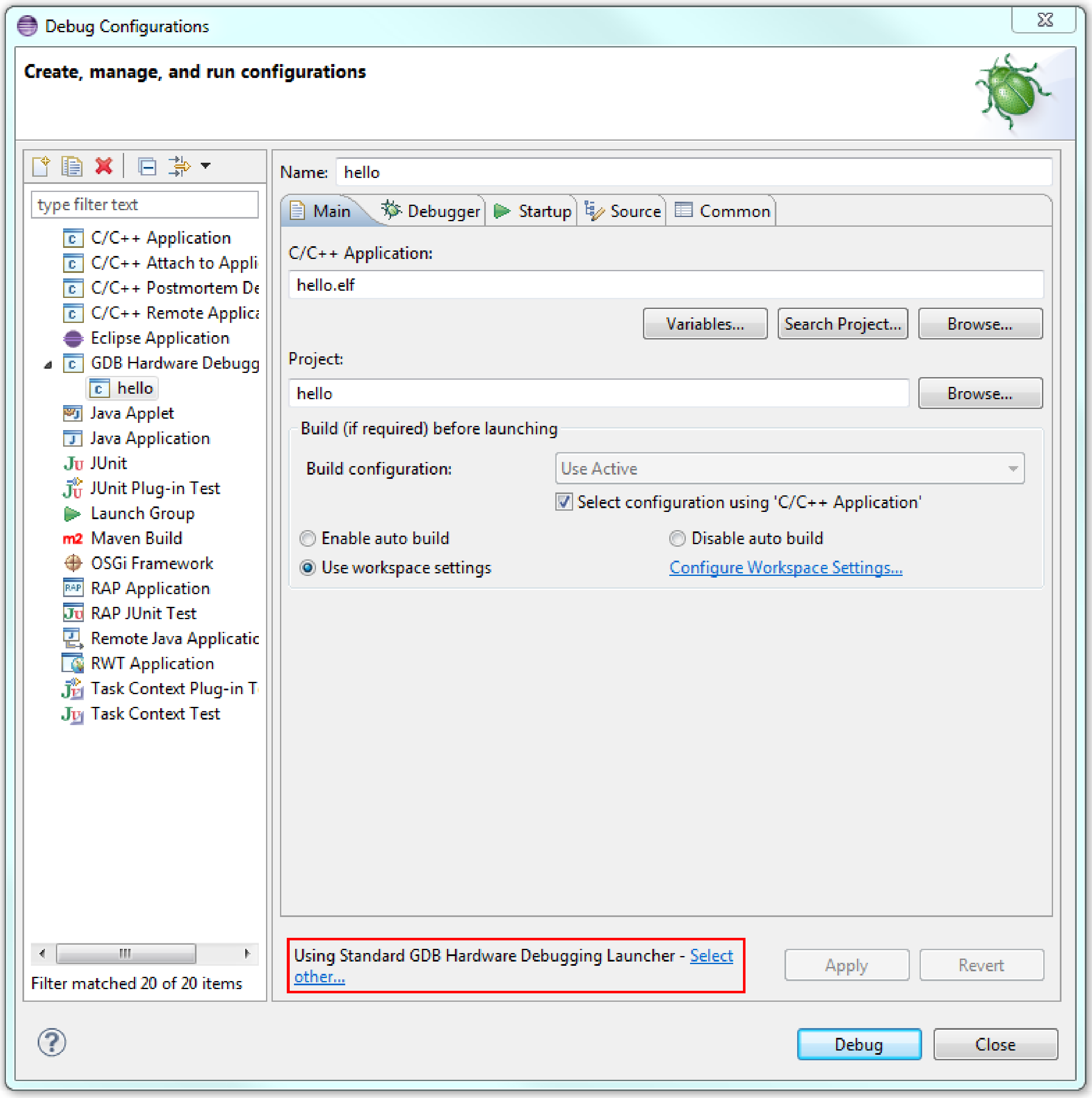
4.2.1 Prerequisites

Eclipse with CDT and the C/C++ GDB Hardware Debugging feature installed.

GNU Tools for ARM Embedded Processors (https://launchpad.net/gcc-arm-embedded).

4.2.2 Creating the debug configuration in Eclipse

Go to the “Run/Debug Configurations…” menu and create a new GDB Hardware Debugging configuration. The first thing you need to modify is the Launcher, instead of the default “GDB (DSF) Hardware Debugging Launcher” select “Standard GDB Hardware Debugging Launcher” – see the image below.



Debug Configurations

Create, manage, and run configurations

Name: hello

Main | Debugger | Startup | Source | Common

C/C++ Application:
hello.elf

Project:
hello

Build (if required) before launching

Build configuration: Use Active

[✓] Select configuration using 'C/C++ Application'

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

Configure Workspace Settings...

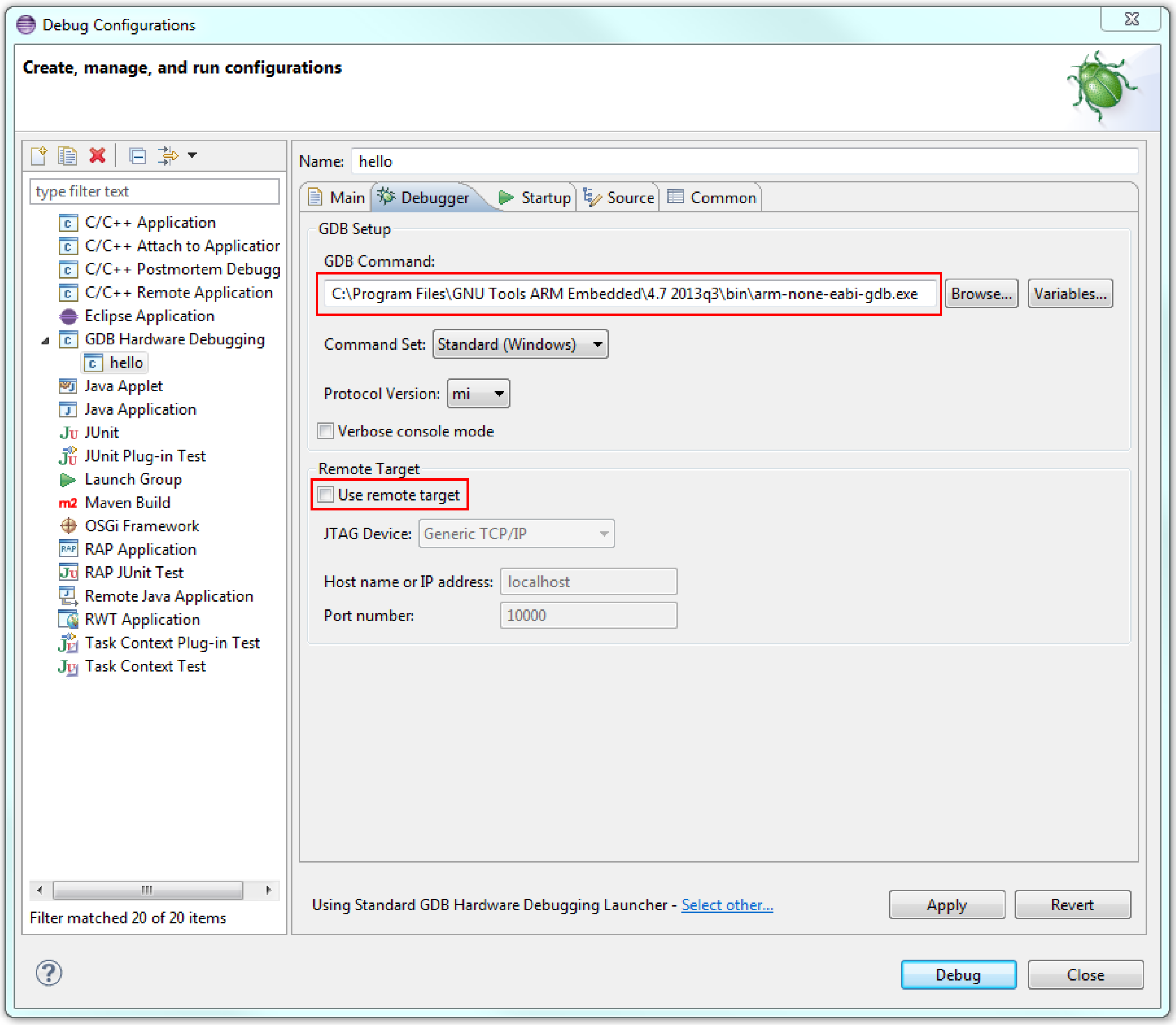
type filter text

- C/C++ Application
- C/C++ Attach to Appli
- C/C++ Postmortem De
- C/C++ Remote Applic
- Eclipse Application
- GDB Hardware Debugg
  - hello
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- Maven Build
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote Java Applicatic
- RWT Application
- Task Context Plug-in T
- Task Context Test

Filter matched 20 of 20 items

Using Standard GDB Hardware Debugging Launcher - Select other...

Next, go to the Debugger tab and set the path to the GDB client from the GNU Tools package and uncheck the “Use remote target option” as in the screenshot below.


Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Applicatior
- C/C++ Postmortem Debugg
- C/C++ Remote Application
- Eclipse Application
- GDB Hardware Debugging
  - hello
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- Maven Build
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote Java Application
- RWT Application
- Task Context Plug-in Test
- Task Context Test

Filter matched 20 of 20 items

Name: hello

Main | Debugger | Startup | Source | Common

GDB Setup

GDB Command: C:\Program Files\GNU Tools ARM Embedded\4.7 2013q3\bin\arm-none-eabi-gdb.exe

Command Set: Standard (Windows)

Protocol Version: mi

Verbose console mode

Remote Target

Use remote target

JTAG Device: Generic TCP/IP

Host name or IP address: localhost

Port number: 10000

Using Standard GDB Hardware Debugging Launcher - Select other...

Now you should set the startup options – go to the “Startup” tab and

Into the “Initialization Commands” text field paste the following content:

```
# connect to the gdb server
target remote localhost:2331
# Set gdb server to little endian
monitor endian little
# Set JTAG speed to 1000 kHz
monitor speed 1000
# Reset the target
monitor reset
monitor sleep 100
# Set JTAG speed in khz
monitor speed auto
# Vector table placed in RAM
monitor writeu32 0xE000ED08 = 0x1FFF8000
```

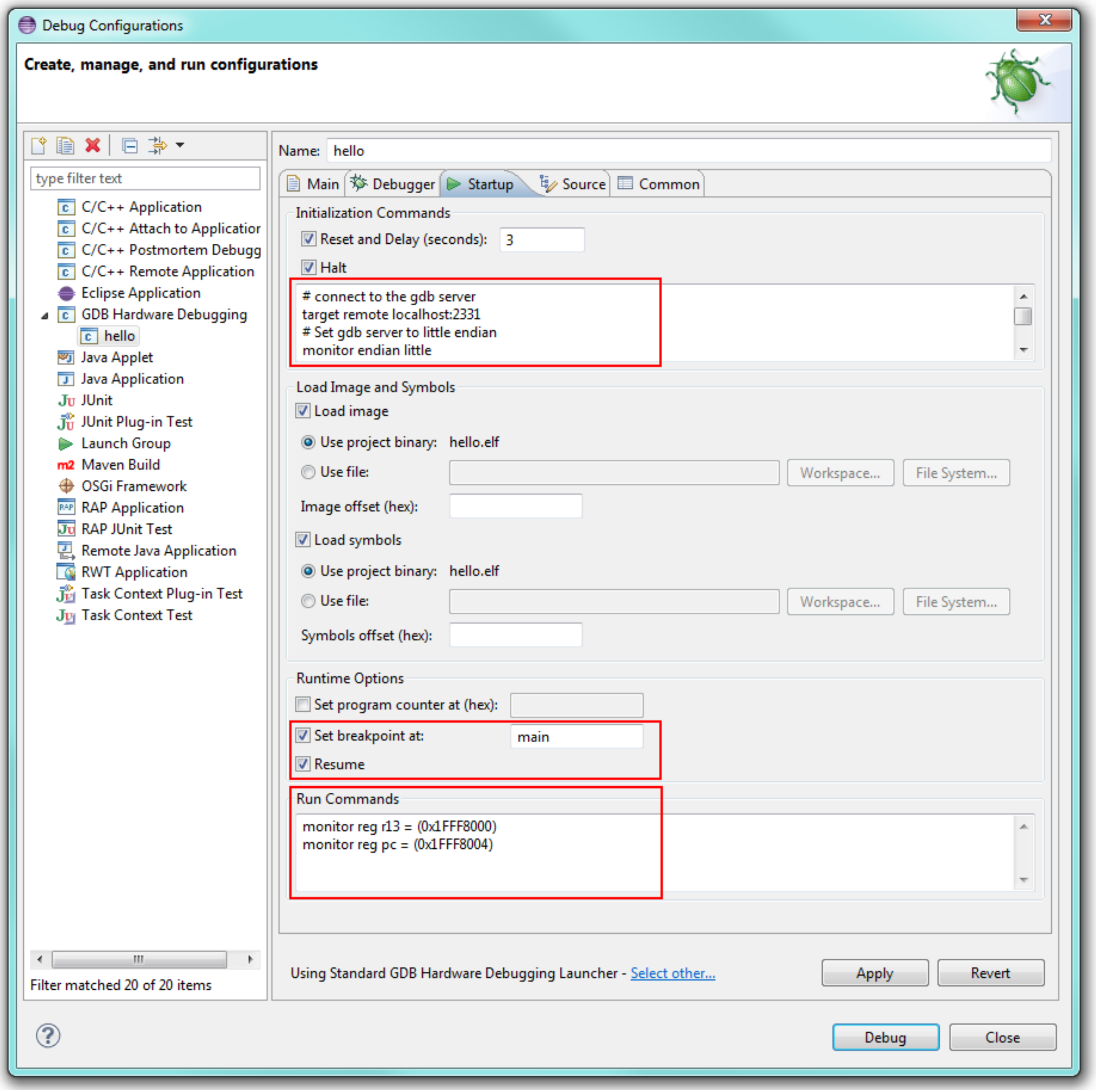
Note: Please update the GDB Server listening port. You may find it in the GDB Server information dialog – see the *Starting the GDB Server* chapter. In our case the port number is 2331.

Into the “Run Commands” text field paste the following content:

```
monitor reg r13 = (0x1FFF8000)
monitor reg pc = (0x1FFF8004)
```

In the “Runtime Options” set the initial breakpoint at the main function and mark the “Resume” option.

The final result should look like in the image below.


Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Applicatior
- C/C++ Postmortem Debugg
- C/C++ Remote Application
- Eclipse Application
- GDB Hardware Debugging
  - hello
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- Launch Group
- Maven Build
- OSGi Framework
- RAP Application
- RAP JUnit Test
- Remote Java Application
- RWT Application
- Task Context Plug-in Test
- Task Context Test

Filter matched 20 of 20 items

Name: hello

Main | Debugger | Startup | Source | Common

Initialization Commands

[✓] Reset and Delay (seconds): 3

[✓] Halt

```
# connect to the gdb server
target remote localhost:2331
# Set gdb server to little endian
monitor endian little
```

Load Image and Symbols

[✓] Load image

(●) Use project binary: hello.elf

( ) Use file:

Image offset (hex):

[✓] Load symbols

(●) Use project binary: hello.elf

( ) Use file:

Symbols offset (hex):

Runtime Options

[ ] Set program counter at (hex):

[✓] Set breakpoint at: main

[✓] Resume

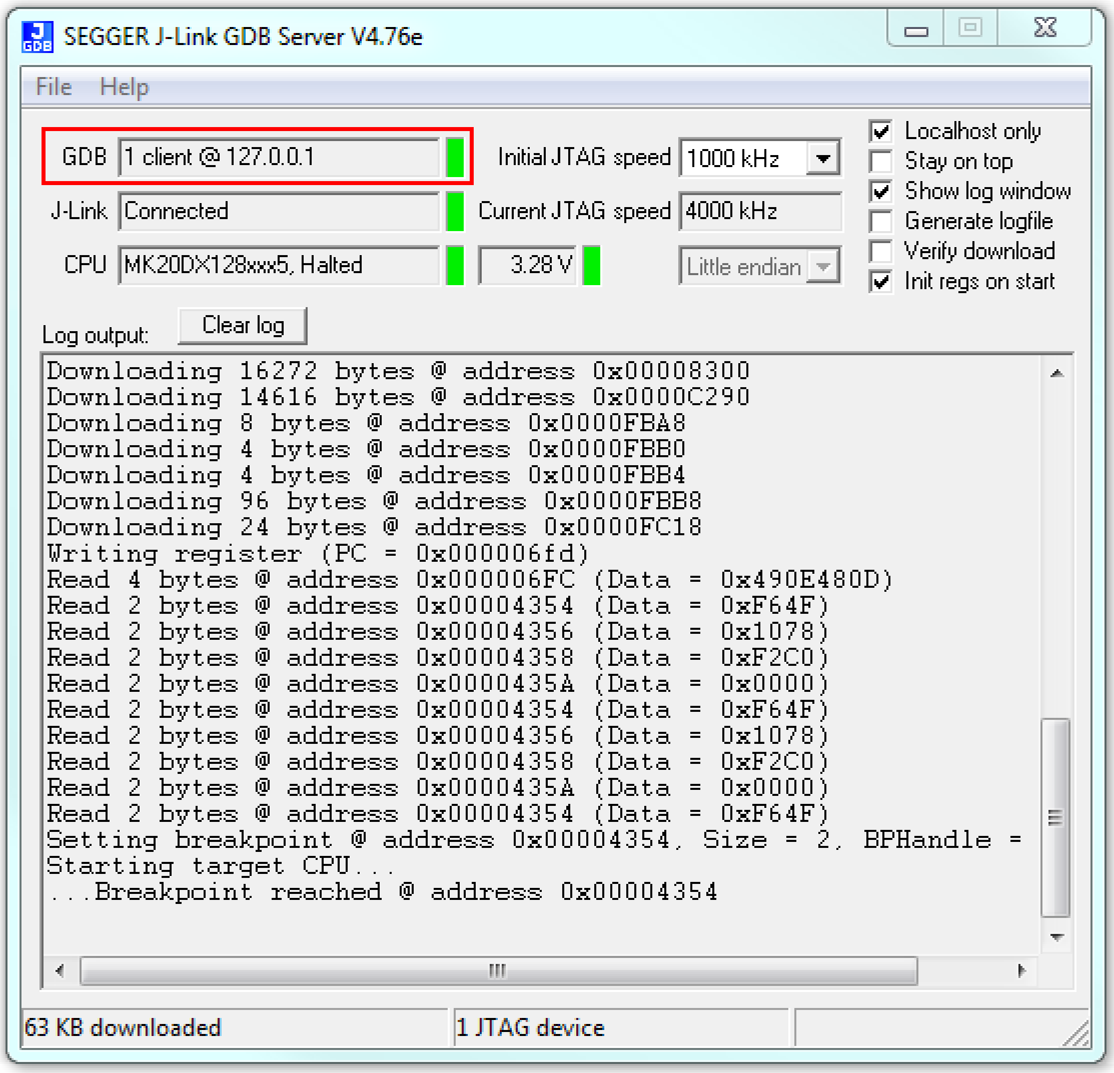
Run Commands

```
monitor reg r13 = (0x1FFF8000)
monitor reg pc = (0x1FFF8004)
```

Using Standard GDB Hardware Debugging Launcher - Select other...

Now you are ready to start the debugger, press the “Apply” button and then the “Debug” button. The GDB Server application should flash the microcontroller with your application and the debugger should stop at the main function.

The GDB Server application indicates the successful connection with the green light next to the “GDB” text field. It also contains the log output, which might be helpful if some problems have occurred.



GDB 1 client @ 127.0.0.1
J-Link Connected
CPU MK20DX128xxx5, Halted
3.28 V
Initial JTAG speed 1000 kHz
Current JTAG speed 4000 kHz
Little endian

[✓] Localhost only
[ ] Stay on top
[✓] Show log window
[ ] Generate logfile
[ ] Verify download
[✓] Init regs on start

```
Downloading 16272 bytes @ address 0x00008300
Downloading 14616 bytes @ address 0x0000C290
Downloading 8 bytes @ address 0x0000FBA8
Downloading 4 bytes @ address 0x0000FBB0
Downloading 4 bytes @ address 0x0000FBB4
Downloading 96 bytes @ address 0x0000FBB8
Downloading 24 bytes @ address 0x0000FC18
Writing register (PC = 0x000006fd)
Read 4 bytes @ address 0x000006FC (Data = 0x490E480D)
Read 2 bytes @ address 0x00004354 (Data = 0xF64F)
Read 2 bytes @ address 0x00004356 (Data = 0x1078)
Read 2 bytes @ address 0x00004358 (Data = 0xF2C0)
Read 2 bytes @ address 0x0000435A (Data = 0x0000)
Read 2 bytes @ address 0x00004354 (Data = 0xF64F)
Read 2 bytes @ address 0x00004356 (Data = 0x1078)
Read 2 bytes @ address 0x00004358 (Data = 0xF2C0)
Read 2 bytes @ address 0x0000435A (Data = 0x0000)
Read 2 bytes @ address 0x00004354 (Data = 0xF64F)
Setting breakpoint @ address 0x00004354, Size = 2, BPHandle =
Starting target CPU...
...Breakpoint reached @ address 0x00004354
```

63 KB downloaded | 1 JTAG device

5 Makefiles structure

In this document the following colors express dependencies on:

mqx/bsp	- is a directory of library or application
bsp	- is a name of library or application
twrk60n512	- is board name
gcc_arm	- is a toolchain name

5.1.1 Makefiles hierarchy

Library build process consists of partial makefiles:

/mqx/bsp/build/make/bsp_twrk60n512/Makefile

/mqx/bsp/build/make/bsp_twrk60n512/tools/gcc_arm.mak

/build/common/make/global.mak

/build/twrk60n512/make/tools/gcc_arm.mak

/build/common/make/verify.mak

/build/common/make/lib-process.mak

Application build process consists of partial makefiles:

/mqx/examples/hello/build/make/hello_ twrk60n512/Makefile

/mqx/examples/hello/build/make/hello_ twrk60n512/tools/gcc_arm.mak

/build/common/make/global.mak

/build/twrk60n512/make/tools/gcc_arm.mak

/build/common/make/verify.mak

/build/common/make/app-process.mak

5.1.2 Partial library makefiles

/mqx/bsp/build/make/bsp_twrk60n512/Makefile

is responsible to setup common SOURCES, INCLUDE paths and mandatory variables:

MQX_ROOTDIR	– path to mqx root directory
TYPE	– type of build, setup to “library” value
NAME	– library name
BOARD	– name of the board

LIBRARY_ROOTDIR	– rootdir of libraries builded for specific board and tool
LIBRARY_DIR	– path to library output directory
LIBRARY_FILE	– path to library output file
POSTBUILD_CMD	– macro to obtain post build command. Depends on HOSTENV

/mqx/bsp/build/make/bsp_twrk60n512/tools/gcc_arm.mak

is responsible to setup tool chain specific SOURCES and INCLUDE paths.

5.1.3 Partial application makefiles

/mqx/examples/hello/build/make/hello_ twrk60n512/Makefile

is responsible to setup common SOURCES, INCLUDE paths and mandatory variables:

MQX_ROOTDIR	– path to mqx root directory
TYPE	– type of build, setup to “application” value
NAME	– application name
BOARD	– name of board
LIBRARY_ROOTDIR	– rootdir of libraries builded for specific board and tool
APPLICATION_DIR	– path to application output directory
APPLICATION_FILE	– path to library output file
LINKER_FILE	– macro to obtain linker command file

/mqx/examples/hello/build/make/hello_ twrk60n512/tools/gcc_arm.mak
is responsible to setup toolchain specific SOURCES and INCLUDE paths.

5.1.4 Partial common makefiles

/build/common/make/global.mak
contains common macros, default definitions of TOOLCHAIN_ROOTDIR, HOSTENV variables.

/build/twrk60n512/make/tools/gcc_arm.mak
contain sub-paths to toolchain binaries, common flags and defines.

/build/common/make/verify.mak
perform existence verification of linker command files, toolchain paths and valid command line variables.

/build/common/make/app-process.mak
contain targets, rules and dependency to build an application

/build/common/make/lib-process.mak
contain targets, rules and dependency to build a library